Tampere University

Ville Siltala

MACHINE LEARNING OPERATIONS ARCHITECTURE IN HEALTHCARE BIG DATA ENVIRONMENT

Batch versus online inference

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
April 2023

ABSTRACT

Ville Siltala: Machine learning operations architecture in healthcare big data environment – batch versus online inference
M.Sc. Thesis
Tampere University
Master's Degree Programme in Computer Science
April 2023

Developing and operating machine learning systems is associated with uncertainties incomparable to traditional software engineering. Managing and mitigating these uncertainties is critical especially when creating machine learning systems for clinical healthcare use. By incorporating processes and tools to develop and deploy machine learning systems in a controlled, automated, and monitored manner, machine learning operations aims to ensure quality and reliability in machine learning systems.

This study provides an examination of machine learning operations in the context of healthcare and big data. First, a study project was conducted to design a machine learning operations architecture for building a machine learning based NLP solution to be integrated into an existing clinical healthcare software application. Two separate model deployment and inference architectures were designed. To test the applicability of these architectures in the context of big data, an empirical study was conducted. The results showed the batch inference architecture using Spark NLP had better performance compared to a Docker container based online inference architecture.

In conclusion, the study project involving the design of a machine learning operations architecture, as well as the empirical comparison of batch inference and online inference, offer insights into the field of machine learning operations. The proposed model and the results of the comparison can be used to develop machine learning systems and make informed decisions on the selection of an inference architecture.

Key words and terms: machine learning, machine learning operations, MLOps, natural language processing, NLP, big data, healthcare

TIIVISTELMÄ

Ville Siltala: Koneoppimisen tuotantoarkkitehtuuri terveydenhuollon massadatakontekstissa – vertailussa eräajo- ja reaaliaikainferenssi
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2023

Koneoppimiseen pohjaavien tietojärjestelmien kehittäminen ja operointi sisältää epävarmuustekijöitä, jotka eivät ole verrattavissa perinteiseen ohjelmistotuotantoon. Näiden epävarmuuksien hallinta ja lieventäminen on tärkeää etenkin luotaessa koneoppimisjärjestelmiä terveydenhuollon kliiniseen käyttöön. Koneoppimisen tuotanto (engl. Machine Learning Operations, MLOps) on joukko prosesseja ja työkaluja, joilla pyritään varmistamaan koneoppimisjärjestelmien laatu ja luotettavuus lisäämällä niiden kehitykseen ja tuotantoon automaatiota ja valvontaa.

Tämä tutkielma tarkastelee koneoppimisen tuotantoa terveydenhuollon massadatan kontekstissa. Ensin kuvataan tutkimusprojekti, jossa kehitettiin koneoppimisen tuotantoarkkitehtuuri NLP-sovelluksen (engl. Natural Language Processing) toteuttamiseksi. NLP-sovellus suunniteltiin integroitavaksi terveydenhuollon kliinisessä työssä käytettävään ohjelmistosovellukseen. Projektissa kuvattiin kaksi erilaista inferenssiarkkitehtuuria, joiden soveltuvuutta projektin kontekstissa testattiin empiirisellä tutkimuksella. Tulokset osoittivat Spark NLP -kirjastoon pohjautuvan eräajoinferenssin olevan tehokkaampi verrattaessa Docker-kontteihin perustuvaan reaaliaikainferenssiin.

Tutkimusprojektissa kehitetty arkkitehtuuri ja prosessimalli tarjoavat esimerkin koneoppimisen tuotantoarkkitehtuurista hyödynnettäväksi koneoppimisjärjestelmien kehityksessä ja tuotannossa. Inferenssin suorituskykyä vertailevan empiirisen tutkimuksen tuloksia voidaan hyödyntää valittaessa inferenssiarkkitehtuuria tietoon pohjautuen.

Avainsanat: koneoppiminen, koneoppimisen tuotanto, MLOps, luonnollisen kielen käsittely, NLP, massadata, terveydenhuolto

LIST OF ABBREVIATIONS

ABAC	Attribute-based Access Control
ACI	Azure Container Instances
AI	Artificial Intelligence
AIM	Artificial Intelligence in Medicine
API	Application Programming Interface
CD	Continuous Delivery or Continuous Deployment
CI	Continuous Integration
CPU	Central Processing Unit
DevOps	Software Development (Dev) and Software Operation (Ops)
EHR	Electronic Health Record
GB	Gigabyte
GPU	Graphics Processing Unit
HIS	Healthcare Information System
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
ML	Machine Learning
MLOps	Machine Learning Operations
NER	Named Entity Recognition
NLP	Natural Language Processing
NLTK	Natural Language Toolkit
REST	Representational State Transfer
RESTful API	A HTTP API using principles of REST
SE	Software Engineering

Contents

1 Introduction

In recent decades, healthcare practice has moved from documenting patient records and clinical notes on paper to using healthcare information systems (HIS) storing data in electronic health records (EHR). This development has led to the accumulation of digital healthcare data. It has also brought the need, and possibility, for using machine help to organise, search, and develop novel ways of using data to support clinical and administrative work in the healthcare domain.

Significant portion of healthcare data is in the form of natural language in medical reports written by healthcare professionals. To make this unstructured data machine readable, we need to transform unstructured natural language to some structured data format. Even though modern healthcare information systems are moving towards providing tools for clinicians to make reports in structured format, it will not help dealing with legacy data created with legacy software. For this reason, creating solutions to process and analyse both historical and new data is key in getting insights from the vast amounts of healthcare data.

To enable machines to work with unstructured natural language, we need natural language processing (NLP). Combined with machine learning (ML) methods, NLP has proven capable of processing and transforming vast amounts of data into structured formats usable for analysis use cases. Named entity recognition (NER) is one of these methods. In practice, NER consists of an ML model that is trained to recognise valuable entities from natural language. The context of this thesis is to study the life cycle processes of implementing such ML solution in the context of healthcare big data.

At the end of the ML life cycle is a productised ML solution. Reaching a functional, reliable, and efficient ML solution that is integrated to existing information systems, is not trivial. Despite the great advancements in ML research in recent decades, ML productisation has proven to be a difficult task not least in the healthcare domain. Compared to traditional software, ML systems bring a new realm of uncertainties not manageable with traditional software development methodology. One proposed solution to ease the process of ML life cycle management and ML system productisation is machine learning operations (MLOps). The ambition behind MLOps is to breach the gap between data scientist work in developing ML models and a functioning ML solution by bringing automation and transparency to the various phases of ML system development.

The goal of this master's thesis is to do a case study in MLOps in the context of productising an ML powered NLP solution in healthcare big data context. To address the challenges and opportunities in this domain, two research questions have been formulated, as presented in Table 1.

Research question 1 (R1) seeks to identify the MLOps architecture and processes required for an ML NLP product. This question is framed with the case study project described in Chapter 3. By examining the specific needs and demands of ML NLP products, the thesis aims to propose a comprehensive model for the MLOps processes and tooling architecture.

Research question 2 (R2) investigates the optimal cloud computation architecture for executing inference with a deep learning ML NLP model in the healthcare big data context. To address this question, the thesis proposes two distinct architectural solutions, each offering unique advantages and capabilities. To test the applicability of the proposed solutions, an empirical study was conducted by executing the inference with various computation resources within environments created according to the proposed inference architectures.

R1	What type of MLOps architecture and processes an ML NLP product requires?
R2	What type of cloud computation architecture is optimal for executing inference with a deep learning ML NLP model in healthcare big data context?

Table 1. Research questions.

Next, in Chapter 2, the necessary background information is provided, beginning with an overview of machine learning and its applications in healthcare. The challenges in developing machine learning systems are discussed, followed by an exploration of the need for machine learning operations and a definition of the machine learning operations process.

In Chapter 3, a detailed MLOps case study is presented, which includes the project description, requirements, and preconditions. A comprehensive plan for the MLOps process and architecture is proposed. The chapter ends with describing the empirical study to test and compare the performance of the proposed inference architectures for batch and online inference.

Chapter 4 focuses on the analysis of the inference performance study results, discussing the findings from the online and batch inference tests, and comparing these results to evaluate the effectiveness of the proposed solutions.

Chapter 5 presents the conclusions derived from the primary findings of this thesis. Following that, in Chapter 6, the implications of the research are assessed, along with the exploration of possible future directions for research in this domain.

2 Background

In this chapter, a brief introduction to research touching on this thesis subject is given. Section 2.1 gives a general view on ML. Section 2.2 describes the advances of ML research in the field of healthcare. Section 2.3 is about the challenges of developing modern ML systems. Section 2.4 goes further in describing the need for MLOps in solving the challenges described in Section 2.3. Finally, Section 2.5 gives an introduction to studies defining the MLOps process.

2.1 What is machine learning?

In the era of big data, getting insights from the vast amount of data gathered in information systems requires machine help. Machine learning, a part of artificial intelligence (AI), is defined as the means for automatically detecting patterns in data and using this gained knowledge to predict the future or to make decisions under uncertainty [Murphy, p. 1, 2012]. The "knowledge" is engineered, or "trained", into a system by creating a program that fits the given data by learning the patterns and characteristics in the data. The process of "learning" happens when a learning algorithm is adjusting modifiable parameters in a system template, i.e., an ML model, based on the data passed through the algorithm [Alpaydin, p. 24-25, 2016]. By learning the patterns in vast amounts of data, the ML model can be used to predict the probability of a given sample having certain characteristics. For example, this can be classifying a sample image by determining the probability of it containing a certain object, e.g., a dog. It can be said the ML model has then learned to recognise the "dogness" of images. The amount of "dogness" can then be used to infer if the image contains a dog and with what probability.

There are three main types of ML: supervised learning, unsupervised learning, and reinforcement learning [Murphy, p. 2, 2012]. This thesis focuses on the usage of most common of them, supervised learning, more precisely, deep learning. A deep learning architecture consists of a multilayer stack of modules capable of transforming the input to an output with increased invariance and selectivity [LeCun *et al.*, 2015]. A key part in these architectures is the individual nodes in each module that are often described as neurons. When we combine these layers of nodes, or neurons as we say, we form an artificial neural network. The usage of big data and increasing computing power has enabled building and increasing the complexity of these deep neural networks.

LeCun *et al.* [2015], in their review, describe how deep learning has significantly advanced the field of ML in the past years. For decades, ML was limited by the ability of data scientists to extract and transform the domain specific features of the given data into such format that was usable for ML algorithms commonly (linear) classifiers. Deep learning and its capability to recognise patterns in multi-dimensional data has been breaking records in various ML domains from speech recognition to gene mutation predicting. Compared to "shallow" linear classifiers, which require extensive engineering work and domain expertise on feature extraction, deep learning enables learning intrinsic features automatically from multi-dimensional input data. The jump from linear classifiers to multi-dimensional classifying with neural deep learning has broadened the capabilities of ML and accelerated its development.

2.2 Machine learning in healthcare

AI and ML has been researched within healthcare context since the 1970s [Patel *et al.*, 2007]. From the initial promise, or threat some might say, of AI and ML systems once being able to replace human clinicians in their work, the progress of incorporating AI into clinical work has been slow. Patel *et al.* [2007] gave a view on the history of artificial intelligence in medicine (AIM). They point out that the barriers for AI breaking into clinical use has not only been technical, but also political, fiscal, and cultural. Yet, the research of AIM has made progress in the past decades and is comparable to the overall advancement of AI and ML. From the traditional ML methods, where the domain expertise of statistical modelling and feature extraction was key, focus has shifted to deep learning and its opportunities of building systems capable of learning key features and patterns automatically from vast amounts of data [Beam and Kohane, 2018]. For example, using deep learning models to build diagnostic tools has been proven capable of detecting referable diabetic retinopathy with accuracy comparable or greater than human clinicians [Gulshan *et al.*, 2016].

In their review, Jiang *et al.* [2017] dove into used data types and mechanisms, and the disease types in AIM research. The data sources used in AIM can be divided into structured and unstructured data. Structured data includes diagnosis imaging, genetic testing and electrodiagnosis. These data sources are researched especially in the diagnosis stage. Jiang *et al.* listed clinical laboratory results and physical examination notes as the main unstructured data sources under research. These data types require some mechanism of transforming the data from unstructured form, such as human written text, into a machine-understandable form.

Natural language processing is an AI mechanism that can be used for transforming natural language into structured form. After this transformation the created structured data can be used with various ML methods. Jian *et al.* noted that with structured data, researchers have aimed at clustering patient traits or inferring disease outcomes using various ML techniques. The three main disease categories researched were identified as cancer, neurology, and cardiology. All three are among the leading causes of death and thus improving diagnostic tools and treatment outcome prediction with AIM could benefit numerous patients.

While HISs are moving towards structured formats and standardisation of data models used to store EHRs, vast amount of data is still aggregated and archived in free-text format, e.g., in medical narratives. Transforming this data into machine readable format requires medical NLP. One of the main goals for medical NLP is to build tools for clinical decision support [Demner-Fushman *et al.*, 2009]. In clinical decision support, medical NLP could extract information from EHRs containing free-text data. This extracted data transformed into machine readable format could then be used to build various kinds of clinical decision support systems. The use of machine learning NLP, especially deep learning models, has already been demonstrated capable of medical adverse event detection [Borjali *et al.*, 2021], adverse drug event detection [Chen *et al.*, 2019], and predictive diagnosis of venous thromboembolism using semantic and sentimental analysis [Sabra *et al.*, 2018].

As AIM keeps advancing and more and more use cases open for AI tools to be incorporated into clinical practices and HIS, it is clear we need standardised ways of building trustworthy AI. In healthcare, a highly legislated and controlled field, this is even more essential, e.g., to ensure patient security. Thus, there is a clear need for systematic methods for developing AI and ML systems to enable taking ideas and researched methods into production.

2.3 Challenges in developing machine learning systems

The development, deployment, and operation of machine learning systems is a process that holds unique characteristics compared to traditional software engineering (SE) [Kumeno, 2019; Amershi *et al.*, 2019]. The phases of an ML system project, i.e., ML workflow [Lwakatare *et. al*, 2020], can be distilled down to context understanding, data curation, data modelling, and production and monitoring [Wan *et al.*, 2021].

The first step in an ML project is context understanding. This phase requires the ML engineering team to interact with the customer to discover and specify the possibilities of ML applications. The data curation process consists of collecting and pre-processing the data. This includes manually labelling a ground truth dataset to be used for training or validating the model. Data modelling is the process of training and evaluating the model using the features extracted from the data. The final step of data modelling is evaluating the accuracy of the model output. In the production phase, the model is deployed to be used as a service integrated into existing information systems. The model performance is then tracked by monitoring changes in the input data and the correctness of the model output. [Wan *et al.*, 2021]

As described above, data plays a key role in ML application development phases. It has been said [Arpteg *et al.*, 2018] that data replaces parts of code in an ML system compared to traditional software systems. Data is used to train an ML model for the system to be able to automatically recognise patterns. In traditional SE, behaviour of a system is implemented with hard-coded rules [Arpteg *et al.*, 2018]. Recognising patterns from the data using a model produced in the training phase by learning algorithms imposes uncertainty to ML systems. Compared to the hard-coded, strictly defined rules in a traditional software, ML models only give an approximation of the relationship between the model input and the inferred output [Kläs and Volmer, 2018]. Despite this inaccurate nature of ML systems, in various cases approximation is the only viable solution. When the behaviour of a system cannot be expressed in traditional software logic, e.g., recognising an object from an image, there is a need for ML [Sculley *et al.*, 2015]. In these cases, constructing an ML model to approximate the behaviour can produce results good enough for real-life use.

Some level of uncertainty in the behaviour of an ML system is inevitable. Training models from data introduces an empirical aspect and the behaviour of an ML system is always linked to the data available. There is no guarantee of the real-world correctness of an ML system [Kläs and Volmer, 2018]. In addition to data related issues, uncertainty and incorrectness of ML systems is also driven by technical debt incurred by multiple aspects of the ML system lifecycle [Sculley *et al.*, 2015]. Sculley *et al.* list *boundary erosion, entanglement, hidden feedback loops, undeclared consumers, data dependencies, configuration issues, changes in the external world, and a variety of system-level anti-patterns* as main causes for technical debt. In traditional software development, building modular decoupled systems mitigates technical debt by making systems and their codebase more maintainable and easier to update. An example of this is the rise of microservice architecture where decoupled software components implement a subset of wider business logic. These components can be developed, deployed, and maintained in isolation. On the other hand, in ML systems signals are entangled and boundaries between modules erode. For example, features used to train an ML model are hard to develop in isolation since they influence the results of the ML model in conjunction. Changing the distribution of a single feature could affect the performance of the model in a negative way. Sculley *et al.* call it the *CACE principle: Changing Anything Changes Everything*. Though traditional SE methodologies and practices have contributed to ML lifecycle processes, they fit poorly because of the fundamental differences in traditional software and ML systems [Kumeno, 2019].

Kumeno [2019] divides the ML lifecycle into requirements analysis, data-oriented works, model-oriented works, and DevOps works. Requirement analysis consists of analysing the system requirements and data. Data-oriented works involve data collection, data cleaning, data labelling and feature extraction. Model-oriented works consist of model training, model evaluation, model optimisation and model design and construction. DevOps works include model deployment, monitoring the system performance and possible retraining iterations. These phases create a feedback loop where any of the steps can loop back to the previous stages if the expected outcome is not achieved. Compared to traditional SE, it is hard to predict success of each phase of the ML lifecycle thus making it difficult to estimate the full cost and duration of an ML project. The feedback loop might need multiple iterations before the ML solution reaches a level of acceptable accuracy. With the uncertainty related to the data available, it is also a possible, that a goal accuracy level is impossible to achieve without going back to the initial phases of requirement analysis followed by data exploration, gathering, and pre-processing.

To minimise the uncertainty in ML projects, automating and improving the ML lifecycle processes have been proposed as a solution. Zhou *et al.* [2020] recognize this need for efficacy and reliability in building and maintaining ML systems. The complexity and the iterative nature of ML lifecycle described by Kumeno [2019] and Amershi *et al.* [2019] call for automation to reduce uncertainty and the possibility of human error in the ML development processes.

2.4 Why we need machine learning operations?

The traditional view of ML development has been described as a process of data scientists performing scientific experiments in isolation [Valohai, 2022]. This data science work revolves around notebook interfaces allowing agile data manipulation and prototyping of ML models but lacking in tools and processes to take these prototypes into production-ready ML solutions [Borg, 2021]. On the other hand, in modern software development, operations are tightly linked to development processes [Lwakatare *et al.*, 2020] adhering to DevOps principles [Karamitos *et al.*, 2020]. The DevOps workflow creates a fast and automated feedback loop where agile software updates are continuously integrated and deployed to end-environments. As of late, this process was not well defined in ML workflow [Lwakatare *et al.*, 2020].

The development of ML solutions has been traditionally conducted by data scientists using ML workflow practices. These new type of software professionals are often lacking in training and experience in traditional software development methods [Borg, 2021]. This issue is highlighted by the fact of ML workflow missing the processes required by modern software development, e.g., continuous development and delivery practices [Lwakatare *et al.*, 2020]. As building ML tools becomes more and more prevalent in the software industry, the need for improving ML software development practices increases. This need is also gaining recognition in the field of ML software professionals [Mäkinen, 2021].

Machine learning operations (MLOps) does not include only the DevOps process for building ML systems. There is also an architectural angle to MLOps. As more and more AI and ML tools reach production phase, the need for integrating ML systems to traditional software systems increases. Designing such integration services that can run efficiently and reliably complex neural network models, is not trivial. The process or "running" an ML model is called inference. In this process a single sample or a batch of samples are fed into the model and the model outputs, i.e., the inference results, are served back in machine readable format. The samples are passed to this process via some sort of application programming interface (API). Setting up the API requires deploying the model into the API service.

The model API service is often implemented using a representational state transfer (REST) architecture. A deployed ML model is then served as a RESTful API and the inference occurs behind the Hypertext Transfer Protocol (HTTP) API. The API receives a single item or a batch of items as the body of a HTTP request. The submitted entries are processed with the ML model to receive the designed output to be served back via the HTTP response. Designing such service architecture that fits this process well, is not well established. In this thesis, part of the MLOps process is to design the inference architecture to solve this problem in a healthcare big data context.

2.5 Defining the machine learning operations process

Previously described challenges in developing and operating ML solutions led to research in closing the gap between these two processes. Lwakatare *et al.* [2020] describe the integration of ML workflow and the DevOps process. Even though they are not referring this integration as MLOps, this can be viewed as a description of it. Their goal was to describe a process that enables *"fast, iterative and continuous development, deployment and operation of AI-based software systems in production"*. In their view, the ML system lifecycle can be described in four distinguishable processes where DevOps should be integrated into ML workflow activities: data management, ML modelling, software development, and system operation. From this point on, the term MLOps is used to describe this integration.

In the data management process, MLOps include creating and using standardised activities and systems for data collection, selection and augmentation, and feature extraction. The code, tools and systems created for data management should be reproducible and support continuous integration and delivery methods. Security is also a key factor when dealing with the data, and MLOps should take secure data access into account. [Lwakatare *et al.*, 2020]

Within data modelling, MLOps should allow for backwards traceability from developed models to training experiments, used data sets and version-controlled code used in training processes. In simple terms, each produced artifact should be stored with related metadata including a name, version, registration date and dependency information. The storage for successfully trained and accepted models and their metadata is called a *model registry*. A registry should contain such information of stored artifacts that it enables reproducing an artifact using the registered dependencies. [Lwakatare *et al.*, 2020]

ML components need to be integrated into traditional software. MLOps provide tools and processes for this. The registered models should be integrated via automatic build systems with verification and integration testing in place. The testing should ensure that the traditional software components work with the ML system using large enough test data. This process can be automated on a level that enables automatic retraining of the model and integration of new model versions into the existing software. The continuous integration system can then be used to rollback to previous model versions to identify bugs and errors surfaced after model updates. [Lwakatare *et al.*, 2020]

In the operational phase of ML system lifecycle, the running system should be continuously monitored for issues. The system should perform consistently over time regarding the model performance (e.g., the prediction accuracy) and execution latency (i.e., inference execution time). Operational stage can include manual, semi-automated or automated process of (re)deploying the ML components to the system. Version deployments and the new component versions should be tested in preproduction environments. [Lwakatare *et al.*, 2020]

Karamitsos *et al.* [2020] also point out the gap between data science work and taking ML systems into production with automated DevOps processes. They set out to design an ML pipeline using continuous integration (CI) and continuous delivery (CD), principals known from DevOps [Fitzgerald & Stol, 2017]. They propose a streamlined, automated process they call *ML automated pipeline with CI/CD*. Automation is required when there is a need for continuous training and testing of models. In the production phase, updating served models requires a reliable, automated deployment pipeline including testing in various stages of the pipeline. This process allows the ML system to adapt to changes in the data or requirements for the models.

First Karamitsos *et al.* [2020] describe a manual process for ML pipeline. This manual process is stated as a baseline for an ML system delivery project and assumes that it is used for small-scale ML projects where no, or only few, iterations of the implementation is required. Compared to an automated ML pipeline leveraging CI and CD principles, in a manual ML pipeline the tasks from developing and testing an ML model up to the point of deploying and operating it are manual. In an ML manual pipeline process an ML system is delivered by separate teams of data scientists developing the model and registering it into a model registry via an API and then the operations engineers testing and operating the model retrieved from the registry. Since the ML pipeline is a one-off type of implementation, no tracking of the model results or actions is required. Machine learning manual pipeline process is shown in Figure 1.

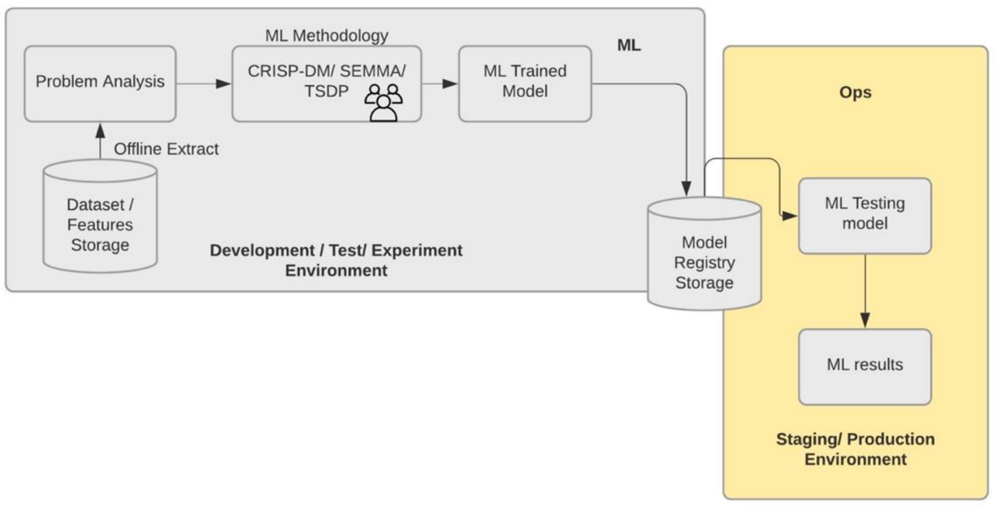


Figure 1. Machine learning manual pipeline process. [Karamitsos *et al.*, 2020]

The *ML automate pipeline for CI/CD* defines a process for streamlining the development of ML models for data scientists and ML engineers. Enabling a CI/CD process for model development enables faster lifecycle of taking a model from the development phases of feature engineering, model architecting and hyperparameter optimisation to deploying the model into a test environment for validation and ultimately into production. It also enables retraining the model in production whenever new data is available. [Karamitsos *et al.*, 2020]

The automated pipeline is shown in Figure 2. ML automate pipeline for CI/CD uses Git repositories for source code management. Jenkins is used to automate the CI pipeline. Jenkins is an open-source automation server [Jenkins, 2023]. Once Jenkins receives a trigger, i.e., a Git repository change, the CI pipeline validates and builds the latest version of the ML code and runs unit tests for it. The ML code is then used to train a ML model. Along with changes in the source code repository, the training can be triggered with new data or adjusted hyperparameters. The trained models are built into a complete ML service using a CD pipeline that builds and registers a Docker image of the service. Docker is an open-source platform that enables the creation, deployment, and running of applications in self-contained and executable packages, i.e., containers [Docker, 2023]. The Docker image is deployed to a staging environment where the service container containing the newly trained model is automatically tested and validated against a test dataset for accuracy and performance. The CD pipeline is finalised with deploying an accepted ML model service into production that is running in a Kubernetes cluster. Kubernetes is a popular open-source system designed to automate the deployment, scaling, and management of containerized applications [Kubernetes, 2023]. The CI/CD process can be rerun to retrain a model if model or data drift is detected when monitoring the performance of the production ML service.

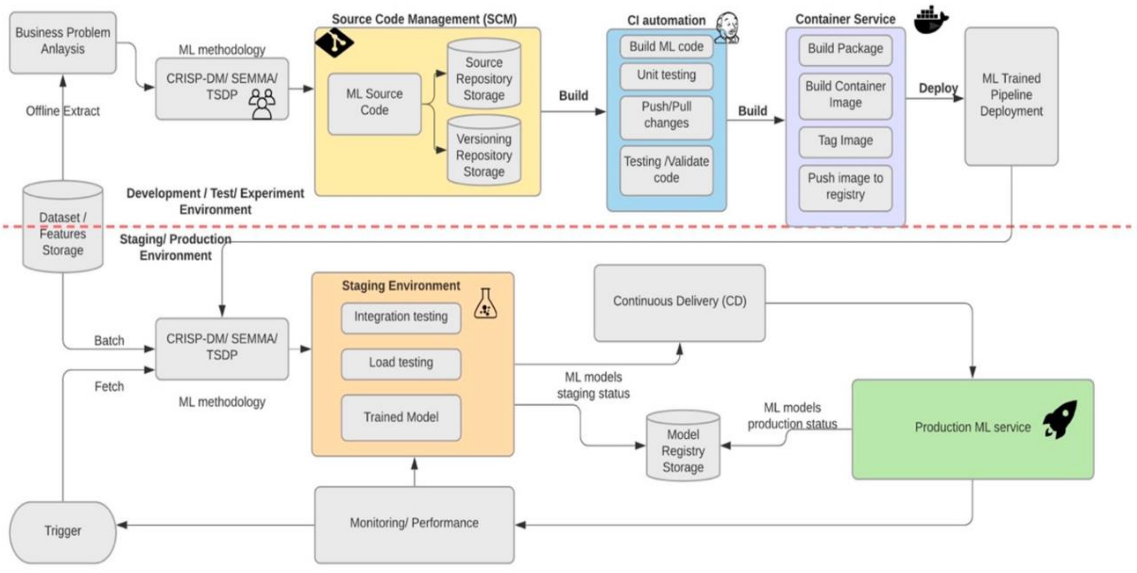


Figure 2. ML automate pipeline for CI/CD [Karamitsos et al., 2020]

The end-result of a CI pipeline is packaged components (e.g., Docker containers), containing the trained and tested model ready to be delivered. Testing should include unit testing the code used for training and building the models as well as testing the performance of the model using test data separated from the data used for training the model. Testing should also incorporate validating data, data schemas and models. The CI process should rely on version-controlled code and data used for building the models. This enables backwards traceability and reproducibility of models. Testing should validate that the models give sufficiently accurate predictions of the real-world phenomena. [Karamitsos *et al.*, 2020]

Before delivering the built components to production environment, the components should be deployed to a preproduction environment to be tested in a production like environment. Models passing this integration test phase are pushed and stored into a model registry. In the CD process, the stored model is then delivered to the production environment to be operated and monitored. [Karamitsos *et al.*, 2020]

Van der Goes [2021] described five best practices for MLOps: pipeline automation, data availability, exchangeable artifacts, observability, and policy-based security. Pipeline automation is achieved with CI/CD processes. Ensuring data availability means storing validated data sets used for training specific model versions to allow reproducibility. Exchangeability of artifacts means using version controlling and descriptive documentation for ML models, code, and configurations. This also includes documenting architectural solutions for integrating deployed models into traditional software systems. With observability, MLOps should aim to achieve monitoring of ML system component performance and fault tolerance by using performance metrics and logging of successful and erroneous events. Policy-based security should cover all four release environments: development, testing, acceptance, and production. Each environment should have security policies in place and allow access with attribute-based access control (ABAC).

As described in the literature, MLOps are a set of processes to be harnessed in the ML lifecycle. Various tools and systems have been built around MLOps to ease the automation of these processes. The MLOps process described in the case study in Chapter 3 is utilising Microsoft's Azure services and its ML life cycle service Azure Machine Learning.

3 MLOps case study

This chapter describes the research and development project for an MLOps process for a deep learning ML NLP solution in healthcare big data context. The provider of project is a Finnish IT software and service company, Tietoevry [Tietoevry, 2022a]. The project was executed as an in-house research and development project during the end of 2021 and the spring of 2022.

Section 3.1 gives a brief description of the project and its requirements. Section 3.2 frames the preconditions for the planned NLP solution and its MLOps process. In Section 3.3, the designed MLOps architecture is described and two alternatives for the inference stage are proposed: online inference and batch inference. Section 3.3 ends with a description of the empirical study on inference performance.

3.1 Project description and requirements

In the research project, the goal was to design an MLOps process for developing and productising an NLP solution using deep machine learning for analysing free text contents in EHRs. The aim was to build a process for training, evaluating, deploying, and maintaining a Finnish BERT model, "FinBERT" [TurkuNLP, 2019], fine-tuned for the task of named entity recognition in medical texts. The NER model would be trained to recognise valuable entities from EHR texts, e.g., mentions about a patient's smoking or other substance usage. The NER results would then be used to create an improved search tool for clinicians viewing EHRs in clinical work. A simplified process of the NER model's func-

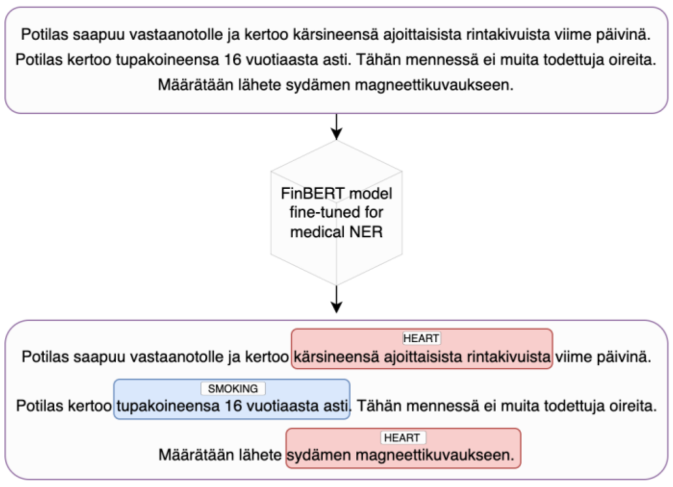

Potilas saapuu vastaanotolle ja kertoo kärsineensä ajoittaisista rintakivuista viime päivinä. Potilas kertoo tupakoineensa 16 vuotiaasta asti. Tähän mennessä ei muita todettuja oireita. Määrätään lähete sydämen magneettikuvaukseen.
FinBERT model fine-tuned for medical NER
HEART
Potilas saapuu vastaanotolle ja kertoo kärsineensä ajoittaisista rintakivuista viime päivinä.
SMOKING
Potilas kertoo tupakoineensa 16 vuotiaasta asti. Tähän mennessä ei muita todettuja oireita.
HEART
Määrätään lähete sydämen magneettikuvaukseen.

Figure 3. The NER model inference input and output for an EHR.

tioning, demonstrated using Finnish language examples, is shown in Figure 3. In the example, the model is capable of recognising mentions about smoking and information related to heart conditions and treatment in a simple EHR containing free text.

The designed ML NLP system should be capable of handling big data stored in a customer's environment, e.g., a cloud environment of a Finnish hospital district, without the need for the data to be delivered or accessed outside the environment. The solution should be easily deliverable to multiple customers. Artifacts, e.g., ML codebase and trained models, created within the MLOps processes should be stored and hosted in the provider's own environment for maintaining intellectual property rights.

The final NLP solution should be integrated with a patient record viewer application called 360° Patient produced by Tietoevry. 360° Patient [Tietoevry, 2022b] is an application built with web technologies. It enables clinicians to view EHRs from both archived and current HISs by integrating the HIS data stored in a data lake, a data storage technology capable of storing big data in structured, semi-structured an unstructured format. The integration to the data in the data lake is implemented by creating data pipelines that extract and transform the raw structured and unstructured EHR data stored in the data lake and load it to the application database. To enable clinicians efficient use of patient data, 360° Patient provides advanced search tools such as free text search and document filtering using various clinical taxonomies.

The NLP solution is intended to enhance the search functionality in the application by providing search labels for frequently searched topics. For example, a clinician evaluating a patient's applicability for a medical procedure could search all mentions about patient's smoking status. Using a deep learning NLP model in recognising the related entities in the texts would cover various forms of expressing the smoking status of a patient and thus improve search accuracy compared to a key-word search performed by clinicians. The 360° Patient application's patient health record view is displayed in Figure 4.

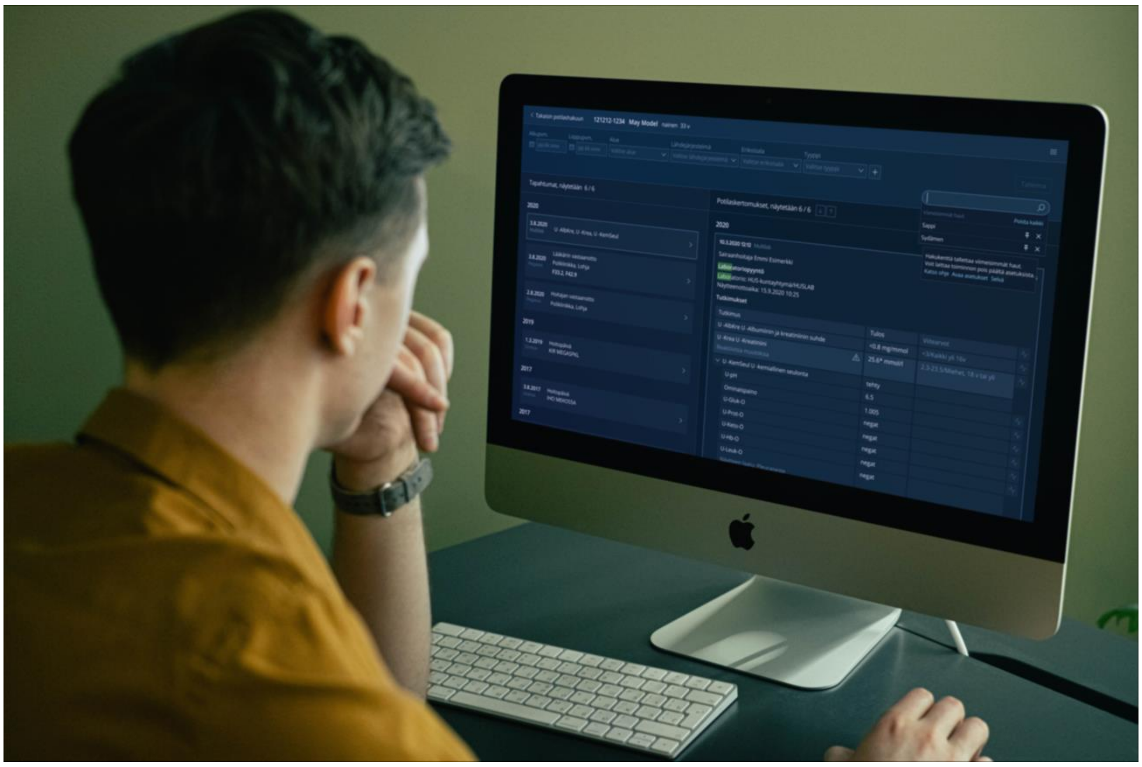


Figure 4. A clinician viewing a patient’s medical history in the 360° Patient application.

The data used in the 360° Patient application comes from data lakes built for Finnish medical districts. The data is ingested with integrations built for various EHR sources. Usually, some of the data is historical and brought to the lake with a single load. Some of the data can be updating in batches periodically and some might be streamed continuously. All these data intake processes should be considered when designing the MLOps architecture. New data should be available for the clinician as soon as possible and the inference process should conform to this need, i.e., new data should be run through the model’s inference and results should be available for clinicians simultaneously as the new data is visible in the application. The MLOps process should also give tools for monitoring changes in the data and logging all events from training to inference stage. The ML system should also support monitoring data drift and notify about performance decline caused by drifting in the input data, e.g., due to changes in clinical recording practices.

3.2 Preconditions

The 360° Patient application gives a window to data stored in data lakes owned by Finnish medical districts. The largest Finnish medical district, HUS, hosted a seminar for clinical NLP [HUS, 2022] on February 4, 2022. It was noted that HUS data lake contains 9 800 million words in patient texts and a total of 11 billion words when all document types in the lake are combined. Presumably, data lakes are expected to expand over time, resulting in an increase in the volume of text data. Therefore, the ML solution proposed in this project must be capable of efficiently and affordably processing such large volumes of data.

The patient records used in training the model and viewed in the application are regulated by Finnish public health law and Finnish special healthcare law. Thus, any unnecessary access to raw data containing personal patient information should be limited. To support this requirement, development and testing environments should only be accessing data that is pseudonymised. Manual labelling any training or validation data sets should be performed by approved clinicians in a closed environment with appropriate access restriction policy in place. As stated above, no data should leave the customer's environment in any stages of the MLOps process.

The end-result of the inference, i.e., the model output data, is viewed in the 360° Patient application. As stated above, the application is built with web technologies. Thus, serving the model output for the end-user should have a latency that is acceptable in a web client environment. To reduce latency of consecutive inference requests, some sort of caching mechanism should be considered. It should also be noted that the clinician only views records of a single patient at a time. Thus, the maximum amount of text to be processed with the model for a single view is the amount of text for a given patient. This should be noted when designing and implementing an online inference architecture. It is assumed that maximum inference request latency occurs when a clinician is viewing contents for a patient with no cached inference results for any of the stored health records. This initial inference request is what defines the compute resource requirements for the online inference API. It is also assumed that there are multiple simultaneous users viewing patient records at a given time. The built service should handle concurrent requests without noticeable performance hindrance in the application.

3.3 MLOps process and architecture plan

To answer research question 1, the MLOps processes and the architecture plan is described in this section. The aim is to give a high-level description of the MLOps architecture, and the tools required for implementing ML lifecycle tasks from training to productising the developed model. First, an overall description of the planned ML product lifecycle is given. Then, the MLOps tools and processes required in different lifecycle phases are defined. Finally, the architecture plan for each phase is described in more detail.

The goal of the MLOps design project was to develop an MLOps process that is automated and conforms to DevOps principals, e.g., by using CI and CD tools for developing, testing, and productising the trained models. As the cloud service provider, Microsoft's Azure cloud platform [Microsoft, 2022a] was selected. As the ML lifecycle tool, Azure Machine Learning [Microsoft, 2022b] was selected. It provides an end-to-end platform for ML development.

ML product lifecycle stages identified for the project are displayed in Figure 5. The project begins with the model requirement analysis phase where ML product requirements and goals are defined. The provider identifies domain specific goals for the NLP system together with domain experts from the customer side. In this NLP project, the NER classes would be defined and requirements for the model accuracy would be established.

The next phase in the ML lifecycle is the model development phase. This phase begins with managing the data required in the model development including data exploration, preparation, and transformation. Data modelling can begin once the required data pipelines are built, and the data is available. The output of this stage is a model training pipeline codebase that can be used to automate the training phase.

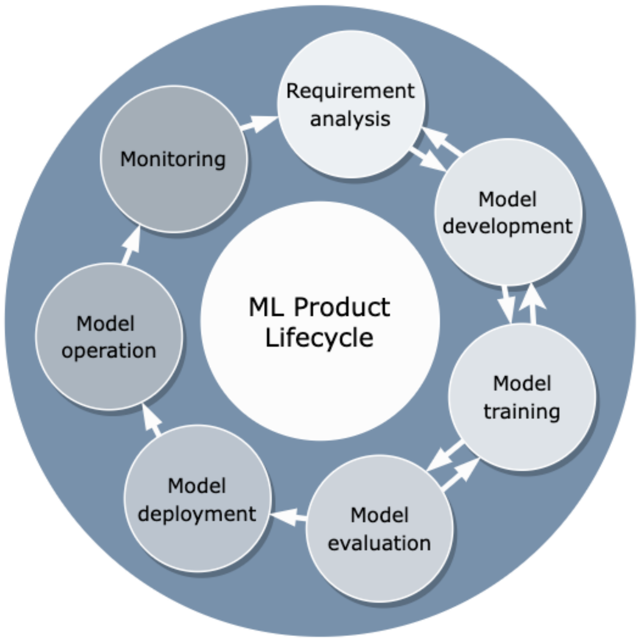


Figure 5. ML product lifecycle.

The next stage is the model training phase where the training pipeline infrastructure is provisioned in the cloud. A CI process is used to trigger the training pipelines. The goal in this phase is to train a model that reaches desired accuracy using evaluation datasets that is separated from the training dataset. Each training experiment and related metadata is stored in the ML lifecycle tool for reproducibility.

In the model validation phase, a human-labelled validation dataset is used to evaluate the model against data that the model has not seen during the training phase. The model validation stage is an iterative process executed together with domain experts from the customer's side. In our case, medical clinicians are involved to validate the final output of the produced model. If the model accuracy is not on an acceptable level, previous stages must be iterated until acceptance is reached. This iterative nature of the model development and training process is illustrated in Figure 5 with the arrows pointing backwards from the first lifecycle phases. If a model reaches the requirements, the produced model artifacts are stored in a model registry along with dependency metadata for backwards traceability and reproducibility.

3.3.1 Infrastructure as code

A key design decision was to incorporate infrastructure as code (IaC) techniques. This involved creating cloud infrastructure configurations as code that could be version controlled and modular, enabling templating. The IaC templates were utilized to provision cloud infrastructure for multiple clients across various environments, such as development, pre-production, and production, allowing for faster infrastructure management. Terraform by Hashicorp [Hashicorp, 2022] was selected as the IaC tool since it is a widely used technology within the project organisation at Tietoevry. The IaC process is visualised in Figure 6.

The Terraform core is used to provision resources in the cloud. The provisioning process is controlled with Azure Pipelines. When the code in the Terraform repository is updated, it triggers a CI process where the DevOps pipeline uses Terraform to compare the state in the provisioned cloud infrastructure to the state to be provisioned by the Terraform code base. By comparing the states, Terraform will create a plan what resources to create, update, or destroy within the cloud infrastructure. If the plan is valid and approved, the CD pipeline will trigger Terraform core to update the cloud infrastructure accordingly.

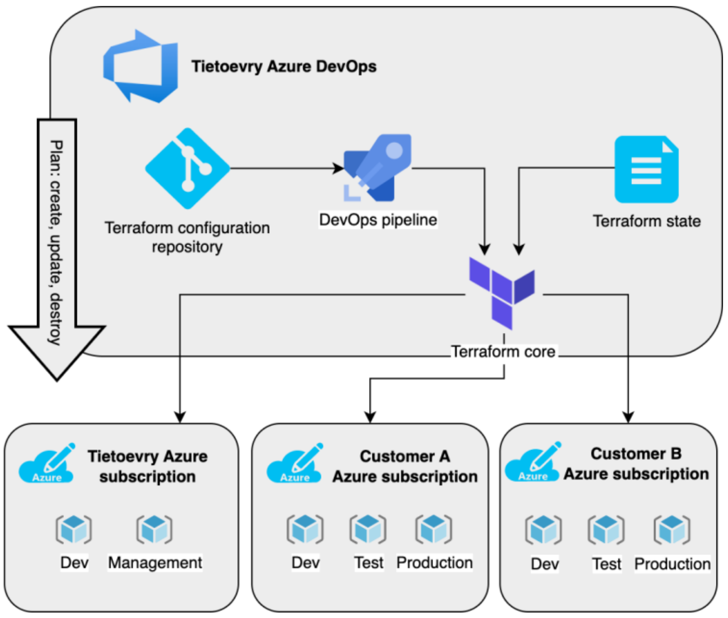

Tietoevry Azure DevOps
Plan: create, update, destroy
Terraform configuration repository
DevOps pipeline
Terraform state
Terraform core
Tietoevry Azure subscription
Dev
Management
Customer A Azure subscription
Dev
Test
Production
Customer B Azure subscription
Dev
Test
Production

Figure 6. All cloud resources are provisioned with an IaC process.

The planned cloud infrastructure in Azure consists of two top-level entities. A subscription is an agreement between an organisation and Microsoft of using Azure resources. A subscription also defines a billing unit. Thus, subscriptions are used to track costs of a set of resources. In our plan, a customer’s subscriptions consist of three resource groups: development (Dev), preproduction (Test) and production. A resource group is logical set of Azure resources and with this distinction resources for each environment required in the ML lifecycle are managed separately. For the provider’s own subscription, we defined two required resource groups. The development (Dev) resource group is used to develop and test the MLOps process and architecture in the provider's own cloud Azure subscription. The management resource group is created for storing and managing artifacts created in the model development phase within client subscriptions. The inner contents for a customer's Azure subscription and resource groups are described in the following sections.

3.3.2 Model development, training, and validation

After the initial requirement analysis phase, where the problem scope and solution requirements are defined together with the customer, the project advances to the development phase. This phase begins with data preparation. In this project, the desired data is the patient journal texts within EHRs stored in disperse data tables in a data lake of a hospital district. Commonly, the data exploration and gathering process would require more extensive work and most likely a separate project for data integrations and engineering. For 360° Patient the data is already pre-processed from raw disperse data tables in the lake into a semi-structured format as the patient events displayed in the application. This format contains the details of the patient event in a structured format and the unstructured medical narrative as the free text content. This transformation takes place in data pipelines that load the data from the data lake, transform it into the data model, i.e., the patient event document format used by the application, and load the documents into

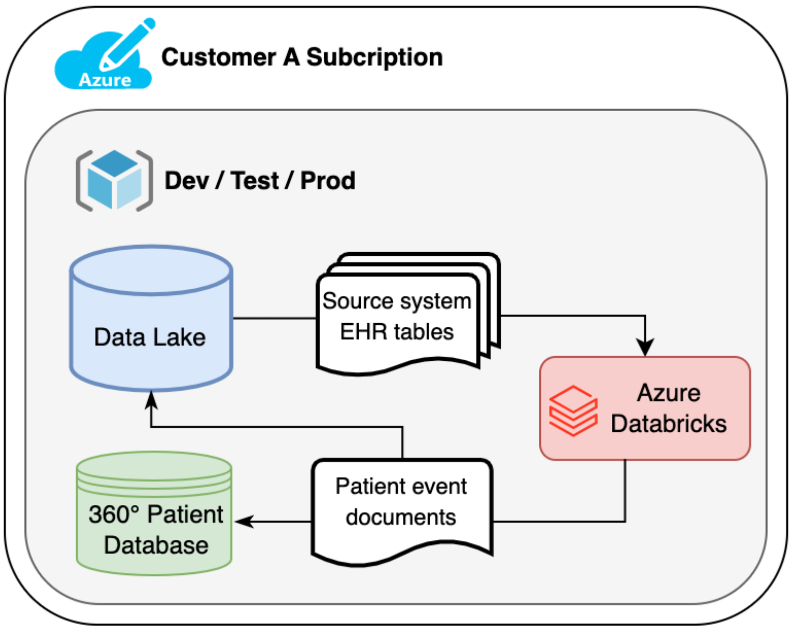


Figure 7. Data flow in the data preparation process.

the database used by the application's backend service. This data flow process is displayed in Figure 7. The free text stored with each event is the source data for our NLP model based on the BERT model architecture and its fine-tuned version for Finnish language, FinBERT by TurkuNLP.

The 360° Patient application event documents still need to be curated. Not all of them are suitable or usable in training the NLP model and its desired NER labels. The focus is on events containing medical narratives, i.e., journal texts. In this data preparation phase, suitable data is gathered from the patient events by analysing the text contents with simple rules not requiring machine learning. These queries are defined as regular expression rules that are used to mine the event data. The goal is to find a representative distribution of patient events to be used for each of the entities the NER model should learn from the data. The selected events should contain enough text with enough samples of the desired NER labels for the model to generalise in recognising many different linguistic patterns of expressing the entities. For instance, if a NER class is designed to identify mentions of a patient's smoking habits, it is essential to have a sufficient number of samples that include diverse ways in which clinicians report smoking. Azure Databricks [Databricks, 2022] and Spark [Apache Spark, 2022] were selected as the tools for preparing the train-

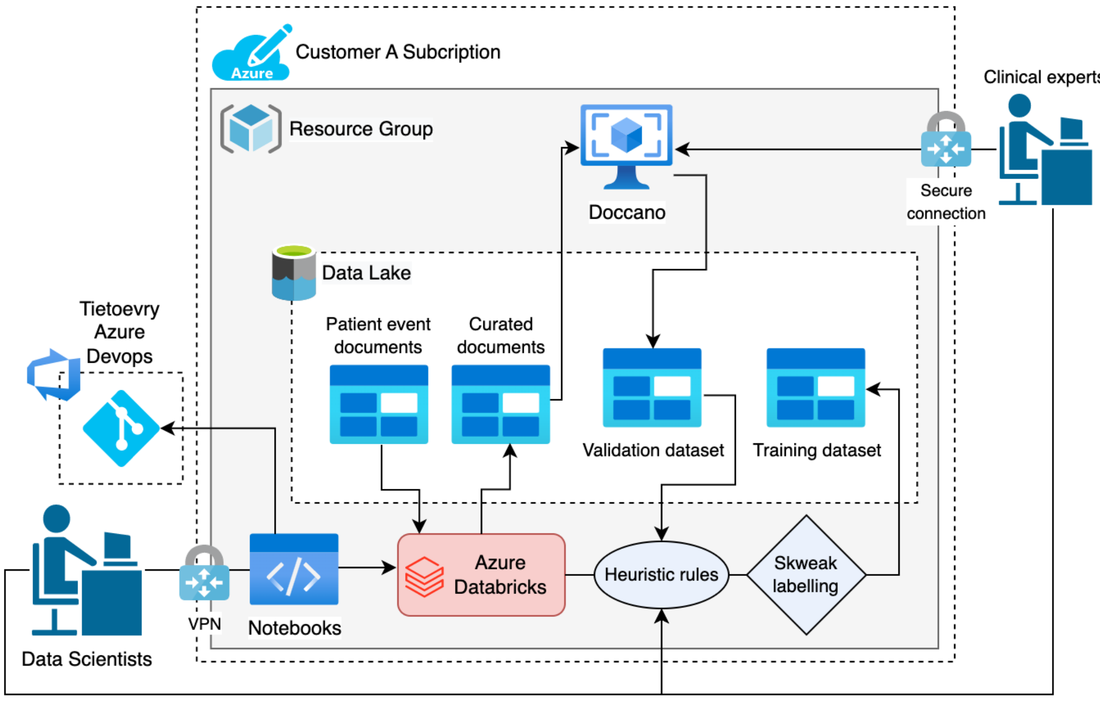
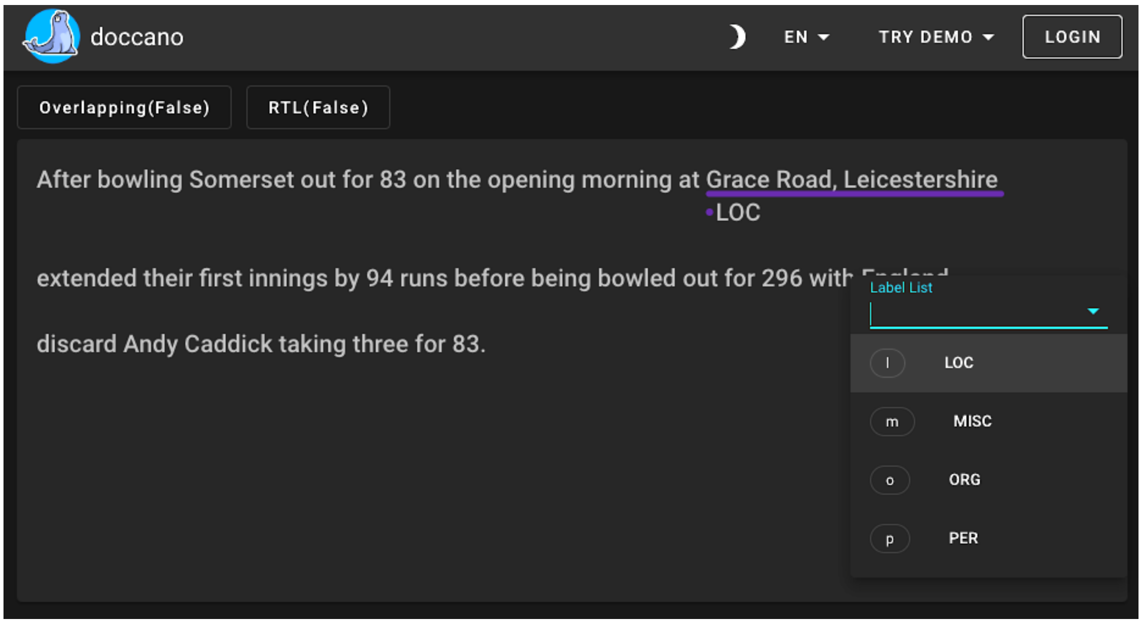


Figure 8. Data curation and labelling processes.

ing dataset. Databricks is used as the computation clustering orchestration tool Spark being the data mining and data engineering tool running in the cluster. The architecture plan for data preparation processes is displayed in Figure 8.

The curated document dataset is used in two processes. First, the dataset is used in manual data labelling. In this process, clinicians go through a subset of the curated documents and manually label the NER entities in the patient event documents. As the labelling tool Doccano [Doccano, 2022] was selected. Doccano is an open-source software enabling various text annotation tasks within an easy-to-use web application. For our purpose, Doccano is used for the manual NER labelling process. Figure 9 illustrates a demo view of Doccano and its labelling functionality. Doccano provides a REST API for interacting with the Doccano database. This enables loading the curated documents into the labelling application and then loading the labelled documents back to the data lake.



After bowling Somerset out for 83 on the opening morning at Grace Road, Leicestershire extended their first innings by 94 runs before being bowled out for 296 with England discard Andy Caddick taking three for 83.

Figure 9. Doccano, an open-source text annotation tool.

The manually labelled dataset will create the ground-truth dataset that will be used in the model validation phase. Generally, human-labelled data can be used also in the training phase of an ML model. In our case, the manually labelled data is left for model validation to minimise leaking information from the validation dataset to the model during the training process. An overview of the planned model training, validation and registering process is visualised in Figure 10. Instead of using manually labelled data for model training, we will use weakly-supervised labelling for creating the training, test, and evaluation datasets. This task is led by data scientists who examine the manually labelled data to identify common linguistic patterns in each of the NER classes. From these patterns we can create a set of heuristic expression rules to be used in the weakly-supervised labelling process. These rules are created with Skweak [NorskRegnesentral, 2022], a soft-

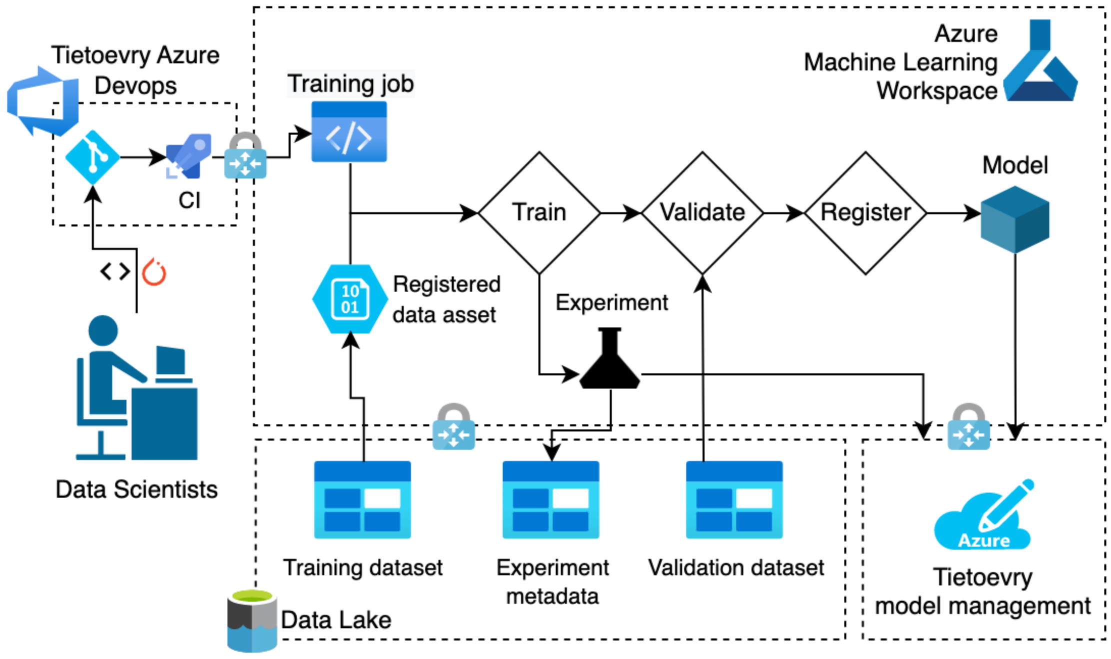


Figure 10. The model training, validation and registering process.

ware toolkit for Python language that enables weakly-supervised NLP data labelling. The curated dataset is run through the Skweak labelling process producing an automatically annotated training dataset. This dataset is then used in the NER model training process.

In addition to data preparations, the data scientists also prepare a training pipeline code base as the final output of the model development phase. This pipeline is coded locally with unit tests validating the functionalities in training the desired model architecture. Since we are dealing with sensitive healthcare data that must not leave the cloud environment of a customer, the development must be done using a simple manually generated dataset that resembles the patient event documents to be used in the actual training phase. All code is version controlled using Azure DevOps [Microsoft, 2022c] services with a Git repository stored in Tietoevry's service account. Finally, a CI pipeline is created using Azure Pipelines. The pipeline will be responsible for triggering training jobs and automating the testing of the training pipeline codebase.

The training phase is an iterative process, where the training pipeline is triggered using pipeline parameters controlling, e.g., hyperparameter tuning and other variables of the training pipeline. The output of the training phase is a model ready to be validated using the manually labelled ground-truth dataset. For a model to be acceptable for this, the training pipeline is rerun with modified variables until the model reaches a desired accuracy. A single training job run is called an experiment. Each experiment is registered and stored in Azure Machine Learning with all the dependency metadata related to each job. This includes dataset versions, the code base version, experiment logs and monitoring data. Using monitoring visualisations, the data scientists evaluate the performance of a model produced in each experiment.

The best performing models are taken to the validation phase. The manually labelled dataset is preserved for this phase. This isolated ground-truth dataset is used in the validation phase to measure model generalisation. In other words, running inference for the ground-truth dataset should result in performance as close as possible to the results obtained in the training phase. The goal is to make the model learn general patterns for each of the NER classes and not to overfit to the data used in the training process. If the trained model performs well against the manually labelled data, we can be confident, the model has learned the actual linguistic patterns used in expressing the entities the model is designed to recognise.

The validation process is also triggered with a CI pipeline in Azure DevOps. Validation logs and monitoring data is stored in a cloud storage linked to Azure Machine Learning Workspace. If a model reaches required performance against the ground-truth dataset, it is registered with all the dependency metadata. The model artifacts and dependency data are registered into Tietoevry's model management subscription. This process is triggered manually from Tietoevry's subscription using Azure DevOps pipelines. All the model related data, except for the sensitive training and validation data, is pulled into a storage account under the providers Azure subscription from where the model can be deployed for usage.

3.3.3 Online inference

Once a model is validated and registered, it can be deployed for inference usage. Two separate inference architectures were designed and tested to examine their validity in the use case. To test the validity of each of the architectures, an empirical study was conducted to evaluate the inference performance. This study is described in detail in Section 3.4.

The online inference architecture enables running the inference in real-time whenever a clinician is viewing a patient's records. The model is deployed packaged into a Docker container behind a RESTful API. Patient journal texts are sent to the API via HTTP requests created in the 360° Patient API backend service. The deployment process is orchestrated with a CI/CD pipeline in Azure DevOps. This pipeline will build the container, run integration tests on the containerised API in a CI test environment, register the container into customer's container registry, and then deploy the container into a container group in Azure Container Instances (ACI) [Microsoft, 2022d].

A container group is a scalable containerised compute resource that serves the inference API in the customer's private network for the 360° Patient API backend service to consume. When a clinician performs an advanced search action in the application, the application API sends HTTP requests to the inference API containing all the EHR free text content of a single patient in a batch request. The model service will run the list of texts through the NER model and serve the results back to the application backend. The backend service will handle the response and use the NER classifications to execute application specific logic, in this case the advanced searching by filtering the documents based on the NER results. The final output is sent back through a secure connection to the clinicians using the application on the hospital premises. Azure monitoring tools are used for monitoring various metrics, e.g., container resource workload, API event success rate, execution times and the distribution of inferred NER classes in the texts. Monitoring and logging events are set to trigger automatic alerts when anomalies occur in the inference API service. A complete view of the online deployment architecture is displayed in Figure 11.

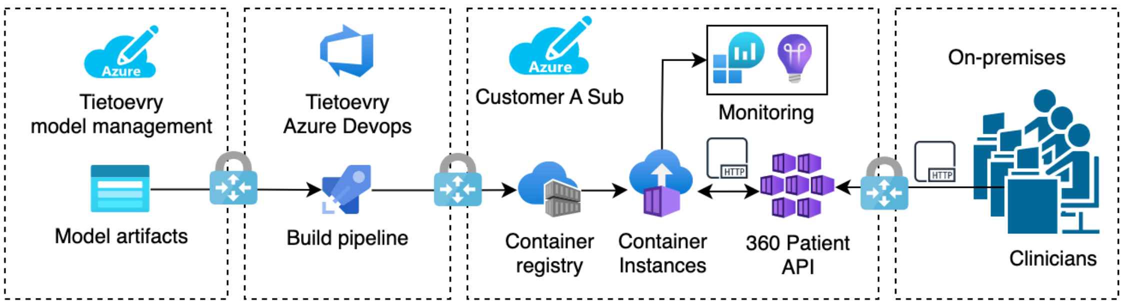


Figure 11. Online inference deployment and operation architecture.

An online inference architecture provides many benefits. For instance, compared to a batch inference architecture where all the patient events must be processed before they are accessed by a clinician, an online inference service would remove the need for a large initial processing for all historical data. A hospital district data lake also contains lots of historical and "cold" data that is never accessed by clinicians in the context of the 360°

Patient application. An online inference architecture would ensure that the computation resources are used only for data that is viewed by the clinicians. Another benefit is the ability to integrate the service for various other services via a standardised RESTful HTTP API. Yet, an online architecture places certain expectations for the API in performance. For example, in the use-case of 360° Patient, the API must respond in an acceptable amount of time. For some cases, the response time may be excessively long, particularly when patients have a substantial medical history. It is assumed response times can be shortened with more powerful virtual machines running behind the inference service.

3.3.4 Batch inference

As an alternative for the online inference, a batch inference architecture was designed for offline big data processing. This architecture is based on Spark NLP [John Snow Labs, 2022], an NLP library for Apache Spark. For the trained model to be executable with Spark NLP, first it must be converted into a TensorFlow [TensorFlow, 2022] model, a model type supported by Spark NLP. Spark NLP library is used to create an inference pipeline with the pretrained model that can be executed as a distributed computation using Apache Spark orchestrated with Databricks.

The Spark NLP pipeline build process is created using Azure Pipelines. With this CD pipeline, the model artifacts are fetched from Tietoevry's subscription, transformed into TensorFlow format, and then packaged into a Python library. The package is deployed to a customer's environment where it can be used to execute batch inference pipelines orchestrated with Databricks.

The source data in the batch process is the patient event documents pre-processed by the application data pipelines. These documents are loaded from the data lake in a distributed execution job with Apache Spark and then passed through the Spark NLP pipeline. After running the NER classification, the results are stored to the application database from where they are ready to be served via the application backend service when clinicians request the documents in the application.

With the batch inference architecture logging and monitoring occurs in the Databricks jobs. Metrics and log events are gathered from the Apache Spark executions. This enables tracking execution performance and then fine-tuning the required computation resources to optimise costs and batch job run time. Classes appearing in the NER inference results are also logged for tracking counts and distribution over time. Changes in these metrics may point to model or data drift. The batch inference deployment and operation architecture are displayed in Figure 12.

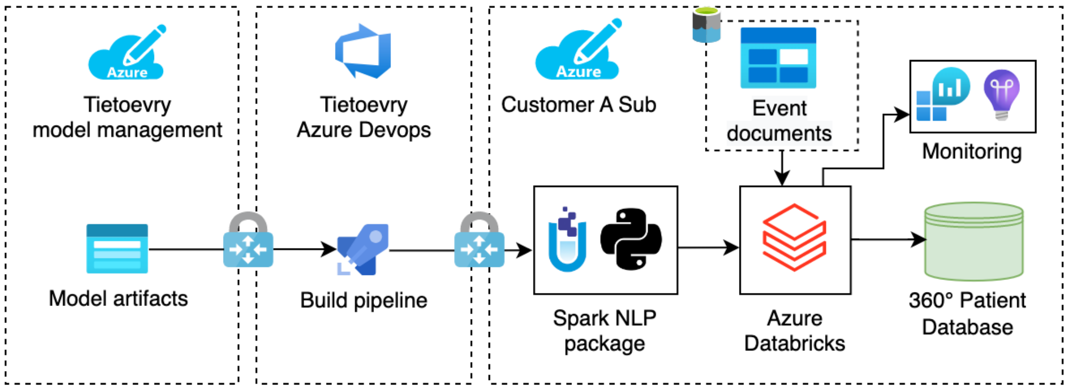


Figure 12. Batch inference deployment and operation architecture.

Apache Spark enables processing vast amounts of data by distributing the execution to multiple worker machines. With this process, we can execute inference for all the data stored in a data lake in a single offline execution. After the initial job, the batch pipeline can be run incrementally for new data loaded into the data lake. Databricks and Apache Spark also support data streaming which enables running the batch inference as a stream job to process new documents as soon as they are available in the lake. In our use case, the 360° Patient does not necessarily require an online inference architecture. Data is loaded offline to the application using the same tools planned to be used for batch inference. This enables integrating the batch inference architecture to the existing data pipelines. As stated before, the downside is the requirement to process all the data available in the lake as it is not easily foreseen what data will be used in the application and what will not be. This brings some overhead to the computation requirements of the inference pipeline, and this results in additional costs. Another drawback is that with batch processing, the data is always somewhat outdated. The amount of delay allowed by the batch processing should be defined joint with the customer. If the performance and cost-effectiveness of batch inference is acceptable, a batch architecture is a viable option in the 360° Patient use case.

3.4 Inference performance testing

To test the validity of the two proposed inference architectures, an empirical study was conducted on the inference performance with both architectures with varying computational resource using Microsoft Azure cloud services. For reproducibility of the study, a pretrained base version of the English BERT model fine-tuned for NER [Lim, 2020] was used. The model was trained using the CoNLL-2003 dataset [Tjong Kim Sang and De Meulder, 2003] and it is available open-sourced in the Hugging Face model registry [Hugging Face, 2022]. The CoNLL-2003 dataset was created for a shared language-independent NER task in which the goals were to train a model capable of recognising three distinct named entities: names of persons, organisations, and locations. It is assumed that the English pre-trained BERT-NER model is computationally as demanding as the final product in our research project. Although the final model in the project will be trained on Finnish language fine-tuned for medical entity recognition, the model architecture is similar in terms of neural layers and parameters and thus in the number of computations required to run the inference. Thus, an assumption was made that the experiment results are comparable to running inference with the final Finnish NLP solution designed in the research project.

For dataset creation, Natural Language Toolkit's [NLTK, 2022] version 3.7 and its corpora were used. Specifically, the Reuters news article corpus [University of California, 1999] was selected as a source for its textual content matching the NER classes trained for the model. The assumption was to generate somewhat realistic NER classification in the performance tests to mimic the final functionality of the planned medical NER model. The model was tested using free text in the corpus and the model was able to recognise the trained NER classes in the test dataset.

For calculating the inference performance, the measurement of tokens per second is used. The more the tokens processed per second, the better the performance. In this context, a token is a single word defined by the NLTK tokenisation. A list of tokens for the experiments is produced by using the "words" method in the NLTK corpora library. This enables reproducing these test results, since the NLTK corpora tokenisation is a standardised process.

The Reuters corpus contains over 10 000 news article documents with about 1,3 million words in total. In the experiments, the full list of words in the corpus was loaded and split into sequences of 512 tokens. The NER model was configured to accept a maximum of 512 tokens meaning each entry in an inference batch resulted in maximum load of computation for the model. To increase the data volume to mimic big data processing, I duplicated the data to reach a maximum of 6 000 sequences. This resulted in a total of 3 072 000 tokens for the maximum load in a single inference execution. To observe the impact of batch size on performance, the number of tokens per inference request was gradually increased.

For online inference, the experiments were run on five different Azure Container Instances computation resource configurations on a container group of a single container. This setup was selected to test the maximum performance of a single container. The configurations, labelled as units A through E, can be found in Table 2. Unit A features 2 virtual CPU (vCPU) cores, 4 GB of memory, and no GPUs. Unit B is configured with 4 vCPU cores, 16 GB of memory, and no GPUs. Unit C includes 2 vCPU cores, 4 GB of memory, one GPU of the K80 type. Unit D consists of 6 vCPU cores, 4 GB of memory, and one K80 GPU. Unit E is equipped with 6 vCPU cores, 4 GB of memory, and two K80 GPUs.

Unit	vCPU count	Memory	GPU count	GPU type
A	2	4	0	-
B	4	16	0	-
C	2	4	1	K80
D	6	4	1	K80
E	6	4	2	K80

Table 2. Azure Container Instances configurations.

The containers were running Python based REST API using FastAPI web framework version 0.74.1. The inference pipeline was created using Hugging Face’s Transformers API version 4.18.0 with PyTorch version 1.11.0. The sequences were posted to the inference service API in a single batch request one request at a time. No concurrency was produced in terms of requests handled by the service, i.e., no concurrent computational load was generated. For GPU-powered ACI containers, GPU was enabled using the PyTorch API. No other optimisations were done to increase performance in any of the configurations.

Azure Databricks notebooks were used to load the test dataset and to trigger the test runs. Execution performance results were measured by counting the time taken to execute the HTTP request. The maximum timeout for the HTTP requests was set to 10 minutes. Results were stored into csv files in the Databricks filesystem. To minimise network latency, the Databricks computation resources and the ACI containers were created in the same Azure region, North Europe.

For batch inference, the Spark NLP inference performance was tested on Azure Databricks using five different computation cluster configurations. The tests were performed using Databricks runtime version 10.2 ML with Scala version 2.12 and Spark version 3.2.0. Three single node clusters were configured, comprising of two CPU-powered configurations and one GPU-powered configuration. The two multi-node configurations each consisted of three standard clusters with one driver node and two worker nodes with the driver node type matching the worker node type. These multi-node clusters were used to test the performance of distributed computation with Spark NLP. In this setup, the driver node controls Spark NLP execution and distributes the inference computation for the available worker nodes. In simple terms, this setup was created to test the performance gain of increasing the inference node count from one to two and allowing Spark NLP to handle the distribution and optimisations of the computations. In Table 3, the specifics of each node type are outlined, with individual cluster configurations identified by test unit labels from F to J.

Unit	Worker type	Workers	Drivers	Memory	CPUs	GPUs	GPU memory
F	Standard_DS3_v2	0	1	14	4	0	0
G	Standard_F16	0	1	32	16	0	0
H	Standard_NC6s_v3	0	1	112	6	1	16
I	Standard_F16	2	1	64	32	0	0
J	Standard_NC6s_v3	2	1	224	12	2	32

Table 3. Azure Databricks cluster configurations.

The Azure Databricks cluster configurations F to H are configured as single-node clusters with one driver and no workers. Unit F has a node type of *Standard_DS3_v2*. The driver has 14 GB of memory, 4 CPU cores, and no GPUs. Unit G features a node type of *Standard_F16*. This configuration includes 32 GB of memory, 16 CPU cores, and no GPUs. Unit H is equipped with a node type of *Standard_NC6s_v3* having 112 GB of memory, 6 CPU cores, 1 GPU, and 16 GB of GPU memory.

Units I and J are multi-node configurations. For these units, computation resources are calculated based on the worker nodes. Unit I utilizes the *Standard_F16* node type, with 1 driver and 2 workers. This configuration provides a total of 64 GB of memory, 32 CPU cores, and no GPUs. Unit J includes a node type of *Standard_NC6s_v3* having a total of 224 GB of memory, 12 CPU cores, 2 GPUs, and 32 GB of GPU memory.

Spark NLP performance was tested by creating a Databricks notebook that loaded the Reuters dataset into a Spark data frame. Each row in the data frame consisted of a single sequence created similarly to the request entries in the online inference testing. Thus, the number of rows and tokens per test matched the amounts used in the online inference testing. The data frame was passed for the Spark NLP NER pipeline. The NER pipeline was created using the same Hugging Face BERT model used in the online inference testing. Running the inference created a new data frame in which the NER results were stored in a new column. To mimic a real-life scenario and to record the actual duration of the Spark computations, the results were loaded into a file on the Databricks filesystem. The inference duration was recorded from the point of passing the data to the NER pipeline to the point the results file was written on the filesystem.

Both the online inference and the batch inference experiments were conducted during a three-day period. Each experiment was run once daily for each compute configuration. This measure was taken to average out the effect of variability in the performance of the reserved computation resources in Azure cloud.

4 Results and analysis

4.1 Online inference results

The results for online inference are displayed in Figures 13 and 14. Table 4 contains all values of average performance in tokens per second for each computation configurations described in Section 3.4. With CPU-powered ACI containers, the performance was between 202 and 286 tokens per second for the configuration with two CPU cores and 4 GB of memory, and between 501 and 605 tokens per second for the unit with 4 CPU cores

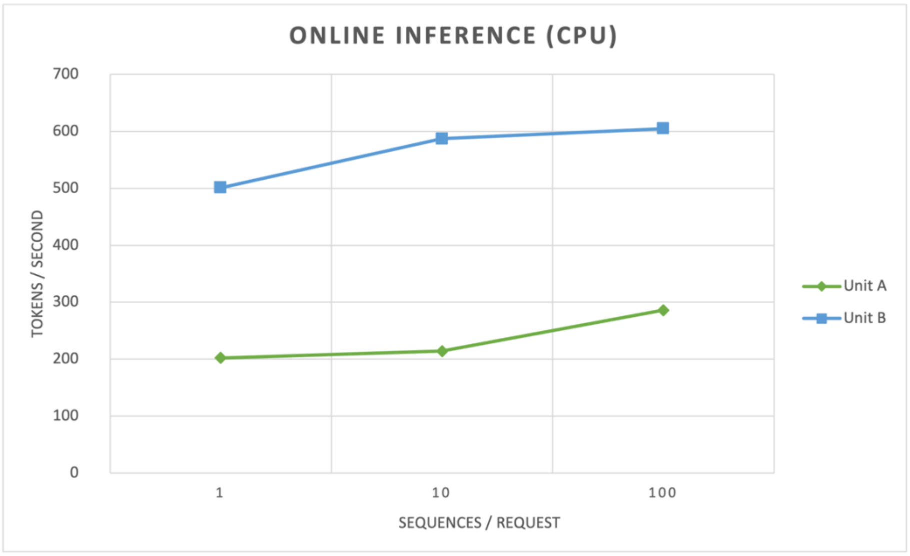


Figure 13. Performance of CPU-powered online inference.

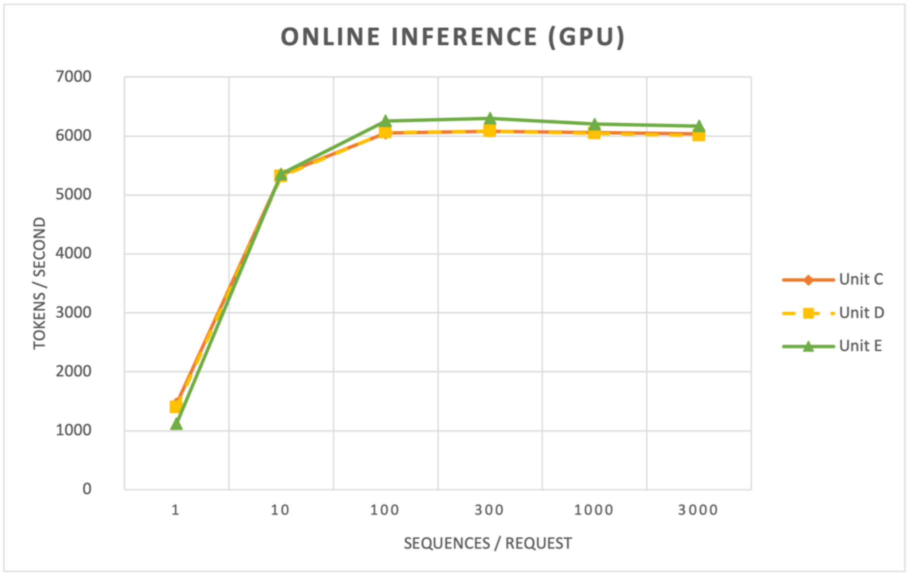


Figure 14. Performance of GPU-powered online inference.

and 16 GB of memory. The performance was improved by increasing the number of sequences passed in a single request. Yet, neither of these CPU configurations was able to produce consistent results with requests of more than 100 sequences with the given timeout of 10 minutes. By doubling the CPU core count and tripling the memory, on average 2.4 times more tokens were processed per second.

Sequences	Tokens	Unit A	Unit B	Unit C	Unit D	Unit E
1	512	202	501	1460	1402	1122
10	5120	214	587	5346	5319	5358
100	51200	286	605	6051	6062	6259
300	153600			6085	6083	6302
1000	512000			6056	6047	6204
3000	1536000			6041	6010	6170
Average		**234**	**564**	**5173**	**5153**	**5235**

Table 4. The average tokens per second results for online inference requests.

The three different GPU-powered ACI configurations gave consistent results compared to one another regardless of the amount of CPU or GPU cores and memory used. With the least powered configuration, Unit C, using 2 virtual CPU cores, 4 GB of memory and 1 GPU, average performance ranged from 1460 to 6085 tokens per second. Increasing the CPU count to 6 did not make significant changes to these results. Adding another GPU core to the configuration, Unit E, the performance first dropped by about 23 % and on the maximum result, 6302 tokens per second, increased by only about 4 % comparing to Unit C, 6085 tokens per second, with the least compute power of the GPU units. All configurations peaked at 300 sequences per request with an average of 6156 tokens per second. None of the configurations were able to run the request with 6000 sequences within the 10-minute timeout.

The performance was significantly higher for the GPU-powered ACI containers compared to the CPU-powered instances. Comparing the average performance for all configurations within the 1 to 100 sequences range of requests, units from C to E averaged with 4264 tokens per second and units A and B only 399 tokens per second. The GPU-powered units, on average, exhibited more than 10 times better performance than the CPU-powered containers. Notably, the performance of GPU configurations did not improve significantly by adding more CPU cores or GPU units. Performance dropped for the request of just one sequence with additional CPU cores. By omitting results for requests containing only one sequence, with Unit E the performance improved only 2 % compared to Unit C. GPU-powered units maxed out at around 6000 tokens per second after 100 sequences and kept this performance until reaching the 10-minute timeout limit for requests.

4.2 Batch inference results

Batch inference results are displayed in Figure 15 and presented in Table 5. Results for Databricks single node clusters that run CPU-powered virtual machines are provided by Units F and G. Unit F averaged a performance from 77 to 4397 tokens per second. Unit G averaged a result from 82 to 9606 tokens per second. Unit H was a GPU-powered single node cluster that averaged a result of 98 to 14966 tokens per second. Units I and J were created with Databricks standard cluster configuration having one driver node and two worker nodes. The CPU-powered Unit I averaged a performance from 28 to 18507 tokens per second. The GPU-powered Unit J averaged a performance from 24 to 23848 tokens per second. All units were able to process all batches from 1 to 6000 sequences.

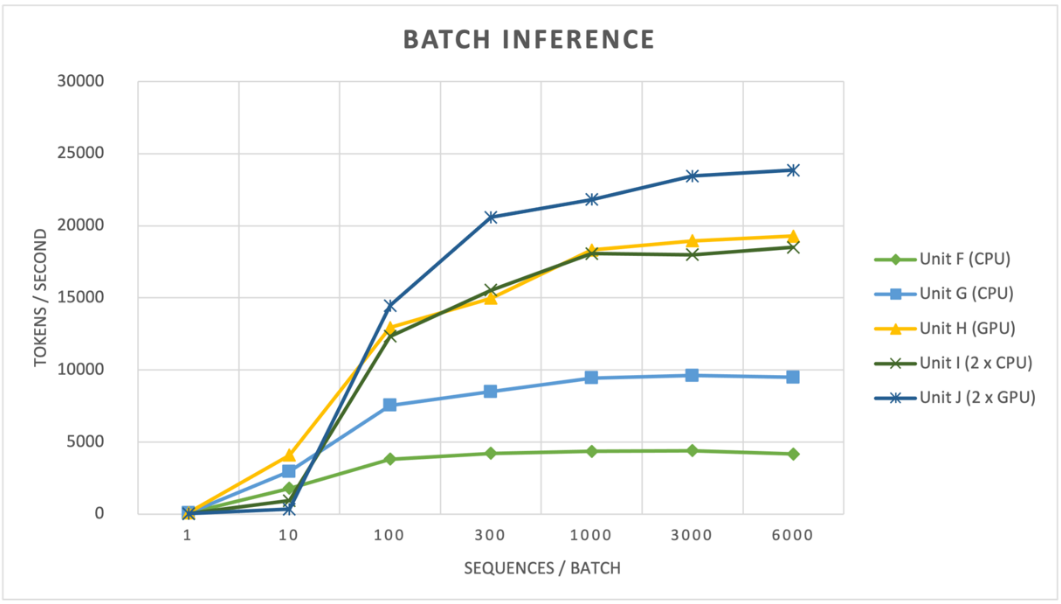


Figure 13. Average results of running batch inference with Spark NLP.

The outcomes obtained from batch sizes 1 to 10 are excluded as outliers and not considered for the subsequent result comparisons. Results for smaller batch sizes gave significantly weaker performance compared to larger batch sizes. The decision to omit these results is reinforced by the expected use case of a batch inference architecture, where the inference pipeline is to be used for processing significant volumes of text sequences in a single batch.

Sequences	Tokens	Unit F	Unit G	Unit H	Unit I	Unit J
1	512	77	82	98	28	24
10	5120	1784	2953	4088	926	336
100	51200	3802	7537	12934	12322	14466
300	153600	4215	8483	14966	15532	20577
1000	512000	4352	9433	18310	18063	21818
3000	1536000	4394	9606	18957	17982	23451
6000	3072000	4158	9478	19284	18507	23848
Average		**3254**	**6796**	**12662**	**11908**	**14931**
Average (omit 1-10)		**4184**	**8907**	**16890**	**16481**	**20832**

Table 5. Average tokens processed per second in batch inference executions.

Batch inference performance improved significantly when increasing the batch size of sequences fed to the inference pipeline. Performance improvement started to plateau at around 1000 sequences in a batch. Unit F, which was powered by CPU, experienced a decrease in performance when processing batches containing 6000 sequences. Unit G had maximum performance at 3000 sequences. All GPU-powered configurations and the CPU configuration with distributed computations enabled, Unit I, were still slightly improving at 6000 sequences.

There was a significant boost in performance when increasing the computational capability of the single node units. With Unit G, processing speed improved 2.1 times compared to Unit F. Furthermore, enabling GPU-powered inference with Unit H, the performance was 1.9 times that of Unit G. Comparing to Unit F with least computational power, GPU-powered Unit H was 4 times more efficient.

Going from single node executions to distributed clusters, the performance gain was more significant in the CPU-powered units. Unit I with one driver and two worker nodes running the same virtual machine type as Unit G, gave a performance boost of 85 %. Comparing the GPU-powered Unit H, running the same virtual machine type to a distributed configuration, Unit J, performance increased by 23 % when running distributed inference. When comparing the distributed computations to single node configurations, Unit I with virtual machine type matching CPU-powered Unit G had comparative results to single GPU-powered configuration Unit H if batch sizes 1 and 10 were omitted.

4.3 Inference results comparison

When comparing overall results between batch and online inference, batch processing outperformed online inference. Regarding computation power, the CPU-powered batch processing Unit F was comparable to online processing Unit B with 4 CPU cores and 14 gigabytes of RAM compared to 4 CPU cores and 16 gigabytes of RAM respectively. Including the smallest batch sizes of 1-10, Unit F resulted an average of 3254 tokens per seconds. Unit B gave an average of 564 tokens per second. On a comparable CPU-powered non-distributed configuration, batch processing was 5.8 times more efficient.

With GPU-powered non-distributed configurations, Unit H outperformed the best GPU-powered online configuration, Unit E, with performance jumping from 5235 tokens per second to 12662 tokens. Unit H resulted in a performance 2.4 times higher compared to Unit E.

Online inference configurations outperformed batch inference only with a single sequence in a request. CPU-powered units maxed a result of 501 and GPU units 4160 tokens per second. With batch inference, the smaller batch sizes gave a significantly poorer performance. For a single sequence, batch configurations maxed at 82 and 98 tokens per second for CPU and GPU-powered units respectively. Notably, batch sizes of larger than 3000 sequences were only handled by the batch inference configurations. CPU-powered online configurations were able to process a mere 100 sequences per request.

5 Conclusions

The goal of this thesis was to introduce an MLOps process model to be used for productising an NLP solution in healthcare big data context. To answer research question R1, an MLOps process was introduced that describes the lifecycle of an ML solution from data processing to training and deploying the ML NLP model. The MLOps process was centred around Microsoft Azure cloud services but depicted on a higher-level supporting generalising the model for other cloud platforms and ML lifecycle tools.

The MLOps process has three distinct phases: data preparation and curation, model training and validation, and model deployment and inference operation. The level of automation planned for each of the phases varied based on the need for human intervention. Both the customer and the provider experts need to participate in the data curation and preparation phase of the ML project. Clinicians play a vital role in labelling the validation dataset to be used in validating the model performance in the training phase. Data scientists are required for generalising clinicians labelling patterns into heuristic rules used in weakly-supervised labelling of the training dataset.

The model training and validation phase was planned as an automated pipeline that can be triggered to run the training and validation process when either the training or the validation datasets are changed. This process can be iterated until the desired performance is achieved for the trained model.

For the deployment and inference phase, two architectures were proposed. To compare them, an empirical study was conducted to evaluate the performance of each architecture with varying computational configurations. The proposed architecture for batch inference outperformed the online inference architecture when using the average amount of tokens processed per second as the metric. Thus, to answer research question R2, the batch inference architecture is selected as the optimal solution for deploying a BERT-based NER model to process big data in the clinical healthcare application context described in this thesis. The Spark NLP batch inference pipeline produced significantly better results for processing high volumes of data compared to Docker container based online inference. The NER inference pipeline built with Spark NLP running on Apache Spark orchestrated with Azure Databricks produced results suitable for productising the inference architecture without much need for further performance optimisations. On the other hand, the online inference did not perform on a level suitable for big data context without improving the performance with optimisations not covered in this thesis.

6 Discussion

This thesis contributes to the research area of MLOps. A concrete MLOps architecture referring to publicly available cloud services gives a reference point for research and implementation of MLOps in practice. MLOps expertise is evolving in response to emerging needs. Private and public sector companies implementing productised ML solutions are in the forefront of pushing MLOps development. In scientific literature though, MLOps is still a novel topic.

The need for productised ML systems is rising and thus the need for research in improving ML life cycle processes is evident and valuable. Especially, in the healthcare domain where robustness of software solutions is necessary, introducing ML solutions to clinical work is not trivial and requires research advancements not only in ML (model) development but also in ML productising. To reach the gap between researching and productising ML and AI in healthcare, MLOps is a promising concept to mitigate the uncertainties that arise with building ML systems. Borrowing concepts from well-established practices of DevOps and bringing automated CI and CD processes to ML model development, training, and deployment phases, brings transparency and maintainability to ML system life cycle. Backwards traceability and the ability to pinpoint erroneous phases in the ML model productisation are key in reducing uncertainty. Once the processes of data management, model training and model deployment have been automated, it enables the ML system developers to focus on delivering robust, efficient, and trustworthy ML solutions.

MLOps brings a promise of improved ML system development. Yet, more research is required for developing MLOps practices and tools. The efficacy of implementing MLOps in improving outcomes of ML projects needs to be studied. MLOps affects various aspects of ML system life cycle. For example, future research should tease out the effects of MLOps automation in improving model accuracy and performance. We can assume automation improves developer experience and reduces human errors in the ML model development phase. Another research area is model deployment. Researching and innovating automated model deployment pipelines can reduce the gap between model research and productisation of ML solutions. Future research should track the advancements in MLOps and its implementation in companies and public actors producing ML solutions. Finally, the effect of MLOps in advancing ML system deployment speed and the amount of productised ML solutions across different industries should be studied.

Much of research has focused on ML methodology and model architecture development. These areas are crucial for pushing the capabilities of ML and AI. Advances in ML methods, and recently especially in deep learning, create possibilities for using ML in ever more areas. But as the possibilities for using ML broaden, so does the need for improving and innovating model deployment. Executing inference with deep neural networks is computation intensive and requires more computation power as model architectures grow in complexity. This is not only a technical issue needing to be solved in the context of ML solution development, but it is also an environmental and societal issue. As the use of ML systems increases, research is needed to mitigate the impact of these systems for energy consumption. In the literature, this need is recognised and studied in the form of model compression and acceleration [Choudhary *et al.*, 2020].

In this thesis, it was recognised that model deployment architecture and inference performance play a vital role in productising the final ML solution. As described, the inference performance affects the energy consumption and computational power requirements of the operated ML system, but it also defines the limitations of the system. In the context of big data, inefficient inference execution can create a bottleneck in an ML solution. In the case study project of this thesis, data to be fed for the inference pipeline would consist of the complete EHR history of a healthcare district. Without a sufficient inference architecture, this task may be impossible in terms of execution time. And even if processing all historical data is achievable within a reasonable timeframe, it could still result in significant expenses. To enable such use cases and minimise the costs of big data processing, research and development are required in inference architecture design. In this thesis, a big data capable inference architecture was planned, tested, and proposed as a viable solution. For future research, it is proposed to combine model compression and acceleration techniques with inference architecture design to produce efficient inference pipelines for big data context with minimised computational and energy consumption requirements.

References

Alpaydin, E. (2016). *Machine Learning: The New AI.* MIT Press.

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software Engineering for Machine Learning: A Case Study. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 291–300.

Apache Spark (2022), Unified Engine for large-scale data analytics. Retrieved June 9, 2022, from *https://spark.apache.org*

Arpteg, A., Brinne, B., Crnkovic-Friis, L., & Bosch, J. (2018). Software engineering challenges of deep learning. *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018.*

Beam, A. L., & Kohane, I. S. (2018). Big data and machine learning in health care. *JAMA - Journal of the American Medical Association, 319*(3).

Borg, M. (2022). Agility in Software 2.0 – Notebook Interfaces and MLOps with Buttresses and Rebars. *Lecture Notes in Business Information Processing*, *438 LNBIP*.

Choudhary, T., Mishra, V., Goswami, A., & Sarangapani, J. (2020). A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, *53*(7).

Databricks (2022). Data Lakehouse Architecture and AI Company. Retrieved June 9, 2022, from *https://databricks.com/*

David S. Lim. (2020). dslim/bert-base-NER. Retrieved June 21, 2022, from *https://huggingface.co/dslim/bert-base-NER*

Doccano (2022). Open source annotation tool for machine learning practitioners. Retrieved June 17, 2022, from *https://github.com/doccano/doccano*

Docker (2023). Docker: Accelerated, Containerized Application Development. Retrieved February 5, 2023, from *https://www.docker.com/.*

Hugging Face. (2022). Models. Retrieved June 21, 2022, from *https://huggingface.co/models*

Fitzgerald, B., & Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, *123*, 176–189.

Gulshan, V., Peng, L., Coram, M., Stumpe, M. C., Wu, D., Narayanaswamy, A., Venugopalan, S., Widner, K., Madams, T., Cuadros, J., Kim, R., Raman, R., Nelson, P. C., Mega, J. L., & Webster, D. R. (2016). Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA - Journal of the American Medical Association*, *316*(22).

Hashicorp (2022). Terraform by Hashicorp. Retrieved June 8, 2022, from *https://www.terraform.io*

HUS (2022). HUS Clinical NLP Seminar. Retrieved June 8, 2022, from *https://fcai.fi/calendar/hus-clinical-nlp-seminar*.

Jiang, F., Jiang, Y., Zhi, H., Dong, Y., Li, H., Ma, S., Wang, Y., Dong, Q., Shen, H., & Wang, Y. (2017). Artificial intelligence in healthcare: Past, present and future. In *Stroke and Vascular Neurology, 2*(4).

Jenkins (2023). Jenkins – an open source automation server which enables developers around the world to reliably build, test, and deploy their software. Retrieved February 5, 2023, from *https://www.jenkins.io/*

John Snow Labs. (2022). Spark NLP. Retrieved June 20, 2022, from *https://nlp.johnsnowlabs.com/*.

Karamitsos, I., Albarhami, S., & Apostolopoulos, C. (2020). Applying devops practices of continuous automation for machine learning. *Information (Switzerland)*, *11*(7).

Kläs, M., & Vollmer, A. M. (2018). Uncertainty in machine learning applications: A practice-driven classification of uncertainty. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *11094 LNCS*.

Kumeno, F. (2019). Sofware engneering challenges for machine learning applications: A literature review. *Intelligent Decision Technologies*, *13*, 463–476.

Kubernetes (2023). Production-Grade Container Orchestration. Retrieved February 5, 2023, from *https://kubernetes.io/*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Lwakatare, L. E., Crnkovic, I., & Bosch, J. (2020). DevOps for AI – Challenges in Development of AI-enabled Applications. *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 1–6.

Microsoft (2022a). Cloud Computing Services. Retrieved June 8, 2022, from *https://azure.microsoft.com/*.

Microsoft (2022b). Azure Machine Learning - ML as a Service. Retrieved June 8, 2022, from *https://azure.microsoft.com/en-us/services/machine-learning/*.

Microsoft (2022c). Azure DevOps Services. Retrieved June 19, 2022, from *https://azure.microsoft.com/en-us/services/devops/.*

Microsoft (2022d). Azure Container Instances. Retrieved June 20, 2022, from *https://azure.microsoft.com/en-us/services/container-instances/.*

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. MIT Press.

Mäkinen, S., Skogström, H., Laaksonen, E., & Mikkonen, T. (2021). Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help? *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, 109–112.

NLTK (2022). Natural Language Toolkit. Retrieved June 21, 2022, from *https://www.nltk.org/.*

NorskRegnesentral (2022). A software toolkit for weak supervision applied to NLP tasks. Retrieved June 17, 2022, from *https://github.com/NorskRegnesentral/skweak.*

Patel, V. L., Shortliffe, E. H., Stefanelli, M., Szolovits, P., Berthold, M. R., Bellazzi, R., & Abu-Hanna, A. (2009). The coming of age of artificial intelligence in medicine. *Artificial Intelligence in Medicine*, 46 (1).

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J. F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 2494–2502.

TensorFlow. (2022). TensorFlow. Retrieved June 20, 2022, from *https://www.tensorflow.org/.*

Tietoevry (2022a). Creating purposeful technology. Retrieved June 9, 2022, from *https://www.tietoevry.com/.*

Tietoevry (2022b). Intelligent Wellbeing for health care and welfare. Retrieved June 5, 2022, from *https://www.tietoevry.com/en/industries/healthcare-and-welfare/healthcare/tieto-intelligent-wellbeing/.*

Tjong Kim Sang, E. F., & de Meulder, F. (2003). Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. *Proceedings of the 7th Conference on Natural Language Learning, CoNLL 2003 at HLT-NAACL 2003*.

TurkuNLP (2019). TurkuNLP/FinBERT: BERT model trained from scratch on Finnish. Retrieved June 5, 2022, from *https://github.com/TurkuNLP/FinBERT.*

University of California (1999). Reuters-21578 Text Categorization Collection. Retrieved June 21, 2022, from *http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html.*

Valohai (2022). MLOps - Machine Learning Operations. Retrieved May 16, 2022, from *https://valohai.com/mlops/*.

Wan, Z., Xia, X., Lo, D., & Murphy, G. C. (2021). How does Machine Learning Change Software Development Practices? *IEEE Transactions on Software Engineering*, *47*(9), 1857–1871.