

Master Thesis



Czech  
Technical  
University  
in Prague

**F3**

Faculty of Electrical Engineering  
Department of Computer science

## Survey of ML Model Serving Solutions and criteria for selection thereof

**Petr Poliak**

Field of study: Open informatics  
Subfield: Software Engineering

Supervisor: Ing. Martin Ledvinka

May 2021



## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Poliak** Jméno: **Petr** Osobní číslo: **439562**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávající katedra/ústav: **Katedra počítačů**  
Studijní program: **Otevřená informatika**  
Specializace: **Softwarové inženýrství**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Srovnání přístupů nasazení modelů strojového učení**

Název diplomové práce anglicky:

**Survey of ML Model Serving Solutions and criteria for selection thereof**

Pokyny pro vypracování:

- 1) Familiarize yourself with the problem of Machine learning model deployment.
- 2) Compare at least three different approaches of deploying Machine learning models.
- 3) Define a framework for evaluating the suitability of specific Machine learning model deployment approaches in different scenarios.
- 4) Evaluate the framework with respect to the problems described in literature.
- 5) Use the evaluation framework on a set of at least three industry-based use cases of various levels of complexity. Compare your findings with the decisions made by experts.

Seznam doporučené literatury:

1. Andrei Paleyes, Raoul-Gabriel Urma, Neil D. Lawrence et al. - Challenges in Deploying Machine Learning: a Survey of Case Studies, NIPS 2020
2. Sculley et al. - Hidden Technical Debt in Machine Learning Systems, NIPS 2015
3. Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, Liang Zhang - LASER: A Scalable Response Prediction Platform For Online Advertising, WSDM 2014

Jméno a pracoviště vedoucí(ho) diplomové práce:

**Ing. Martin Ledvinka, skupina znalostních softwarových systémů FEL**

Jméno a pracoviště druhého(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.02.2021**

Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **30.09.2022**

Ing. Martin Ledvinka  
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.  
podpis děkana(ky)

## III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

\_\_\_\_\_  
Datum převzetí zadání

\_\_\_\_\_  
Podpis studenta



## Acknowledgements

First, I want to thank my supervisor for his guidance and for his courage to explore an unfamiliar topic. Further, I want to thank Vojta for his domain expertise & leadership and everyone in my team at Avast, thanks to whom I could explore MLOps more than a single person in a year could.

Big thanks to Radek, João Da Silva (Avast), and Alex Hagerf (Socialbakers) for the help, patience, and information used in the case studies.

And lastly, to my friends Petr and Marek for mutually motivating each other to grind till the end.

## Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 20. May 2021

## Abstract

Machine learning operations is an emerging field in software engineering. It aims to streamline the machine learning development process from the model's concept to its application. In this thesis, we focus on machine learning model serving approaches for online inference use cases.

First, we provide a brief background for machine learning operations and delineate the situations we consider in this thesis. Then, we propose four criteria used to describe and evaluate the deployment approaches. Next, we identify five different schemes to deploy machine learning models and assess them with regard to the defined criteria. We provide the architectures of the solutions and the use case where they fit along with example implementations.

Further, we propose a framework to guide selecting the approach that best fits the use case. Lastly, we provide two industry case studies describing the architecture, implementation, and reasoning behind those decisions.

**Keywords:** Machine Learning Operations, Machine Learning, MLOps, Online inference

**Supervisor:** Ing. Martin Ledvinka

## Abstrakt

Machine learning operations (provoz strojového učení) je začínající obor softwarového inženýrství. Má za cíl zefektivnit proces vývoje strojového učení od konceptu modelu po jeho nasazení. V této diplomové práci se zaměříme na architektury nasazování modelů strojového učení pro online klasifikaci.

Nejprve je čtenáři krátce představen obor machine learning operations a vymezíme zaměření této práce. Poté navrhne a vysvětlíme čtyři kritéria, která jsou později použita k popisu a vyhodnocení jednotlivých přístupů nasazování modelů. Dále popíšeme pět různých schémat pro nasazování modelů strojového učení a u každého rozebereme jednotlivá kritéria. Každé schéma má popsanou architekturu a příklad situace, kde by se dalo použít, včetně ukázkové implementace.

Dále navrhne framework pro výběr schématu, které nejlépe vyhovuje situaci pro nasazování modelů. Nakonec poskytneme dvě detailní studie z průmyslu popisující architekturu, implementaci a argumentaci za rozhodnutími, které k danému výsledku vedly.

**Klíčová slova:** Machine Learning Operations, Provoz strojového učení, MLOps, Strojové učení

**Překlad názvu:** Srovnání přístupů nasazení modelů strojového učení

# Contents

<b>1 Introduction</b>	<b>1</b>	5.1.2 Second option	28
1.1 Notation	2	5.1.3 Discouraged	28
<b>2 Scope delimitation and context</b>	<b>3</b>	5.2 Providing self-service platform	28
2.1 Machine Learning lifecycle	3	5.2.1 Completeness first	29
2.2 Running inference	4	5.2.2 Extensibility first	29
2.3 Machine learning operations	5	5.2.3 Special use case	29
<b>3 Schema evaluation criteria</b>	<b>7</b>	5.3 When to switch framework selection	29
3.1 Scale	7	<b>6 Case studies</b>	<b>31</b>
3.2 Simplicity	8	6.1 Avast	31
3.3 Completeness	8	6.1.1 Initial Solution	32
3.4 Extensibility	9	6.1.2 Transition	33
<b>4 Deployment scheme definitions</b>	<b>11</b>	6.1.3 Current MLOps platform	34
4.1 Native serving	12	6.2 Socialbakers	37
4.1.1 Context	13	6.2.1 Initial state of ML serving	38
4.1.2 Scale	13	6.2.2 Motivation for change	39
4.1.3 Simplicity	13	6.2.3 Present MLOps initiative	39
4.1.4 Completeness	14	6.3 Survey	43
4.1.5 Extensibility	14	6.3.1 Survey sections	43
4.2 Software stack integration	14	6.3.2 Survey results	44
4.2.1 Context	16	<b>7 Conclusion</b>	<b>49</b>
4.2.2 Scale	16	<b>A Tech glossary</b>	<b>51</b>
4.2.3 Simplicity	16	A.1 TeamCity	51
4.2.4 Completeness	16	A.2 MLFlow	51
4.2.5 Extensibility	17	A.3 Airflow	52
4.3 E2E/cloud solution	17	A.4 Kubernetes	53
4.3.1 Context	17	A.4.1 Helm	53
4.3.2 Scale	18	A.5 Seldon	54
4.3.3 Simplicity	18	A.6 GitOps	54
4.3.4 Completeness	18	<b>B Large figures and originals</b>	<b>57</b>
4.3.5 Extensibility	19	<b>C Attachment</b>	<b>63</b>
4.4 Tech stack integration	20	<b>D Bibliography</b>	<b>65</b>
4.4.1 Context	21		
4.4.2 Scale	21		
4.4.3 Simplicity	22		
4.4.4 Completeness	22		
4.4.5 Extensibility	22		
4.5 In-house custom solution	23		
4.5.1 Context	23		
4.5.2 Scale	24		
4.5.3 Simplicity	24		
4.5.4 Completeness	24		
4.5.5 Extensibility	25		
<b>5 Deployment scheme selection</b>	<b>27</b>		
5.1 Minimize (initial) investment	27		
5.1.1 First option	28		

## Figures

1.1 Diagram demonstrating the conventions used in this thesis. <i>Component A</i> calls the <i>Service’s API</i> , which uses <i>Component B</i> to produce the <i>Stored Object</i> . . . . .	2
2.1 Four stages of Machine Learning lifecycle — Data Management, Model Learning, Model Verification, and Model Deployment. Source: Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges [ACP19] Enlarged: (Figure B.1) . . . . .	4
4.1 Native serving architecture diagram. The clients issue requests to the API, which forwards the data to the initialized model for inference. The results are then returned to the client. . . . .	12
4.2 Software stack integration architecture diagram. Blue parts belong to the library, and green are implemented code. The clear distinction for this scheme is the way the consumer interacts with the model. The model is exported to an intermediate format, and the inference happens in the actual application instead of it happening on a remote service. . . . .	15
4.3 Functionality offered by the Amazon SageMaker. Note that AWS changes its product offerings frequently, and the same functionality can be provided under a different name in the future. Source: <a href="https://aws.amazon.com/sagemaker">https://aws.amazon.com/sagemaker</a> Enlarged: (Figure B.2) . . . . .	19
4.4 Architecture of Uber’s Machine learning platform Michelangelo. At the top of the diagram, there are stages marked in orange, and we can see how the individual components interact. Source: Meet Michelangelo: Uber’s Machine Learning Platform [HB] Enlarged: (Figure B.3) . . . . .	21
6.1 Legacy architecture diagram for the Device Identification service. The <i>ML Classification</i> package is an HTTP client calling the ML service’s endpoint for inference. . . . .	33
6.2 Architecture diagram of the Schnitzel (MLOps) platform at Avast. The top part depicts the process and user interaction. The bottom is the component interaction. Source: Avast Original: (Figure B.4) . . . . .	35
6.3 Architecture of the new Device identification service. The service now embeds the model evaluation using the ONNX [BLZ+] and retrieves the model from the MLFlow model repository during deployment. . . . .	37
6.4 Diagram of the MLOps serving platform at Socialbakers. Maratonec is an internal build/compute platform, <code>mlflow-serve</code> is a GitLab repository integrating MLFlow serving and Maratonec, [MLProject] symbolizes projects producing ML models, the Model repository is MLFlow backed by Amazon S3 object storage. . . . .	41



## Tables

6.5 Diagram of the sentiment pipeline demonstrating an example usage of the Message queue service model deployment. Blue – input queues, green – output queues, red – other output queue. Orange – deployed models. At the center is the sentiment analysis application connecting the queues implemented with Akka streams. Source: Socialbakers Original: (Figure B.5)	42
6.6 Histogram of the number of employees	44
6.7 Histogram of ML operations team	45
A.1 Apache Airflow platform architecture. Source: <a href="https://airflow.apache.org/docs/apache-airflow/stable/concepts.html">https://airflow.apache.org/docs/apache-airflow/stable/concepts.html</a>	
	53
A.2 Kubernetes components diagram. Source: What is Kubernetes[lea]	54
A.3 Seldon Core stack Source: <a href="https://www.seldon.io/tech/products/core">https://www.seldon.io/tech/products/core</a>	55
A.4 GitOps workflow diagram for push-based deployment Source: <a href="https://www.gitops.tech">https://www.gitops.tech</a>	55
B.1 Machine Learning Lifecycle Diagram Source: Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges [ACP19]	
	58
B.2 Functionality offered by the Amazon SageMaker Source: <a href="https://aws.amazon.com/sagemaker">https://aws.amazon.com/sagemaker</a>	59
B.3 Architecture of Uber’s Machine learning platform Michelangelo. Source: Meet Michelangelo: Uber’s Machine Learning Platform [HB]	60
B.4 Avast Schnitzel original diagram. Author: João Da Silva, Avast	61
B.5 Sociabakers sentiment analysis pipeline original diagram. Author: Alex Hagerf, Socialbakers	62
6.1 Contingency table showing the relationship between company size and MLOps adoption. We find no significant difference between different company size categories.	45
6.2 Histogram of model deployment schemes. We see the Tech stack integration and Native serving are the two most popular approaches.	46
6.3 Table showing the relationship between model deployment scheme and number of production models.	46
6.4 Relationship between the MLOps adoption and used model deployment approach. Notable is the usage of E2E platform as a complex scheme without a designated MLOps team to maintain it.	47





# Chapter 1

## Introduction

Machine learning and artificial intelligence are being adopted by the industry in ever-increasing trends[aim20][ent19]. However, the rapid adoption tends to neglect certain aspects that have been shown to be detrimental in the long term. Technical debt, the concept that moving quickly accumulates costs that need to be addressed later, has been shown[SHG<sup>+</sup>15] to be resurfacing in machine learning solutions. The software engineering best practices are being rediscovered in the context of systems working around/with machine learning models.

This thesis focuses on deploying machine learning models in industry settings. The goal is to identify different approaches and situations for exposing and managing ML models. We describe the MLOps' responsibilities and background and present several schemas for serving machine learning models.

The structure of this thesis is as follows. First, we provide a brief background of machine learning operation in chapter 2. We explain the Machine learning lifecycle and which parts of it are important for this text.

In chapter 3 four criteria are defined that we use to evaluate the machine learning model deployment schemas. The criteria are described by what they encompass, how to understand them, and how they relate to each other.

Chapter 4 identifies five different schemas of deploying machine learning models for serving with varying degrees of maturity and feature completeness. The schemas are described using the previously defined criteria, and either general use case or an example of specific implementation is given.

Chapter 5 provides the reader with a framework to help decide which scheme from the previous chapter best fits their use case. Two views are presented. The first is intended for the initial stages of machine learning experimentation, where the focus is on having results fast; the second for more mature use cases and more extensive scope. The framework was based on personal experience and knowledge gathered while working on industry case studies.

In chapter 6 we analyze in-depth two industry case studies. We describe the initial technical solution and the new architecture. Furthermore, we explain the triggers that led to the change and the reasoning behind the decisions extending on the previous chapter. Additionally, we look at a survey to

analyze the machine learning operations trends in the industry.

The thesis is concluded in chapter 7 where we evaluate the findings of this thesis.

## 1.1 Notation

We use two types of references in this thesis. One is the standard citation reference (e.g., [ACP19]), the second is a footnote reference (e.g., this<sup>1</sup>). We use the former to point the reader to the relevant literature, be it an article or a white paper. The latter is used to provide a concrete identification of a resource being discussed.

Further, the figures created for this thesis use the following notation (see Figure 1.1):

**solid line** dependency or computation flow

**dotted line** inter-process/service communication

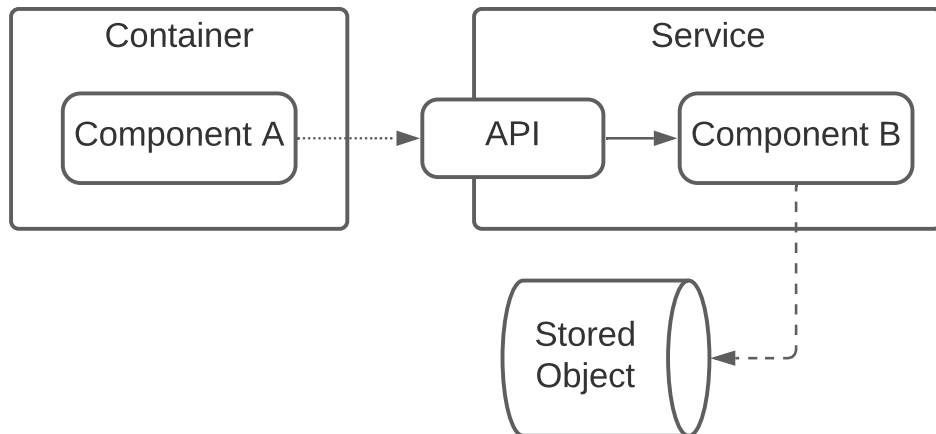
**dashed line** origin produces artefact at the end

**cylinder** stored object, most commonly a serialized model

**rounded rectangle** component

**sharper rectangle** container of components (app, service, ...)

However note, that not authored diagrams that were redrawn for clarity preserve the original notation.



**Figure 1.1:** Diagram demonstrating the conventions used in this thesis. *Component A* calls the *Service's API*, which uses *Component B* to produce the *Stored Object*.

<sup>1</sup>this is a footnote reference

## Chapter 2

### Scope delimitation and context

This chapter provides the reader with some background information about the MLOps (Machine Learning operations) field. It describes the Machine Learning lifecycle that is at the MLOps' core focus. Further points out the specific parts of the ML lifecycle which are important for this thesis and lastly explains the responsibility division of the interested parties.

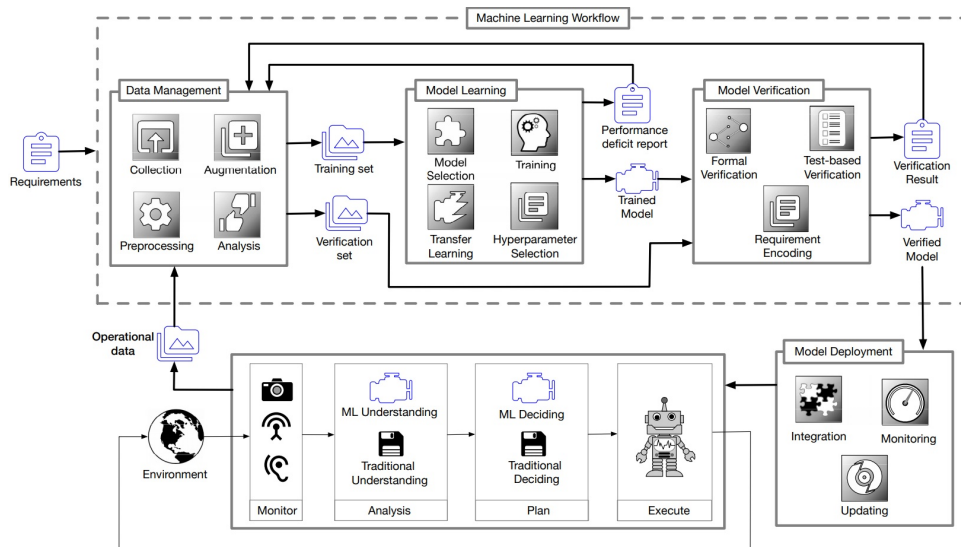
Machine learning operations concept builds on top of the DevOps[DPL15] movement. DevOps, shorthand for development and operations, is a concept reaching back to the early 2000s. DevOps are practices combing the development and the operations of the software development life cycle — the loop of planning, analysis, design, implementation, and maintenance. The core idea is to shorten the time from concept to release. MLOps extends this idea by applying it to machine learning, where additional problems appear, such as data management. Whereas DevOps is a blend of developers and operators, MLOps combines ML researchers, engineers, and operators. It aims to connect the development of machine learning with application/deployment.

#### 2.1 Machine Learning lifecycle

Machine Learning lifecycle [ACP19] describes the process of developing machine learning models and integrating them into the broader system. It consists of four stages — Data Management, Model Learning, Model Verification, and Model Deployment — depicted in Figure 2.1. **Data management** is the first stage and considers all things data before it reaches the model. In the **Model learning** stage, the model is designed and trained. **Model Verification** validates the model indeed performs as it should also with the data from the first stage. Lastly, the **Model deployment** in which the trained model is released and used.

In this thesis, the main focus is on the fourth stage — Model deployment. As part of the model deployment, we consider:

- Releasing/storing the model
- Integrating the model to the system (i.e., exposing it to be used)
- Running the inference on (managed) infrastructure



**Figure 2.1:** Four stages of Machine Learning lifecycle — Data Management, Model Learning, Model Verification, and Model Deployment.

Source: Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges [ACP19]

Enlarged: Figure B.1

### ■ Monitoring the model

These responsibilities are not solved directly in this thesis, but they are considered for the different approaches described in *Deployment scheme definitions* (chapter 4). However, some of the solutions described here reach beyond the Model deployment stage, as they innately solve problems from other stages as well (e.g., *End-to-end* (section 4.3) also possibly contains tools for model training and verification).

## ■ 2.2 Running inference

There are two ways a model can be applied to data. The first is called offline inference (also called static in some literature). The offline inference usually happens in batch mode, where the data is already collected, and the model is used to classify all the records.

The second way is called online inference (alternatively called dynamic inference). The difference to offline inference is that the data is coming into the system in real-time. This use case also benefits from micro batching for performance, but the pressure is on the data being evaluated immediately instead of being stored and classified later.

This thesis focuses on the latter — online inference. However, there are many problems and technologies dealing with the respective issues that overlap (e.g., model storage). The support for online inference can range from exposing the model as a minimal service, not offering anything more,

to providing a self-service platform with automation for model release and serving integrated with the training toolchain.

## ■ 2.3 Machine learning operations

The machine learning development process can be described as having two aspects — data science and engineering. Based on the two aspects, we assign the responsibilities based on the person’s expertise. The steps in the process are usually a combination of both, so the parties need to cooperate to reach the goal.

In this thesis, we understand the ML researcher/data scientist role to be responsible for the machine learning model. As the first step, they need to explore and understand the data and the desired goal to be achieved. Based on that, they design the data transformation pipeline, the model architecture (including the feature engineering), and select the metrics for measuring the model’s performance. They are responsible for the model’s output and limitations.

On the other hand, the MLOps role (Machine Learning operations) comprises software engineers that are responsible for the engineering side of the model development and assist the data scientists with engineering aspects of the tasks. That includes the infrastructure where the model is trained, the orchestration, and also the model deployment and how it is being served or accessed. The role’s goal is to enable data scientists to do their work and allow consumers to use the model.

A way to measure the cooperation of the roles is the MLOps Maturity model. In this thesis, we use the definition authored by Microsoft [mlo], but similar frameworks are described across the industry. In the Microsoft model, five levels are introduced (numbered from 0) that subsequently improve the level of automation and separation of concerns.

**Level 0 — No MLOps** Data scientists receive limited support and are responsible for the model from idea to serving. The majority of the tasks are done manually by the teams.

**Level 1 — DevOps but no MLOps** Data scientists receive some support. The majority of the tasks are still done manually, but software engineers assist or take responsibility for the model deployment.

**Level 2 — Automated Training** Automation is introduced for model training and directly related tasks (e.g., data ingestion). That improves the observability of the development stage through experiment tracking and robustness.

**Level 3 — Automated Model Deployment** Brings automation to the model release and improves the model integration.

**Level 4 — Full MLOps Automated Operations** The full process is automated, and a feedback loop is introduced to enhance the observability





## Chapter 3

### Schema evaluation criteria

We chose the following criteria to evaluate the different deployment schemas: scale, simplicity, completeness & extensibility. Each of them is described in more detail in a separate section below. We explain what purpose the criterion has, how to compare the deployment schema, and (when applicable) what to expect on either side of the spectrum. For some criteria, it can be easily said which "value" is better, but some describe the context best fit for an approach or when to avoid it.

Some of the criteria correlate to a degree not in terms of their definition or the informational value they provide, but that not all combinations would be usable. For example, a system with close-to-none extensibility but missing necessary functionality in completeness would not be usable in a production setting.

#### 3.1 Scale

The scale describes both the organizational scale as well as the use case scale. It describes the context in the respective areas in which the deployment schema would fit the best.

The *use case* scale considers the (possibly long-term) goals of the project. Some questions answered in this context could be:

- is the system intended for internal use only?
- do we need to test a hypothesis quickly to estimate the total investment in the project?
- is the system expected to grow in size soon?
- how robust do we need the system to be?

The *organizational* scale refers to the size of the team/company that participates in the project and the available resources they hold. A larger team can have a higher degree of freedom in selecting technology, as it has the human resources to deal with problems. On the other hand, they need the system to be operable by a more significant number of people, which introduces functionality and cooperation requirements.

This criterion links to the other criteria that it provides bounds to a certain degree for the trade-offs that have to be made for the schema to be viable for consideration. For example, on a smaller scale, the *simplicity* (section 3.2) and *completeness* (section 3.3) could be preferred for the cost of *extensibility* (section 3.4) as there are not enough resources (time or human resources) and it is required the system is easy to use and offers all the required functionality to reach the goal.

The scale criterion is important because it tries to prevent misuse of an approach based on the available resources. If a deployment schema is selected without the necessary resources, in the best-case scenario, they are only wasted; in the worst case, the whole initiative will be struggling with delivery deadlines and results. The second important value of the criterion is that it evaluates the "scaling" capability of the solution to the number of customers to balance the potential growth and the resources consumed.

## ■ 3.2 Simplicity

Even though simplicity is intuitively understood, we do not aim to provide only a "shallow" rating of a schema on a 3 point scale. The goal of the criterion is to bring specific problems (or lack thereof) to attention that are either baked into the core of the deployment schema or might be encountered later when for example, reaching the end of the *scale* (section 3.1) bounds. It aims to describe what to expect from both sides—the ML operations and the user (data scientist) perspective.

The topics in this section might include:

- complexity of the schema itself
- ease of operations/maintenance
- ease of use from the end user's perspective

## ■ 3.3 Completeness

Completeness provides a unified functionality overview of each of the approaches. Subsequently, it gives the reader an insight into how much extensibility will be needed to make up for the missing functionality. Completeness is hard to define explicitly and exhaustively, so in this criterion, we have a set of desired functionality from the production system that is considered of value as described in [ACP19].

- Integration
- Model lifecycle
- Monitoring

The *integration* describes how the trained model is integrated into the schema and how a specific approach is integrated into the other components of the machine learning pipeline. Partially relates to *simplicity* (section 3.2) as better, more straightforward integration means introducing new models is easier.

*Model lifecycle* is used to update the model's trained parameters that are being served. It can be as simple as manually redeploying the service with a new exported model bundled with it. Or, as advanced as an automatic rollout of several concurrent different versions of the model (e.g., release candidate, production, archived, ...) to multiple environments (e.g., production, canary, stable, ...).

*Monitoring* might be the most important out of the three listed functionalities. The other two can be compensated by manual work in place of an automated system. Monitoring, however, is used to collect metrics about models (possibly multiple versions): their input distributions, outputs, performance, availability, etc. More advanced systems can facilitate automatic anomaly detection or even discover training/serving skew to validate the model's behavior in the deployed environment.

Additionally, similarly to *simplicity* (section 3.2), we focus on negative delimitation as well when certain *nice to have* features are missing. For this criterion, we consider only the suite of functionality provided by the base version of the deployment or using the most common modules provided with it such that *extensibility* (section 3.4) is considered but is not the main driver of this criterion.

## 3.4 Extensibility

Extensibility is used to describe how easy or difficult it is to alter or extend the system's functionality.

The value provided by this criterion depends on the desired use of the system. It can be evaluated in terms of the range of how one can modify the system's behavior and how easy it is to do so.

A system can provide an effortless way to make changes due to the tech stack used. For example, with Python's WSGI[wsg] framework, it allows you to plug in any functionality, but you need to understand how to leverage that. On the other hand, the system can give you a comprehensive API that's hard to learn, but then all the operations are as simple as a single function call.

The other side of the spectrum can be represented by a solution that is entirely closed proprietary software where you are restricted only to the functionality provided out of the box.

There are cases where the extensibility is at the cost of *simplicity* (section 3.2) or *completeness* (section 3.3) (e.g., *Tech stack integration* (section 4.4) vs. *End-to-end* (section 4.3)).



## Chapter 4

### Deployment scheme definitions

In this chapter, we describe individual deployment schemas. The sections follow a similar structure where the scheme is first defined on a high level to understand the core concept. Then a *context* is provided to argue for the use cases where this scheme fits well, and subsequently, the scheme is evaluated with the defined *Criteria* (chapter 3).

We consider the following schemas:

- Native serving
- Software stack integration
- E2E/cloud solution
- Tech stack integration
- In-house custom solution

**Native serving** (section 4.1) is the baseline solution for serving a machine learning model. It exposes the model as a (web) service with an API to query it for inference.

**Software stack integration** (section 4.2) aims to embed a trained model into a specific stack such that it is directly available as a function call instead of evaluating the inference in a standalone service.

**E2E/ cloud solution** (section 4.3) deploys the models onto a (possibly self-service) platform that contains all the components necessary to do so. This platform is not developed or designed from the ground up, but a complete solution is used (e.g., a cloud solution).

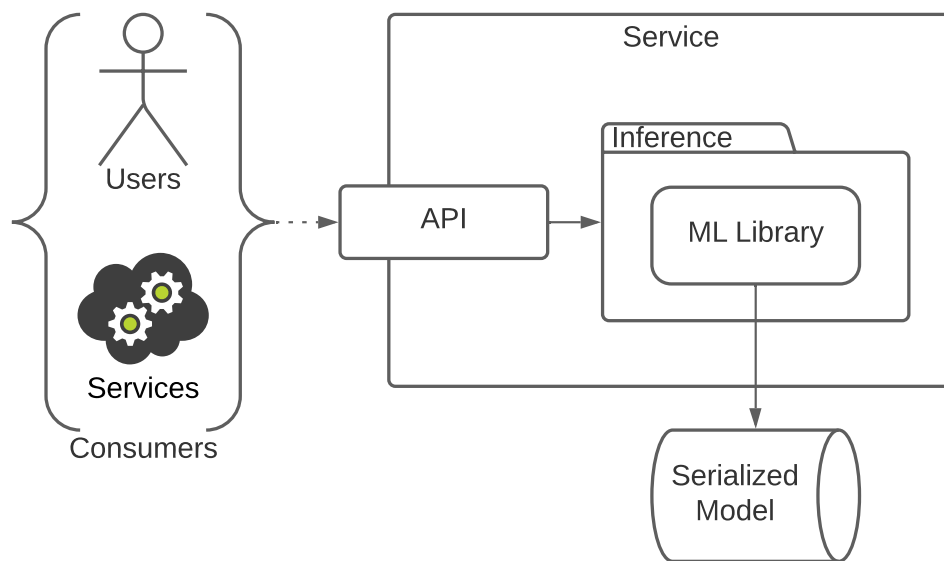
**Tech stack integration** (section 4.4) composes a custom platform for serving the machine learning models from available components to satisfy a functional requirement. Compared to the E2E approach, it selects the components that are the best fit for the whole use case and potentially leverages already deployed systems.

**In-house custom solution** (section 4.5) designs and implements a custom platform from the ground up tailored to the setting. It can leverage existing systems but is mainly created from scratch.

## 4.1 Native serving

Native serving is the most straightforward machine learning model deployment scheme on the list. It considers the situation where the model training is done via the same software stack as the serving, and then it is exposed to be used through an API for the final consumers. The architecture overview is given in *Native serving diagram* (Figure 4.1). Commonly an HTTP interface is used with a human-readable format (e.g., JSON) for simplicity, but more performant alternatives exist such as gRPC<sup>1</sup>.

The *MLOps Maturity level* (section 2.3) for this approach is mainly *level 0*, but higher can be achieved with available tooling. This approach can be handled manually at the bare minimum due to its simplicity (later described in subsection 4.1.3). However, nothing prevents this approach from being used along with more advanced software development tools and practices (i.e., level 1). Though later is described in *Context* (subsection 4.1.1) why advancing further too much is discouraged.



**Figure 4.1:** Native serving architecture diagram.

The clients issue requests to the API, which forwards the data to the initialized model for inference. The results are then returned to the client.

An example could be the industry-wide baseline — a Python [Ros95] stack where the model is developed using the Scikit learn [PVG<sup>+</sup>11]. The model is stored as a `pickle` file and then served using a WSGI web application framework like Flask[fla].

<sup>1</sup><https://grpc.io>

### ■ 4.1.1 Context

Native serving is the baseline of serving ML models in a production environment. It is usually the case where a decently performing model has been developed, and the next step is to make it available with minimal investment. This is achieved by introducing as little overhead in the deployment and serving as possible. The same technologies are used for both the development and serving as one can reasonably expect the libraries to have a format that enables both export and import of the models.

The important thing to note here is that this approach is recommended in such a restricted form because unless you need a specific implementation of particular functionality, there is a high chance a system exists already supporting it. Thus integrating it like in the *Tech stack integration* (section 4.4) would be more efficient.

### ■ 4.1.2 Scale

The scheme's scale is *small* for both the organizational and use case scale. The lower bound is not present in this case.

One could argue that just because the training and serving are done on the same platform, it does not necessarily mean that the system will be small. That is, however, not the specific case described by this approach. A clear distinction to, for example, *In-house custom solution* (section 4.5) has to be understood that this scheme is designed for the small scale.

The small use case scale is more of a soft bound. We do not recommend this approach for more extensive operations. In many cases, it would constitute reinventing the wheel of systems that would be easier to integrate with the resources needed to implement it. Examples of such functionality can be found in the *Tech stack integration* (section 4.4).

The small organizational scale is considered more from the provider's side, less from the consumer's. Even if the system were not the most performant, it could easily be scaled horizontally behind a load balancer, as the inference can be made stateless. However, having too many resources would be wasteful because either there is no use for it, or we can pick a more complex solution that gives access to a wider variety of features out of the box (e.g., *End-to-end* (section 4.3)).

### ■ 4.1.3 Simplicity

As mentioned in the general description, the crux of the scheme itself is very simple.

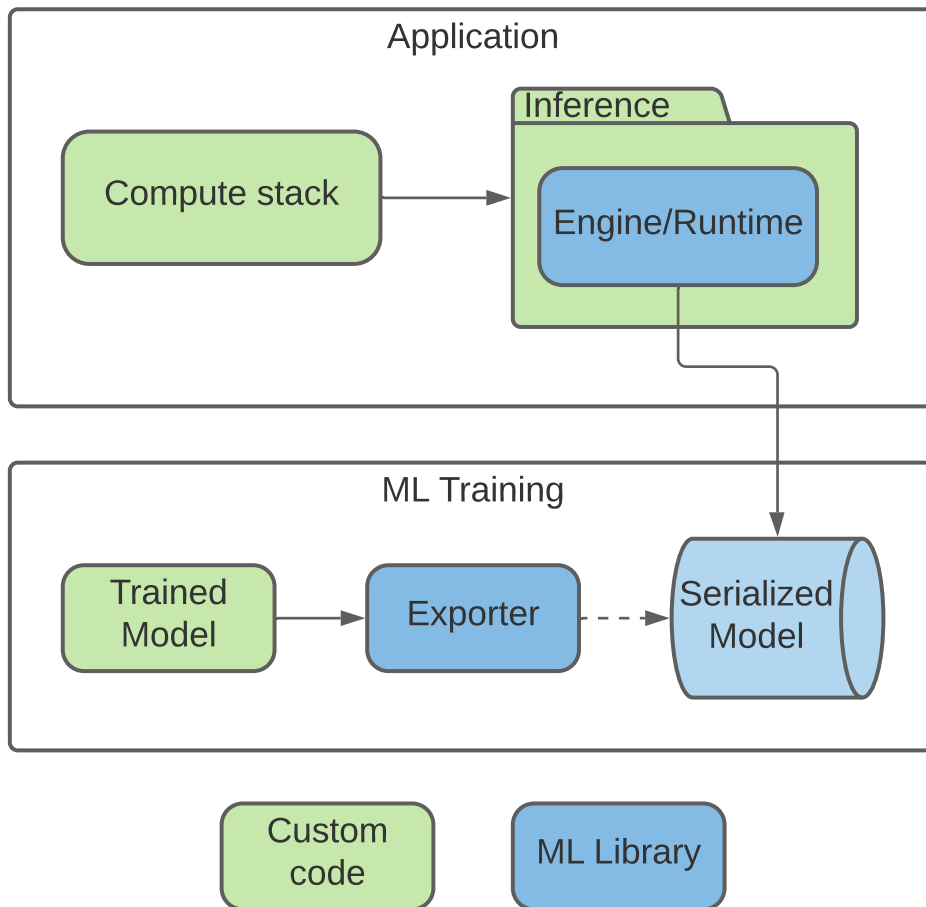
Since this approach consists of building everything from the ground up and mainly integrating existing libraries, the implementation simplicity directly relates to what we expect from the system. At the bare minimum, we need some API layer for the system to interact with the clients and instantiated model for the inference.





Two components need to accompany the intermediate format. The first is used to export a trained model to the format. This is specific for each ML library used, and thus attention needs to be paid before selecting the library such that it supports the ML library used for *training*. The second component is a runtime/evaluator for the inference that needs to be compatible with the *target system* (e.g., having a JVM implementation, Python bindings for a c++ library, ...).

The *MLOps Maturity level* (section 2.3) is closely related to that of *Native serving* (section 4.1) — *level 0* or higher, where advancing too much is discouraged except for special cases.



**Figure 4.2:** Software stack integration architecture diagram. Blue parts belong to the library, and green are implemented code.

The clear distinction for this scheme is the way the consumer interacts with the model. The model is exported to an intermediate format, and the inference happens in the actual application instead of it happening on a remote service.

The format can be human-readable (for example, in the case of PMML [pmm] the model is serialized to XML, and for PFA [Gro] it is JSON) or binary (in case of ONNX [BLZ<sup>+</sup>] the model is stored using Protobuf<sup>2</sup>).

<sup>2</sup><https://developers.google.com/protocol-buffers>

### ■ 4.2.1 Context

A business need can occur that prohibits using the service-oriented *Native serving* (section 4.1) scheme while still requiring/expecting small-scale operations. An example can be a system that needs to be isolated from the internal network (running in a sandbox), an application that might not have network access, or generally when microservice architecture is not an option.

Another use case for exporting a trained model to an intermediate format and running the inference in an evaluation engine is runtime performance. As noted in [Sta21] — ONNX runtime was used to speed up the inference of a PyTorch<sup>3</sup> model used in Natural language processing for detecting phishing emails.

### ■ 4.2.2 Scale

The scale this approach fits may be unbound. The simplest case of having a few models integrated into their respective services can be handled manually and individually, although this would be discouraged for longer-living products.

Even though the scheme scales well since the workload is distributed across projects, at one point, there would be a lot of work repeated. This naturally leads to improving the tooling around this approach to ease the integration from the consumer’s perspective. However, serving the ML model from a service provides benefits such as independent instance scaling, better resource management, development isolation, etc.

### ■ 4.2.3 Simplicity

The scheme itself is also trivial to understand as it is mainly dictated by the use case.

The additional overhead of porting the model and integrating the runtime increases this scheme’s implementation difficulty compared to the first approach. However, implementing it is still relatively simple. The integration consists of selecting a library that supports both programming languages (training and serving), and a generally good practice is to wrap the library with a custom interface. The integration pain point is usually preparing the data for the model to ingest as the format/type system needs to be generic enough to support various data structures, or the types are omitted.

### ■ 4.2.4 Completeness

A problem specific for this solution is the required feature completeness compatibility of the training environment with the runtime. As those are two different technologies, a specific operation (for example, the BIRCH clustering algorithm in the case of ONNX and SciKit-learn [omn]) may be offered differently in each of them without a compatible way to transform between the two definitions or that an operation is missing entirely (for

---

<sup>3</sup><https://pytorch.org>

example, when a new version of the library used for training is released until the runtime implements the operator as well).

The application evaluating the model is responsible for the model lifecycle implementation. For example, it can load the model during startup, and thus to update the model, we need to restart the whole service, or it can implement a mechanism to provide a way to update the model during runtime. However, this can lead to the problem of multiple trained versions of the same model being live concurrently, as was identified in the *Socialbakers case study* (section 6.2).

The model monitoring is problematic. Of course, it can be monitored along with the service itself as part of the metrics that it produces or even deal with the problems directly. However, any other form of monitoring poses a problem since we lack any centralized access to the model. This makes it hard to detect some problems easily detected in the other schemas, for example, if the input distribution changes.

### ■ 4.2.5 Extensibility

The extensibility of this approach closely matches that of the *Native serving* (section 4.1). Because the system is customly implemented, anything can be changed. The only limitation is the intermediate format and the engine to evaluate it. However, many libraries (such as the mentioned ONNX [BLZ<sup>+</sup>] or PMML [pmm]) enable the user to implement custom functions/operators that allow to make up for it to some degree.

## ■ 4.3 E2E/cloud solution

End-to-end or potentially cloud solution leverages an existing full-fledged platform for deploying and serving an ML model, possibly managing the whole machine learning lifecycle.

The realization can take two approaches. First, using a managed system that is provided by a vendor — cloud or as a service. Second, an end-to-end platform is deployed and managed by the MLOps team.

The *MLOps Maturity level* (section 2.3) depends on the specific system deployed/service provider chosen and the integration. However, having a platform with maturity level 1 or less would not defend the added complexity and cost over the, for example, *Native serving* (section 4.1) approach.

### ■ 4.3.1 Context

The end-to-end solution takes a holistic approach. It uses a platform providing all the desired functionality "out of the box" while also coupling the individual components together. What remains is providing tooling (or even just mediating it) to the users (ML researchers) and integrating the platform to the wider system.

The end goal is to make the system available to the users as a self-service. Assisting when necessary, of course, but otherwise, the platform should allow the user to manage their projects independently. A desired functionality is having the means to monitor and restrict users' resources for isolation and budget overview.

### 4.3.2 Scale

The managed platform can be used on a small scale, both in terms of use case as well as organizational. As it will be noted in *Simplicity* (subsection 4.3.3), the managed platform can be set up by a single person, with an initially restricted functionality, and the overhead is negligible, which enables using this approach even for a smaller number of models.

On the other hand, deploying an end-to-end system like this is not trivial. Other than the same setup as the managed platform, it is necessary to configure and deploy the actual system. That can be pretty difficult because the system encompasses a lot of features (and possibly components). Because of that, the resource investment for setting it up is higher, and thus for it to have value, the expected use case volume should be higher.

### 4.3.3 Simplicity

When using a managed platform, setting up an example project is simple. However, this becomes more complex later based on the production qualities required and expected to be integrated into the ecosystem, although the features are usually "just a click" away from being operational.

As described in *Scale* (subsection 4.3.2), when manually deploying the platform (to an on-premise cluster, for example), the simplicity of having all the necessary components becomes a hurdle as all need to be correctly configured and connected. After the system is running, the same setup as with the managed platforms is required.

### 4.3.4 Completeness

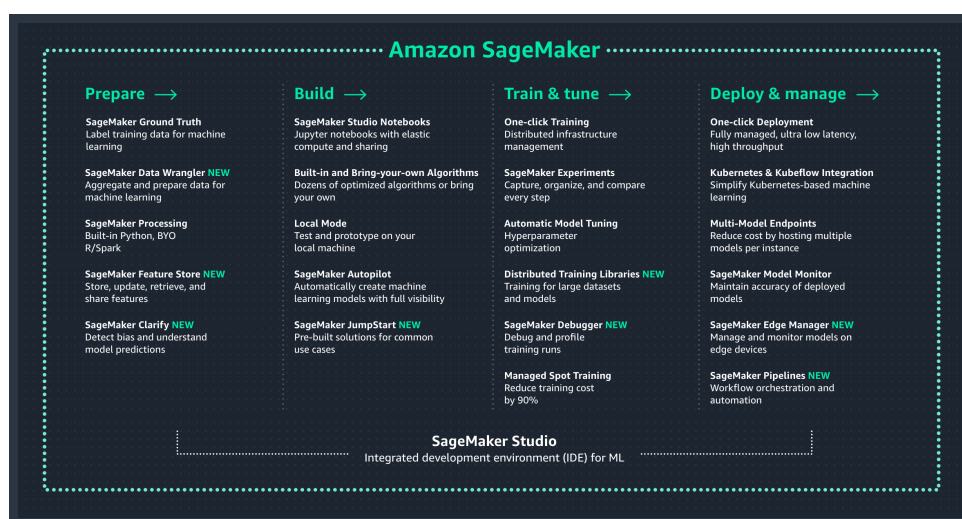
Both managed and manually deployed approaches can be considered offering a complete set of functionality as outlined in section 3.3. As of writing this thesis, an example can be the Amazon SageMaker<sup>4</sup>, an ML platform from AWS<sup>5</sup>(Amazon Web Services) with support for the full ML development process (Figure 4.3).

The most significant advantage of the single platform approach is the natural integration of the system to the other parts of the machine learning pipeline. For example, in the case with TensorFlow Extended [MKF<sup>+</sup>17], a component named TensorFlow Pusher<sup>6</sup> integrates the training part of the

<sup>4</sup><https://aws.amazon.com/sagemaker>

<sup>5</sup><https://aws.amazon.com>

<sup>6</sup><https://www.tensorflow.org/tfx/guide/pusher>



**Figure 4.3:** Functionality offered by the Amazon SageMaker. Note that AWS changes its product offerings frequently, and the same functionality can be provided under a different name in the future.

Source: <https://aws.amazon.com/sagemaker>

Enlarged: Figure B.2

pipeline to the TensorFlow Serving<sup>7</sup> component for the model to be available for online inference.

One notable drawback of some of these solutions (e.g., TensorFlow Extended or PyTorch Serving<sup>8</sup> and the surrounding ecosystem) is they are not universal. Although they provide an excellent user experience because they are tailor-made for a specific ML library, they do not support any other. This means that it allows only users of those specific libraries to leverage that infrastructure. If any other team requires a different ML library, they are either out of luck or a whole another system has to be deployed and maintained. So it is a tradeoff between the out-of-the-box integration and the generality of the platform.

### 4.3.5 Extensibility

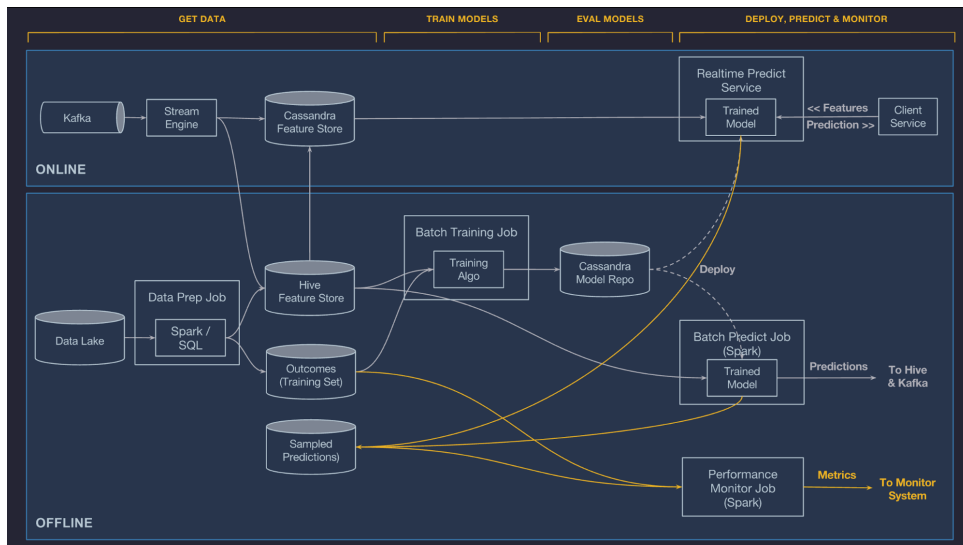
For the managed platform, the customization is minimal, if possible at all. Usually, cloud products are black boxes that can be configured to an extent, if at all, but cannot be extended. This is even more the case when leveraging a platform's integration of its components, like in this case. The core part where the modifications come into place is the built tooling surrounding it. It can provide a completely custom interface, limit functionality, and apparently extend the behavior by combining multiple services.

Modifying a custom deployment of an end-to-end platform is possible on two levels but depends on the platform's components. On the higher level, some platforms allow the user to exchange the actual components that perform certain functionality (but comes in with compatible solutions as compared to

<sup>7</sup><https://www.tensorflow.org/tfx/guide/serving>

<sup>8</sup><https://pytorch.org/serve>





**Figure 4.4:** Architecture of Uber’s Machine learning platform Michelangelo. At the top of the diagram, there are stages marked in orange, and we can see how the individual components interact. Source: Meet Michelangelo: Uber’s Machine Learning Platform [HB] Enlarged: Figure B.3

#### 4.4.1 Context

An organization just expanding to use machine learning in production systems presumably already has several technology platforms. This might make it less desirable to invest in an *End-to-end* (section 4.3) solution as it might duplicate the functionality that is already provided and integrated into the tooling and the surrounding ecosystem.

This deployment scheme uses the fact that the systems are maintained by other teams and focuses on catering to a platform that eases the user’s ML tasks. This means that the user is expected to take advantage of the present tools to assist them with their goals supported by the MLOps team either directly or indirectly by the implemented tools.

#### 4.4.2 Scale

It follows from the scheme’s description at the beginning of this section that the minimum recommended scale for this scheme is a dedicated MLOps team with multiple members. However, as the result comprises multiple systems cooperating and leveraging the current ecosystem and tooling present, a whole organization is expected to be present behind it. That subsequently allows the MLOps team to focus the resources on the actual platform instead of maintaining the dependees. Otherwise, this approach would bring problems that are innately solved by a managed *End-to-end* (section 4.3) platform instead.

It naturally follows from the nonmarginal resources required to create the





that would be thrown away if all the individual systems would have to be managed and deployed from scratch.

The other degree of modification is on the level of individual components and between them. As the platform comprises multiple independent systems, they are bound to have an interface between them where a proxy service can be inserted, providing additional functionality. However, a more direct way to interact with them can be expected.

## 4.5 In-house custom solution

As the name suggests, this approach comprises creating the machine learning deployment platform from the ground up, with most of the components implemented from scratch. Because this scheme describes an approach that fits a specific use case or context, there is no general architecture that can be presented here.

The significant difference to the *Tech stack integration* (section 4.4) is the functional integration and the design of the components. A presumably large part of the platform's core services needs to be implemented because it either does not exist or integrates with a specific selection of libraries would be as demanding as implementing a better fit solution.

The difference to the *End-to-end* (section 4.3) is that the MLOps team does not just deploy and maintain the platform but also authors it. For example, the case of Google with the TensorFlow Extended [MKF<sup>+</sup>17] – although an end-to-end platform, it was developed internally and later released for public availability.

The *MLOps Maturity level* (section 2.3) depends on the design, implementation, and integration of the platform and its components. Levels 0 and 1 would, in this case, be closer to the *Native serving* (section 4.1) approach, so it can be reasonably assumed that such platform is designed with the aim for ML operations being level 2 or higher.

Beyond the examples mentioned above, another example can be the LASER [ALT<sup>+</sup>14] system developed at LinkedIn. The goal was to make large-scale machine learning possible in terms of data and the number of models (to support cold model parameters trained on the whole dataset and hot parameters that are trained on the data specific to a user or a running ad campaign). The solution was a restriction that only Logistic regression where only specific input variables interactions are allowed. This restriction allowed the system to be optimized for the specific use case in terms of performance and was integrated into the local ecosystem.

### 4.5.1 Context

There are multiple situations where implementing a custom solution mainly from scratch is a viable solution.

The first that comes to mind is when such solutions did not exist at all. Of course, as mentioned several times in this thesis, there are a plethora

of available implementations now. However, if we look at the years these systems were published, we can see that they are very recent in their origin. Some examples are — TensorFlow Extended [MKF<sup>+</sup>17] 2017, Amazon SageMaker [sag17] 2017.

Another case is when the use case requires it or dramatically benefits from a custom approach. An example can be the aforementioned LASER from LinkedIn that tailored the platform for a specific kind of models. Alternatively, a platform in big corporations (such as Facebook) where the optimization for the fit saves cost due to the large user and customer base.

### ■ 4.5.2 Scale

It is expected that most components have to be developed from scratch as otherwise the platform could be composed of them as in the *Tech stack integration* (section 4.4) approach. The cost of creating a completely custom platform for serving machine learning models is not trivial. It requires a large group of people to develop and maintain. Thus, it makes sense to invest in such a thing only when the scale demands and allows it.

As this solution can be considered a product on its own (for example, from the point of view of *End-to-end platform* (section 4.3) as a result as is the case with TensorFlow Extended), it is expected to have multiple dedicated teams from all areas of the software development process.

Both arguments imply that the scale is recommended not only for a larger team but also for the whole organization. The problem with not enough resources would mean that the platform's development would take a long time and thus hinder the whole project/cause as the other initiatives would be blocked by it.

### ■ 4.5.3 Simplicity

Intuitively this scheme is not simple, but indeed very complex. Not only does the system as a whole have to be architected, but the individual components need to be designed, implemented, and integrated such that it balances the completeness and openness to extension in the future (or even from the other users if it is expected to be released publicly). Another hurdle in the overall design is the user experience that has to be considered during the design phase, so the solution fits the desired use case well.

### ■ 4.5.4 Completeness

None of the other approaches described here are as specifically complete as this one. This naturally follows from the motivation of building a completely custom solution. Further, even other functions that are not generally applicable have to be considered here. One can reasonably assume that if the implementations of the other schemas described here sufficed, there would not be an incentive for developing a custom platform from scratch. However,

this also means that some features might be missing as the use case might not require them.

#### ■ 4.5.5 Extensibility

The range to which the platform can be modified is unbound since it is developed from scratch. On the other hand, the difficulty of the modifications can be considered higher than the other approaches. This springs from the size of the project and the available "degrees of freedom." It can be reasonably assumed that any open-source component used in the other approaches is designed with how and to what extent it can be modified by the user (MLOps member in this case). This makes it a requirement of the design process to include such a decision. It can be minimal or, on the other side, provide a plugin API to introduce an arbitrary functionality. However, since the custom system might not be designed to be deployed by other users, the changes could very well be integrated into the core. This alleviates the complexity of designing such a mechanism with the cost of dealing with the problems when the situation occurs.



## Chapter 5

### Deployment scheme selection

In this chapter, we compare the deployment schemas described in the *Deployment scheme definition* (chapter 4) and introduce a guide on picking the best approach. Two frameworks are presented here, each having a different end goal.

The first framework minimizes the cost (in terms of time and resources) it takes to show results, which is helpful when either experimenting or the goal is to have the model served as soon as possible. Its aim is not to provide a mature self-service platform but rather to enable further steps.

The second view focuses on the case where the intent is to invest into a platform with a longer-term commitment which is naturally accompanied by more requirements on the platform itself.

Alternatively, the two somewhat orthogonal frameworks can be viewed as another manifestation of the traditional software engineering dichotomy — speeds vs. quality. The first one optimizes for speed at the expense of quality represented as robustness and feature completeness. The second accepts the resources it takes to build a mature platform that satisfies advanced requirements.

#### 5.1 Minimize (initial) investment

The first view aims to minimize the time and effort it takes from having a trained machine learning model to serving it. It fits into the early development stages, where the end goal might not be determined yet. We identified two main scenarios where using this view is beneficial.

The first scenario is when serving the model is a necessity and not the primary objective. This can happen when the aim is to develop the model and not produce any specific system while still needing to interact with it. The second scenario targets a situation where an engineer (or a team) needs to enable the data scientist to deploy the model. In this case, the model is indeed to be served as the end goal.

The criteria most important for this view are *scale* (section 3.1) and *simplicity* (section 3.2). It is required from the scheme that the scale enables a small team or even just a single person to be deployed. If a scheme is recommended for a bigger minimal scale, it increases the effort needed

to implement it both in terms of time and resources, which hinders the goal of having results fast. The simplicity accompanies that with a similar sentiment. The scheme is expected to be easily understood and operated as well; otherwise, it might be a wasted effort because of the undetermined end goals.

### ■ 5.1.1 First option

With the goal being a short time to production at a low cost, the *Native serving* (section 4.1) dominates all other deployment schemas. Depending on the solution's required robustness, it can be a matter of a single afternoon having an instance serving a model. The recommended upper bound for this solution (when another scheme would be better suited for the case) depends on the software tooling available. If methods enabling production qualities such as service monitoring and deployment automation are available, it can go a long way, and there can be cases when it suffices completely.

### ■ 5.1.2 Second option

The other option that qualifies for this context is the *End-to-end* (section 4.3) approach. Although with nontrivial initial time investment in learning the system's ins and outs, the basic functionality is easy to set up once one is familiar with it. The benefit is the growth potential in supported functionality once it is desired. A managed system (i.e., a cloud service like Amazon SageMaker) would be easier to set up for prototyping than deploying a system manually.

Of course, with the goal being a fast start, it will not be the self-serve platform (described in subsection 4.3.1) at first. That would require a more significant initial investment of resources. This scenario focuses on enabling the deployment of the models.

### ■ 5.1.3 Discouraged

Except for *Software stack integration* (section 4.2) that could arise from the project's constraints, the rest of the deployment schemas described in chapter 4 are not a good fit for the initial stage. The other schemas require more considerable investment before reaching a usable state.

## ■ 5.2 Providing self-service platform

The second view selects a deployment scheme that aims to be (or become eventually) a self-service platform. The assumption is that there is a goal set and available resources to be committed, which leads to the requirement that schemas can not be upper bounded with a small scale lifted. This also creates a circular requirement that the service should aim to at least eventually be self-service as the scale can grow.

As the scale is lower bounded, it enables us to tackle the scheme selection without regard for simplicity as a significant criterion which leads to having to balance between *completeness* (section 3.3) and *extensibility* (section 3.4).

### ■ 5.2.1 Completeness first

The *End-to-end* (section 4.3) is the most complete scheme out-of-the-box from the schemas described in chapter 4. There are platforms providing all the functionality required from a self-served production platform. The benefit over the *Tech stack integration* (section 4.4) is the minimization of glue code (which is regarded as a significant factor for technical debt [SHG<sup>+</sup>15] in machine learning systems) as the components come integrated, and the resources for development can be focused on providing the tooling around it.

A wide range of available platforms are described in [Helb] and [Hela].

### ■ 5.2.2 Extensibility first

As described in subsection 4.3.5, even though the End-to-end approach can be modified/extended to a certain degree, you are restricted in terms of the location for the changes. The *Tech stack integration* (section 4.4) allows you to have modifications on a much finer level as you design the platform. Maybe it is specific lifecycle management rules or a way to access the platform. Since you pick the components and integrate them, you gain an additional level of tuning for the platform.

As the solution consists of selecting and integrating components as the use case demands, there is an infinite number of possible configurations. A complete example can be the MATS stack [DSK] from Avast (described more in-depth in section 6.1) or, Michelangelo from Uber [HB].

### ■ 5.2.3 Special use case

In the industry, there are several cases where a custom platform had to be developed. An example of a general platform can be the aforementioned TensorFlow Extended [MKF<sup>+</sup>17] from Google, which was motivated by the lack of available offerings at that time. Another example is the LASER [ALT<sup>+</sup>14] used at LinkedIn because of the requirements on the amount of data and number of models.

This approach is usually chosen in corporations with enough resources available for development and the need for such a system before open-source projects were available/mature enough for their scale.

## ■ 5.3 When to switch framework selection

From the *Case studies* (chapter 6) described later, we can observe a pattern that usually the companies start with the *Native serving* (section 4.1) in some form. It can be as simple as a service manually being deployed to a VM over

`ssh` or more advanced if the infrastructure allows it, leveraging automation and DevOps practices.

Further, we see two situations that can trigger the pursuit of a more mature platform. First is the planned future growth of online inference workloads, as we can see in the *Avast case study* (section 6.1). Due to the history of software engineering, we can expect the complexity to increase with it. Thus the aim is to prevent the engineering from blocking the business growth. The second case is when the technical debt accumulates during the slow workload increase over time and makes further progress cumbersome; see the *Socialbakers case study* (section 6.2).

Supporting this, in Table 6.3 we can see that *Native serving* (section 4.1) is preferable with lower number of models (five or less) while *Tech stack integration* (section 4.4) and *End-to-end* (section 4.3) have preference with increasing numbers of models.



## Chapter 6

### Case studies

In this chapter, we explore two industry use cases for machine learning deployment platforms. The first case study describes ML operations at *Avast* (section 6.1) where we observe the transition from the probing phase of machine learning to establishing a dedicated MLOps team implementing the *Tech stack integration* (section 4.4). Secondly, the *Socialbakers* (section 6.2) case study explores another implementation of the Tech stack integration with an asynchronous event-driven online inference design. As part of the case studies, we conducted a survey presented in section 6.3 to observe general trends in the industry.

Important components of the individual solutions are introduced more in-depth in *Tech glossary* (Appendix A).

One of the case studies described the machine learning operations at Cisco Cognitive Intelligence, the AI/ML organization inside Cisco. However, due to creative differences, we haven't received permission to publish it alongside the other case studies presented here.

#### 6.1 Avast

The first case study describes the Machine learning operations at Avast<sup>1</sup>. Avast is a Czech cybersecurity company with over 30 years long heritage. Its signature product is the Avast antivirus; however, over the years, through acquisition and innovation, Avast expanded from local machine malware detection into other fields and is a cybersecurity and privacy powerhouse.

Avast has recently increased its ML/AI focus which led to improvements in Machine learning operations. We first describe the original state of the matter. Follows a brief explanation of what changed the approach Avast took towards ML. Lastly, the current architecture is described, including future goals that are not yet implemented. Two examples are presented to show the original architecture and the different integration in the new platform.

Along with the higher focus on ML/AI, a bigger budget to support the larger scale and increased expectations for results came with it. Currently, there are over 30 production models deployed from around ten projects. Out

---

<sup>1</sup><https://www.avast.com>

of these models, approximately 20 are used in customer products, and the rest is used for internal services.

### ■ 6.1.1 Initial Solution

Before the focus shift towards machine learning happened, there were long-running initiatives to determine what value machine learning could bring to the existing products. They were bound by projects running at that time and were aimed at whether it is possible to improve the functionality and not provide a completely new feature. An example is given later in this section, where an ML model was used to experiment by extending a rule-based system.

As it is usually the case with long-running (R&D) experiments — they receive limited support from engineering/operations. Data scientists on ML projects were expected to leverage already deployed software systems and using the present infrastructure. Introducing new services supporting the exploration was not in the current scope, which limited the options available for the ML researchers.

Data scientists' primary focus is producing well-performing models, which means lower attention is paid to the engineering side of deploying them. From that and the limited support they received for introducing new technologies, it naturally follows that *Native serving* (section 4.1) approach for exposing the models emerged (i.e., simple service manually run as a container), as the serving was a hurdle, not an objective on its own.

The researcher's workflow would usually consist of model development on their local machine, including training and validation. Then the model would be manually deployed to the service for serving, without any tracking of the experiments. This would imply the MLOps Maturity (section 2.3) level 0.

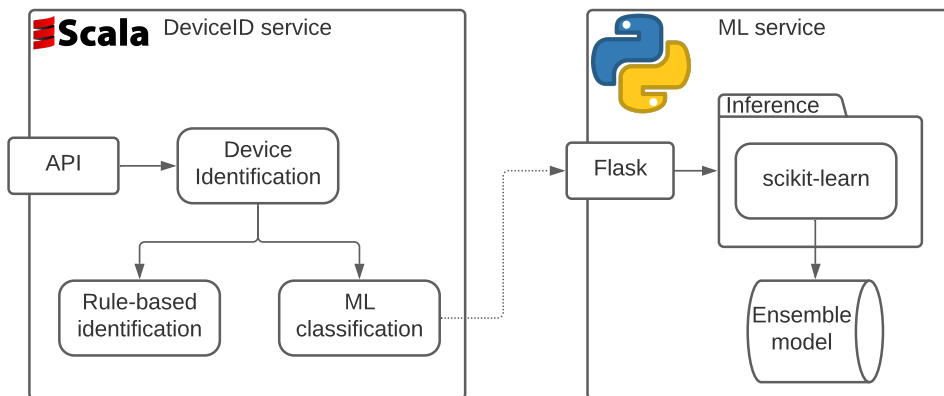
Even though software development tooling was available in the company, there was no standardized way for machine learning projects to use that. It was at the volition of the researcher and their incentive to do so. Thus the Maturity level was 1 in some cases but dependent purely on the researcher's engineering predispositions.

### ■ Device Identification service

The first example (Figure 6.1) is a project that identifies device class (e.g., smart TV, printer, ...) based on its network traffic. The high-level architecture is a service that uses two inputs for the identification — a rule-based decision system and an ML model. The rules are evaluated locally, and the ML model is exposed as a simple web service (*Native serving* (section 4.1)).

### ■ URL phishing detection

The second example is the project internally called *Angler* — a URL phishing detection using TensorFlow. Compared to the DeviceID project, Angler would classify as MLOps maturity level 1. The individual segments of the pipeline were automated (training and deployment) using *TeamCity* (section A.1), but



**Figure 6.1:** Legacy architecture diagram for the Device Identification service. The *ML Classification* package is an HTTP client calling the ML service’s endpoint for inference.

the artifacts were handled manually, and when the production model changed, first, the serving configuration needed a manual update, and then manually trigger the serving deployment. It was using TensorFlow<sup>2</sup> to implement the ML model and TensorFlow Serving<sup>3</sup> for exposing an HTTP API for inference.

The Angler project was a typical example of an ML pipeline in the original setting. It was one of the core projects during the design phase of the new platform. The MLOps team temporarily adopted the project. Based on its requirements and architecture (but of course, also of other projects), the new platform was designed.

## 6.1.2 Transition

The trigger of change for MLOps in Avast was a series of leadership changes. Originally, machine learning was on a sidetrack from the other production projects and was only explored for its potential. The changes in leadership led to the increased focus on machine learning as a first-class tool to solve current tasks.

The motivation for ML projects changed from experimentation and exploration to delivering production results. Production systems have higher requirements than development projects. The systems are expected to be reliable, scalable, and with high-quality codebases, which meant more attention needed to be paid to the engineering aspects.

The change brought a higher budget for the ML initiative, which led to establishing a designated MLOps team. As with any new project, there is a phase of experimentation at first, which naturally happened in Avast. Over time, as experiments were underway and more knowledge was gathered, a standardization emerged.

<sup>2</sup><https://www.tensorflow.org>

<sup>3</sup><https://www.tensorflow.org/tfx/guide/serving>

### 6.1.3 Current MLOps platform

The motivation for the new platform was to support data scientists in a standardized and structured way. The structure eases the communication between MLOps and ML researchers. Further, standard project structure allows ML researchers (and MLOps also) to work on another project faster, focusing on the issues at hand and not wildly browsing through the repository. An example supporting why common project structure is beneficial can be the Apache Maven project[MVM10], which introduced the directory layout for Java/JVM projects[mav] that is the de-facto industry standard as other build systems adopted it as well[gra][sbt].

The standardization, however, also brings some restrictions. The major limitation is that not all technologies are available. Using a new system isn't just deploying it, but it needs to be integrated into the ecosystem to conform with the structure in place. Otherwise, the system would accumulate technical debt and would eventually be more expensive to maintain than the value it would bring.

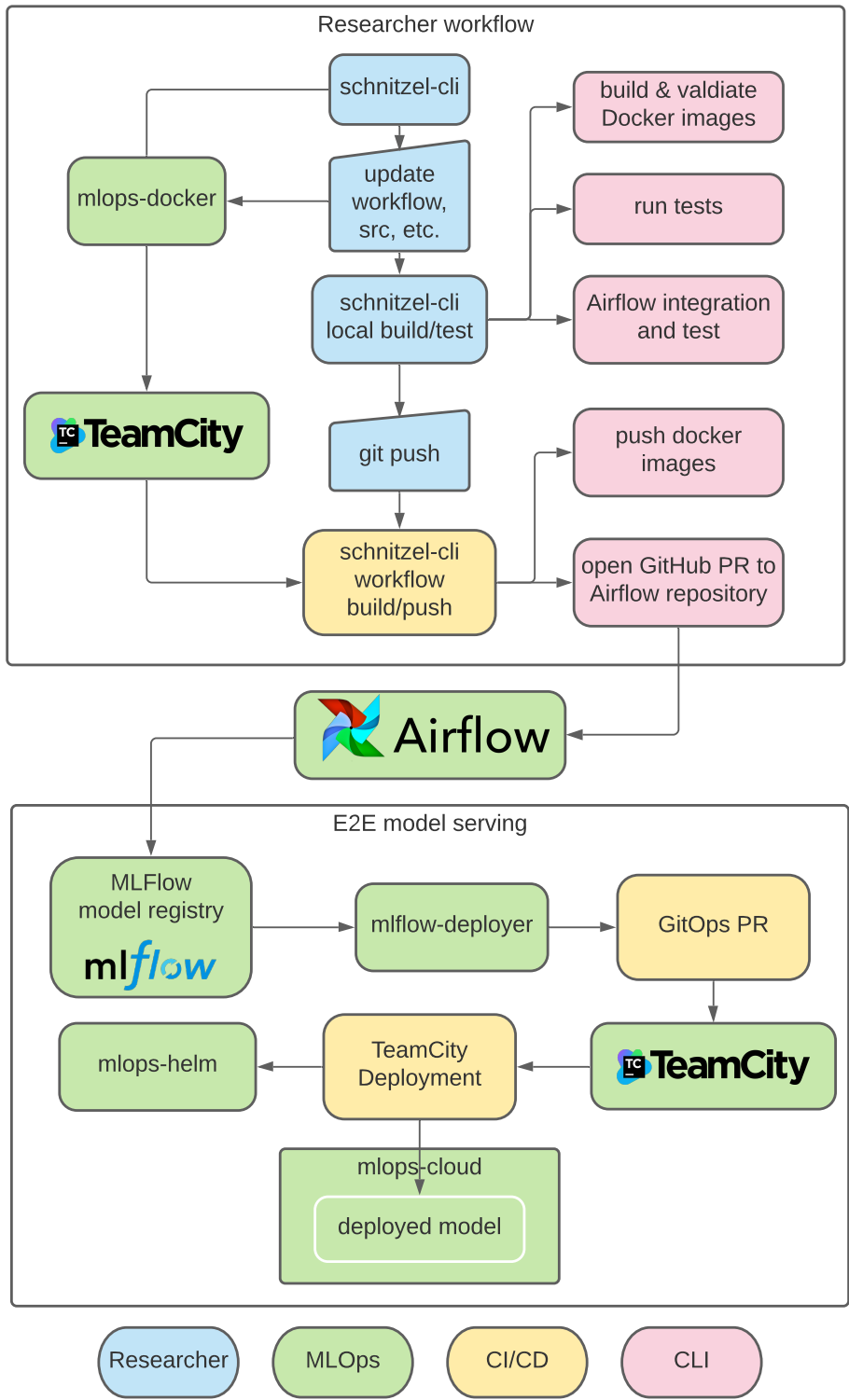
The end goal of the initiative is to provide a self-service platform for the whole machine learning lifecycle. This supports the standardization as before everything was done on a per-project basis, but now common tooling will be available to all projects. The ML researchers should be able to work with the platform independently. The MLOps team should provide support, maintenance, and drive the platform forward, but the researchers should not need direct assistance for everyday tasks.

The created platform (named Schnitzel) fits the *Tech stack integration* (section 4.4) approach. It comprises several open-source components and few custom components that bind everything together to provide a single platform experience for the ML researchers. The primary reason for selecting this approach was the existing technical infrastructure. There were already deployed systems supporting software development that can be leveraged for MLOps development. Hence, the idea was to extend it to support machine learning better. Furthermore, the present compute infrastructure also swayed the decision from using a cloud solution to running the platform on the on-premise servers.

### Architecture

The new architecture, displayed in Figure 6.2, originated from the MATS stack[DSK]. There are two parts to the diagram — the top half describes the user interaction with the platform, and the bottom half displays the systems behind it.

Build automation for the platform is done via *TeamCity* (section A.1) as it is generally used at Avast for build management and continuous integration.



**Figure 6.2:** Architecture diagram of the Schnitzel (MLOps) platform at Avast. The top part depicts the process and user interaction. The bottom is the component interaction.  
 Source: Avast  
 Original: Figure B.4

The integration of the platform for the user (ML researcher) is implemented via the `schnitzel-cli` — a command-line interface that shields the user with a custom facade from the systems behind it. This utility unifies the platform’s view — it is hidden behind tooling and isn’t exposed as a service. However, the users are expected to work directly with the systems in some cases (for example, with *MLFlow* (section A.2) to track experiments).

In the center of the diagram, there is the *Apache Airflow* (section A.3). Apache Airflow is used for cross-system orchestration of the workflows, for example, the model training. The main reason to choose Airflow versus other workflow managers is its support for running jobs on multiple clusters, which was required since Avast has multiple computation clusters for use cases outside of machine learning.

Next, there is the *MLFlow* (section A.2) component. The reason behind choosing MLFlow is its simplicity in both the use for the ML researchers and deployment. MLFlow has two uses in the Avast platform. The first is experiment tracking during training. ML researchers can track experiments (by producing graphs and metrics) and validate the models based on that. The second is as a model repository for trained models. ML Researchers can store the models and promote them to different phases of the model development (e.g., `dev`, `prod`, `archive` ...). Further, it provides a standard format for packaging models supported by various tools for model inference.

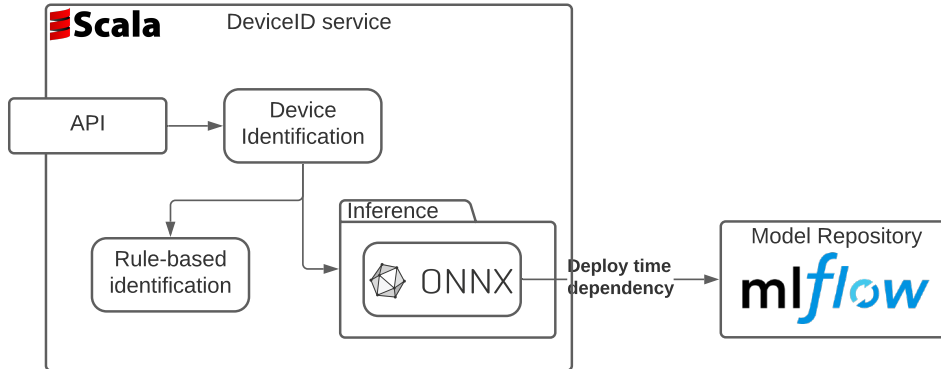
MLFlow Deployer is a custom service that monitors and deploys the models stored in MLFlow based on their stage. It does so by periodically comparing the currently deployed state with the status in MLFlow. When a model is promoted by the ML researcher in MLFlow (e.g., from `dev` to `prod`) a TeamCity deployment is triggered. The serving is done on a *Kubernetes* (section A.4) cluster via several serving services, for example TensorFlow Serving or *Seldon* (section A.5). The deployment uses a custom set of *Helm charts* (subsection A.4.1) templates to deploy the model to cluster for it to be served.

Although we can see the usage of TensorFlow in the Angler project, TensorFlow Extended listed as an example of the *End-to-end* (section 4.3) approach is not considered in the new architecture. The reason behind it might be the diverse nature of the running projects. TFX would provide an excellent experience for a TensorFlow project; however, other projects would still require a comprehensive platform to deploy models. That would lead to maintaining and integrating multiple production systems and shows a significant disadvantage of the end-to-end platforms created for a specific ML library.

### ■ Device Identification service integration

In the new architecture, the Device identification service (architecture in Figure 6.3) no longer has the manually deployed ML service. Because of performance reasons, the model is being exported to the ONNX format and is evaluated in the main Device identification service. The model is stored in the MLFlow model repository. Once promoted to be the latest `prod` model, the MLFlow Deployer changes the definition in the DeviceID repository to comply

with the project’s bi-weekly release schedule. This practice is commonly called *GitOps* (section A.6) — declarative environment/infrastructure definition with automation through versioning system and continuous deployment tools.



**Figure 6.3:** Architecture of the new Device identification service. The service now embeds the model evaluation using the ONNX [BLZ<sup>+</sup>] and retrieves the model from the MLFlow model repository during deployment.

## ■ URL phishing detection integration

The Angler project was one of the core projects acting as a *golden path* case study and the kickstarter during the design phase of the new platform (documented in the MATS stack[DSK]). The goal was to explore upgrading MLOps maturity from level 1 to level 4. The project is integrated into the new Schnitzel platform. It is automated end-to-end from training to deployment with automatic metrics collecting.

The training pipeline is implemented in *Airflow* (section A.3), storing metrics, statistics, and the trained model to *MLFlow* (section A.2). In MLFlow, the model, based on metrics and performance, is upgraded to `prod` stage. From there, the *MLFlow Deployer* (subsubsection 6.1.3) triggers the model deployment. The model is then automatically deployed via *Helm Charts* (subsubsection A.4.1) to the *Seldon* (section A.5) cluster using the TensorFlow Serving flavor[mlfa]. Seldon then collects the metrics from serving and stores them in Grafana<sup>4</sup>, an analytics visualization web application.

## ■ 6.2 Socialbakers

The second case study describes the Machine learning operations at Socialbakers<sup>5</sup>. Socialbakers is an AI-powered Czech company working in the field of social media marketing analytics. They offer performance indicators of social media outreach and benchmarks against competitors and industry standards.

<sup>4</sup><https://grafana.com>

<sup>5</sup><https://www.socialbakers.com>

### 6.2.1 Initial state of ML serving

Historically, there had been more offline inference workloads than online inference use cases. Thus the focus was on enabling and improving offline inference through tooling and infrastructure. Unfortunately, online inference projects were piggybacking off the present systems (intended for offline inference) instead of redesigning the solutions to fit both use cases, which introduced technical debt reaching over to the original systems.

There was a standard approach how to publish trained ML models. The idea was to generalize the API for using all models at Socialbakers by creating an Abstract base class[pyt] that all models would extend. The ML model is then published as a versioned Python package. Whenever the model changed, a new version needed to be published. The model's trained weights were commonly stored in Amazon S3<sup>6</sup>, a managed object store from AWS.

#### Emerg ed problems

This abstraction brought with it multiple problems that became apparent after the usage of this pattern increased. Because neither were the present systems integrated nor the process automated, the ML researchers depended on the engineers for the model release. The deployment was primarily done manually and, due to this, very tedious and error-prone. When an ML model was done training, the ML researcher needed to hand it over to the engineers. Afterward, the model would get released as a new version of the appropriate Python package. Lastly, all dependent systems would need to update the version dependency on this model and release/deploy the version using the latest model.

The above problem also led to multiple versions being used concurrently at the same time on different projects leading to inconsistencies for the result data. For example, one ML model could be served for online inference with the latest version, but since the dependency needed to be updated manually, some jobs using Spark<sup>7</sup>, a big data analytics engine, could still be using the old version of the model.

Further, this approach of enforcing codebase sharing across several projects and the design of extending the Abstract base class by each model led to issues with transitive dependencies. Transitive dependency issues arise when the dependencies of a project share common dependencies but different versions. Imagine example project A depends on lib-a and lib-b. lib-a depends on lib-c version v1.0 but lib-b depends on lib-c version v.2.0 where the core API changed from v1.0 to v2.0. Since the versions of lib-c as a transitive dependency are incompatible either the dependency system will fail the resolution or the application might fail during runtime.

---

<sup>6</sup><https://aws.amazon.com/s3>

<sup>7</sup><https://spark.apache.org>



### ■ 6.2.2 Motivation for change

Problems such as different model versions for online and offline inference have been overlooked because the reach of such issues was tolerable, and the technical debt was localized primarily to these use cases. However, the number of online inference workloads is increasing. Thus the previously described problems were identified as potentially becoming significant issues in sight of the current trends and determined to be resolved.

Similar to the Avast case study (subsection 6.1.2) the transition was made possible by higher focus and investment to innovation by the organization. As the issues were identified to surpass a reasonable threshold, a designated MLOps team has been established to deal with the accumulated technical debt and design a new future-proof solution.

Other than bringing structure in place of the common base class, another goal was to decouple the individual components in the online inference use case. The looser coupling[Fow01] of the workflow components would lead to establishing clear boundaries between individual steps. That would, in turn, streamline the automation/integration while lowering the complexity of each, thus removing the wall between ML researchers and engineers for the deployment problems previously described.

Even though there was a typical pattern of how to design ML models, although, with issues on its own as described above, there was no standardization across the different projects otherwise. A project would frequently spread across several repositories, i.e., one repository containing the model definition, a separate repository responsible for the training pipeline, etc. The Designated MLOps team was intended to consolidate this.

### ■ 6.2.3 Present MLOps initiative

The new MLOps team defined three goals to deal with the identified problems of the previous solution:

- single source of truth for inference workloads
- loose coupling of participating components
- minimize MLOps intervention in the release process

Single source of truth requirement originated from the problem of a model having multiple live versions being used concurrently. A unified way of storing and accessing the models in a standardized format in a central model repository solves that.

Loose coupling of the participating components, generally regarded as a good practice[Fow01], means components/modules are separated with a designed contract between them. It has two reasons behind it. The first is reducing technical debt, as well-defined boundaries and proper module separation support higher cohesion. Additionally, it improves the cooperation of the teams as the components have better-defined boundaries and can be improved internally independently.

Further, loose coupling enables easier A/B testing of components, a desired functionality of the platform, as the configuration level is finer. A/B testing is the practice of using multiple concurrent live versions while collecting metrics for the experiment's evaluation. The difference is that multiple live versions exist by design and are distinguished.

To minimize the MLOps intervention during the machine learning development and model deployment, the designed platform needs to be self-serviced for the ML researchers — easy to work with and simple to understand. That is supported by leveraging present systems that everyone is familiar with and automating tasks that previously needed manual intervention.

## Architecture

The new architecture is depicted in Figure 6.4 and implements the *Tech stack integration* (section 4.4). It integrates the internal build & compute platform *Maratonec* with *MLFlow* (section A.2) backed by Amazon S3 as a model repository.

*Maratonec* is the internal build & compute platform. It provides a CI/CD platform for building projects residing in GitLab<sup>8</sup>. Further, it allows running services as defined by Procfiles[pro] in the repositories. Finally, *Maratonec* provides UI for tracking the build process and setting up run configurations for the services (i.e., required CPU, memory, number of instances, environment variables, ...).

Managed MLFlow from Databricks<sup>9</sup> is used as a Model repository. The reason for going with the managed service instead of deploying it locally to *Maratonec* is the already present contract for Databricks<sup>10</sup>, a managed Spark platform, and also the company that created MLFlow.

The crux of the platform lies in the `mlflow-serve` repository. This project integrates the compute platform *Maratonec* with the MLFlow model repository to deploy models for online inference. Each model is deployed as a run configuration of the `mlflow-serve` project. It contains a generic way to serve the models by configuring the necessary parameters via environmental variables.

There are currently two implemented modes of deployment for online inference — an HTTP service and a message queue service. The HTTP service exposes the model via an HTTP API. This implementation leverages the MLFlow Models local deployment[mlfb] option. Internally the model is exposed using Gunicorn<sup>11</sup> and Flask[fla].

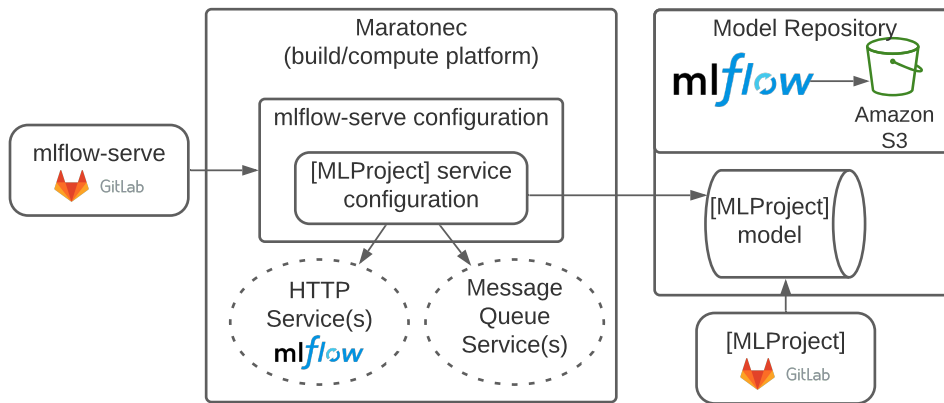
The message queue service processes the requests asynchronously. The deployed model is wrapped in a service that feeds it messages from a source queue and pushes results into possibly multiple result queues. The messaging queues are not part of the model deployment. Thus, except for the specific interaction when pulling messages, the queues and models can be scaled

<sup>8</sup><https://about.gitlab.com>

<sup>9</sup><https://databricks.com/product/managed-mlflow>

<sup>10</sup><https://databricks.com/>

<sup>11</sup><https://gunicorn.org>



**Figure 6.4:** Diagram of the MLOps serving platform at Socialbakers.

Maratonec is an internal build/compute platform, `mlflow-serve` is a GitLab repository integrating MLFlow serving and Maratonec, `[MLProject]` symbolizes projects producing ML models, the Model repository is MLFlow backed by Amazon S3 object storage.

independently. Another benefit of this approach is that the service will never get flooded with requests and won't throw them away as it pulls messages from the queue at its own pace.

Example usage of the Message queue service model deployment is depicted in Figure 6.5 showing a possible Sentiment analysis application. The problem it solves is pipelining multiple deployed models. First, it needs to detect the language in the message. Based on that, it uses the correct queue for the model trained for the detected language. The application is implemented using Akka Streams<sup>12</sup> as the library naturally supports the event-based use case of IO via message queues. The messaging queues currently used are RabbitMQ<sup>13</sup>.

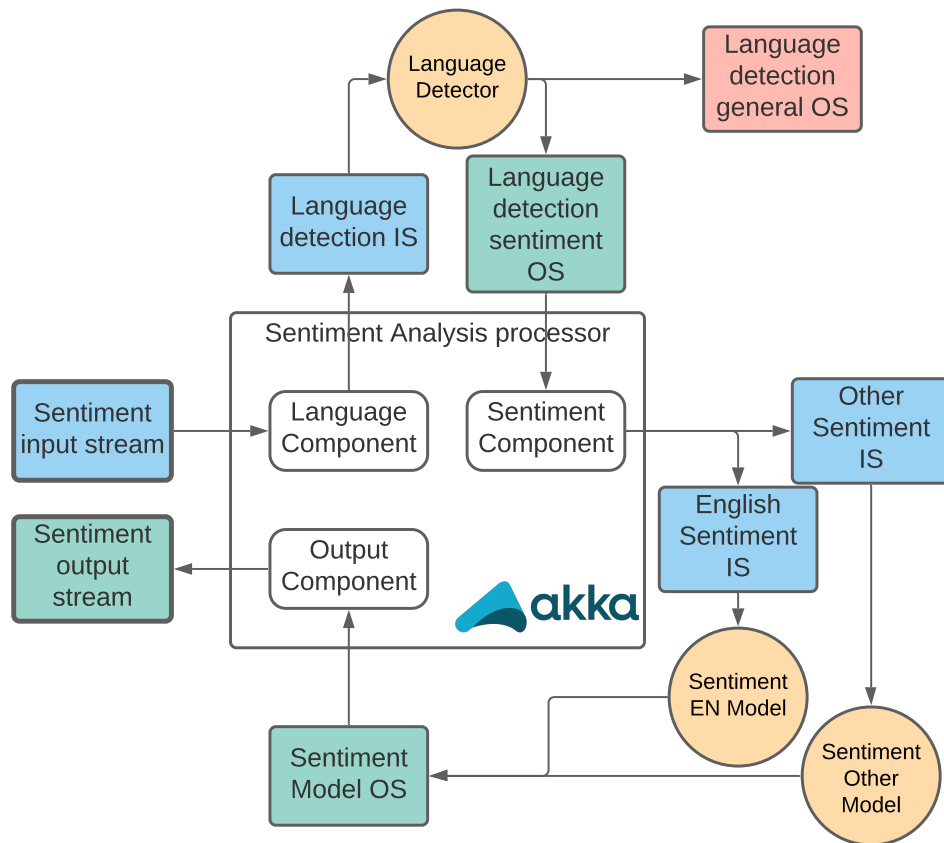
### ■ Limitation of the new solution

There have been identified two issues with the current solution — lack of access restrictions to the configurations and limited use case scale due to the configurations all being at the same place. Both can be solved by forking the GitLab repository`[gita]` as an immediate fix until a permanent solution is implemented into the system. However, forking the repository would not be the best long-term solution as upgrading the `mlflow-serve` might break dependent projects without a central way of testing. Cumbersome upgrades were one of the initially identified problems and a situation best be avoided.

The first problem is missing access restriction as Maratonec supports permission management per project and not for each configuration. The `mlflow-serve` Maratonec configuration contains parameters of all the deployed models. It does not need to be a malicious intent but a simple mistake

<sup>12</sup><https://doc.akka.io/docs/akka/current/stream/index.html>

<sup>13</sup><https://www.rabbitmq.com>



**Figure 6.5:** Diagram of the sentiment pipeline demonstrating an example usage of the Message queue service model deployment.

Blue – input queues, green – output queues, red – other output queue. Orange – deployed models. At the center is the sentiment analysis application connecting the queues implemented with Akka streams.

Source: Socialbakers

Original: Figure B.5

that takes down a model from being served (perhaps replacing the language model with the sentiment model breaking the pipelines depending on it). Although this violates the principle of least privilege[pol], the localization of the issue and safety boundaries in place, such as canary deployment and monitoring, should quickly detect this. Thus it isn't considered a significant threat.

The second problem is limited scaling by having all the configurations in the same project. The run configurations are defined via Maratonec UI, which means the more models are being served, the more cluttered it becomes (amplified by multiple deployment environments for each model, i.e., `dev`, `prod`, ...) — eventually crossing a threshold of usability. This issue can also be resolved by forking the `mlflow-serve` project. Each model or team would then have a fork with all the Maratonec configurations for relevant models.

## ■ 6.3 Survey

As the last case study, we conducted a survey as quantitative analysis. The survey was shared publicly on LinkedIn & other social media of the author and Machine Learning Meetups<sup>14</sup>.

### ■ 6.3.1 Survey sections

The survey contained five multiple-choice questions to inspect the state of MLOps in the industry and two optional questions identifying the respondent — company/project name, contact to share the results.

*Company size*, number of employees bucketed into 5 categories:

- 1-4
- 5-9
- 10-49
- 50-99
- 100+

*Your role in the company* determining the occupation of the respondent:

- ML researcher/data scientist
- MLOps/data engineer
- Neither

*How are the models used* determining the deployment scheme used as described in chapter 4 with the first option describing offline inference workloads (multiple answers possible) and the rest to the *Deployment scheme definitions* (chapter 4):

- Models are used in analytical workflows (the model isn't deployed for serving)
- Simple web service wrapper around the model (eg. Flask wrapping pickled scikit model)
- Export to an intermediate format with evaluation engine (eg. ONNX, PMML)
- E2E platform (eg. TFX, cloud solution like AWS SageMaker)
- Custom platform from open-source components (eg. Airflow + MLFlow)
- Other (with optional text field for more information)

---

<sup>14</sup><https://www.brno.ai/Akce/MLMU-14-MLOps-Building-feature-stores>

To what extent is the machine learning deployment in your company supported by the MLOps team to describe the cooperation model between ML researchers and MLOps (if present):

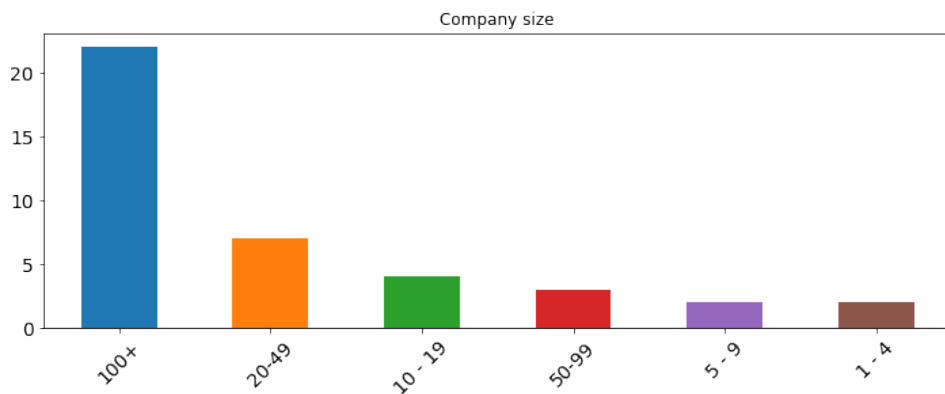
- Teams do everything on their own (No MLOps team)
- Dedicated MLOps team
- Self service platform provided by the MLOps team

Volume of machine learning models bucketed into 4 categories:

- None (0)
- Couple (1-4)
- Multiple (5-14)
- A lot (15+)

### ■ 6.3.2 Survey results

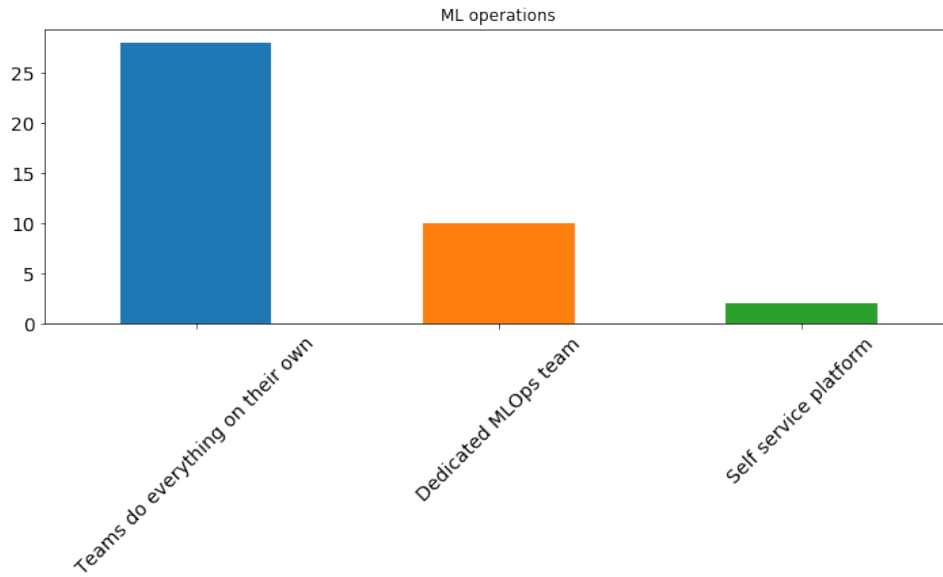
The survey collected 45 responses in total. Out of those, 40 were valid. Figure 6.6 displays the histogram of the company size of the respondents to show the background from where these answers were collected. The respondents were predominantly from larger companies.



**Figure 6.6:** Histogram of the number of employees

In Figure 6.7 we can see that most companies have the responsibility for deploying ML models in the hands of the individual teams. It shows that MLOps is still in the very early stages of adoption in the industry. This statistic is further separated in contingency Table 6.1 by company size, showing that dedicating a team for machine learning operations is not more frequent for a specific size category. This might be contrary to the expectation of larger companies having more resources and solving problems centrally.

In Table 6.2 we can see that dominant is the *Tech stack integration* (section 4.4) closely followed by *Native serving* (section 4.1). It seems that



**Figure 6.7:** Histogram of ML operations team

Company size	No MLOps team	MLOps team	Self service platform
1 - 4	2	0	0
5 - 9	1	1	0
10 - 19	3	1	0
20-49	4	2	1
50-99	2	1	0
100+	16	5	1
sum	28 (70%)	10 (25%)	2 (5%)

**Table 6.1:** Contingency table showing the relationship between company size and MLOps adoption. We find no significant difference between different company size categories.

companies choose the Native serving due to simplicity or lack of need for more advanced features. Tech stack integration is expectably popular as it provides an option to leverage present systems and thus alleviates some of the tasks necessary for building such a platform ground up and thus supports migration to/exploration of machine learning.

Surprisingly the *End-to-end* (section 4.3) approach is not very favored across companies. One reason behind this might relate to the previous argument for Tech stack integration, that it is desirable to avoid duplicating functionality by deploying multiple systems with the same responsibility (e.g., having another workflow scheduler integrated into the E2E platform in addition to the one used across the company). Another issue with the End-to-end solution might be the tight coupling of components (less flexibility) and vendor lock-in. Vendor lock-in prevents switching providers without substantial cost, which might discourage some people from adopting a specific solution.

Deployment scheme	count
Tech stack integration	22
Native serving	19
Offline inference	16
Software stack integration	7
E2E platform	7
Other	7

**Table 6.2:** Histogram of model deployment schemes. We see the Tech stack integration and Native serving are the two most popular approaches.

The Table 6.3 shows the relationship between the number of production models and the selected approach to serving them. There are 45% of Native serving use cases for small workloads (less than four models). However, for Tech stack integration and End-to-end approaches, almost 70% and 85% of the responses, respectively, indicated workloads of five or more models. This supports the natural expectation that full-fledged platforms are intended for higher usage to be cost-efficient concerning the resources initially invested and further required for maintenance. On the other hand, this can also be caused by an opposite phenomenon. A more accessible and feature-rich platform enables easier deployment, leading to more models being developed and deployed.

Number of models	Couple (1-4)	Multiple (5-14)	A lot (15+)
Tech stack integration	7	7	8
Native serving	9	5	5
Software stack integration	3	1	3
E2E platform	1	3	3
other	0	3	4

**Table 6.3:** Table showing the relationship between model deployment scheme and number of production models.


Lastly, in Table 6.4 we have the contingency table of deployment scheme and designated MLOps team. One exciting aspect shown in this table is that advanced platforms (Tech stack integration and E2E) are adopted in companies even without the MLOps team. This might indicate either natural MLOps adoption or misunderstanding of the survey assignment.



How are the models used?	No MLOps	MLOps	Self service
Tech stack integration	16	5	1
Native serving	14	3	2
E2E platform	6	1	0
Software stack integration	6	1	0
other	2	5	0

**Table 6.4:** Relationship between the MLOps adoption and used model deployment approach. Notable is the usage of E2E platform as a complex scheme without a designated MLOps team to maintain it.





## Chapter 7

### Conclusion

This thesis explored the emerging field of Machine learning operations aiming to streamline machine learning development by incorporating software engineering best practices into the ML development process. We specifically focused on deploying trained machine learning models for online inference use cases.

First, we presented four criteria — *scale* (section 3.1), *simplicity* (section 3.2), *completeness* (section 3.3), and *extensibility* (section 3.4) — to provide a unified way to describe and evaluate the deployment schemes.

Then, we identified five schemes to deploy ML models. *Native serving* (section 4.1) as the baseline approach to exposing the model as a service. *Software stack integration* (section 4.2) embedding the model evaluation into the application while possibly using an intermediate format with an accompanying evaluation engine. Then we described three more extensive and mature platforms. The *End-to-end* (section 4.3) leverages solutions providing most of the functionality out-of-the-box, and the *Tech stack integration* (section 4.4) comprising designing and building the platform out of existing functional components. The last approach, *In-house custom solution* (section 4.5), supports use cases where none of the previous solutions fit, and the available resources allow to design the platform to the expected users.

Next, we proposed a framework for selecting the best approach out of the identified deployment schemes depending on the intended use case. Two views were described. The first fit for initial MLOps stages optimized for fast results (section 5.1). The second (section 5.2), for situations where a more mature platform is desired.

Lastly, we presented two case studies describing the ML operations at Avast and Socialbakers. The case studies provided the architecture and implementation of the MLOps approaches used in the companies and the argumentation behind the decisions that led to them. We can see that simple solutions were initially used until a more mature approach was required due to the business development.

As part of the case studies, we conducted a survey (section 6.3) where we observed the *Native serving* (section 4.1) and *Tech stack integration* (section 4.4) to be the most popular approaches of the respondents. We noticed that there is no dedicated ML operations team in 70% of the companies participating in

the survey. The two in-depth case studies show that establishing an MLOps team proves beneficial. We recommend companies to explore and invest in MLOps as currently in 50% of organizations it takes up to 90 days to deploy a single ML model to production[ent19]; thus, unification and automation might provide significant savings in the ML development.

Chapter 6 further shows that end-to-end approaches, which appear to be excellent solutions at the first look, have limitations that are problematic for the industry setting. Therefore, these kinds of solutions might be a better fit for specific use cases rather than picking them in general settings as described in chapter 5.

Despite no existing historical data for machine learning operations, we can take inspiration from the DevOps movement's popularity and industry trends and expect MLOps to take a similar path. As machine learning is becoming even more prevalent in present software systems, MLOps adoption will soar in the following years, and more companies will adopt a similar approach.

## Appendix A

### Tech glossary

The following sections contain brief descriptions of the systems used in the case studies.

#### A.1 TeamCity

TeamCity<sup>1</sup> is a build management and CI/CD (continuous integration and continuous delivery) tool developed by JetBrains. It supports management via web app or as a code so that the build definitions (and supporting tooling, e.g., build step uploading artifact to a repository) can be stored in a version control system. For code definitions, it uses the Kotlin<sup>2</sup> programming language (also developed by JetBrains).

TeamCity's web UI can also be used to check the status of builds/pipelines. It also provides integration with many systems around build and source code management for an easier development process. For example, the build status of a branch in a GitHub repository can be displayed in the PR (pull request). If the build is failing, it can prevent the PR from being merged into the master branch.

Although TeamCity integrates well with build systems (e.g., Maven, Gradle, ...), it is not its only use. It is not tightly coupled with the build intention and can be used for general automation workloads, as it can run arbitrary code in a containerized environment, as is, for example, the case at Avast (subsubsection 6.1.3).

#### A.2 MLFlow

MLFlow<sup>3</sup> is an open-source machine learning lifecycle platform developed in Python to streamline ML development. It comprises four components that each provide a different functionality and can be used independently:

**Tracking** — model development tracking

---

<sup>1</sup><https://www.jetbrains.com/teamcity>

<sup>2</sup><https://kotlinlang.org>

<sup>3</sup><https://mlflow.org>

**Projects** — code packaging format using Conda and Docker

**Models** — model packaging format and tools to deploy them

**Model Registry** — centralized model store

MLFlow fits at the center of machine learning development. The ML researcher interacts with the platform through its REST API (with clients implemented for Python, R, and Java) or a CLI.

Using the *MLFlow Projects* and *MLFlow Tracking* allows the user to run machine learning experiments and model training flows in a standardized way. Declaratively defining the environment to run the experiments and logging the metrics, graphs, and other (meta)data about the model and experiment. Further, it provides a web UI to display the collected data.

Then the *MLFlow Models* and *MLFlow Model Registry* can be used to store and release the trained models. As with the other components, the functionality is integrated into the web UI where it enables the user to manage the state transitions of the models' versions. Further, it allows the user to deploy the model for serving on a local machine or few integrated environments (Amazon SageMaker, AzureML, Spark UDF). It supports a lot of the popular ML libraries[mlfa].

### ■ A.3 Airflow

Apache Airflow<sup>4</sup> is an open-source workflow management platform. It contains time-based scheduling of running the defined workflows or based on external triggers. The basic Airflow project provides plethora of integrated services for both on-premise services (e.g., Hadoop<sup>5</sup>, Redis<sup>6</sup>, ...) and managed/cloud services (e.g., Slack<sup>7</sup>, AWS, ...).

Workflows are represented as *DAGs* (Directed Acyclic Graph). Each DAG consists of *Tasks* (nodes in the graph) that should be run. The tasks are connected by edges symbolizing the precedence of the tasks. Workflows are implemented as standard Python files, with the tasks being instances of *Operators*. Two base types of operators exist — one for flow control structures (e.g., `BranchPythonOperator`) modifying the DAG, second for running tasks (e.g., `BashOperator`).

In the Airflow's architecture diagram (Figure A.1) there are two significant components. The first is the Web server with Executor (depicted separately but runs within the same context) that orchestrates the DAG's evaluation. The web server also provides the UI for the user to see the evaluation of DAGs, their state, and schedules. The second is the Worker component — processes that run individual tasks. Workers can be running on a different machine, which is leveraged, for example, at *Avast* (subsubsection 6.1.3).

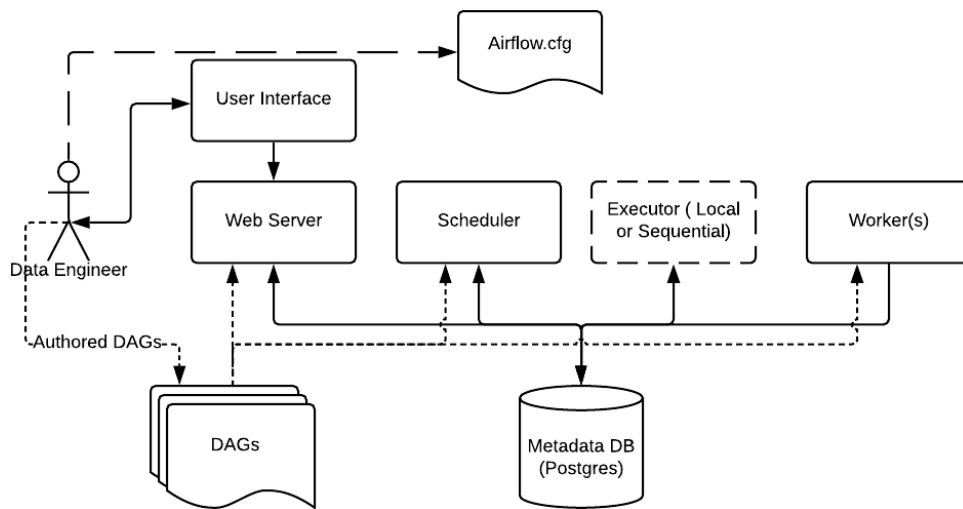
---

<sup>4</sup><https://airflow.apache.org>

<sup>5</sup><https://hadoop.apache.org>

<sup>6</sup><https://redis.io>

<sup>7</sup><https://slack.com>



**Figure A.1:** Apache Airflow platform architecture.

Source: <https://airflow.apache.org/docs/apache-airflow/stable/concepts.html>

## ■ A.4 Kubernetes

Kubernetes<sup>8</sup>, commonly abbreviated to *k8s*, is an open-source container orchestration platform. A Kubernetes cluster (Figure A.2) consists of a *control plane* and worker *nodes*. The control plane manages the global state of the cluster, while worker nodes host *pods*, the basic unit of deployment, describing the containers and other resources. Worker nodes run a *kubelet* service that manages the running containers, the container runtime, and a *kube-proxy* that allows the networking of the containers.

Kubernetes uses the concept of *Kubernetes objects* through which the user configures the platform. A Kubernetes object is a record of intent that describes the desired state. Once the object is defined, Kubernetes manages the cluster state to ensure the object exists.

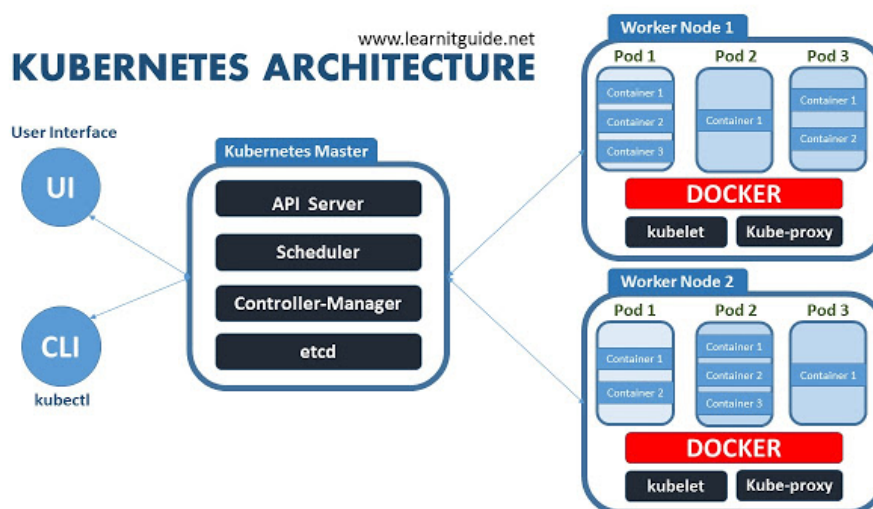
The user interacts with the platform via an API or `kubectl` CLI.

### ■ A.4.1 Helm

Helm<sup>9</sup> is a package manager for Kubernetes. It uses a packaging format called *charts* — a collection of files defining the Kubernetes objects, chart dependencies, etc. Helm charts allow the user to define, package, and version applications for Kubernetes instead of sharing the objects as files. Further, charts allow defining templates to enable customizing the application deployment.

<sup>8</sup><https://kubernetes.io>

<sup>9</sup><https://helm.sh>



**Figure A.2:** Kubernetes components diagram.

Source: What is Kubernetes[lea]

## A.5 Seldon

Seldon<sup>10</sup> is a machine learning serving platform exposing models as REST or gRPC services. It is built on *Kubernetes* (section A.4) which allows it to run both on-premise and in the cloud. Seldon offers three components (Figure A.3) — Core, Alibi, and Deploy. Seldon Core is an open-source platform for serving ML models with features such as model routing, inference graphs, and support for experiments. Seldon Deploy is a UI component allowing testing, monitoring, and deploying models. Seldon Alibi is an open-source Python library for model inspection and interpretation.

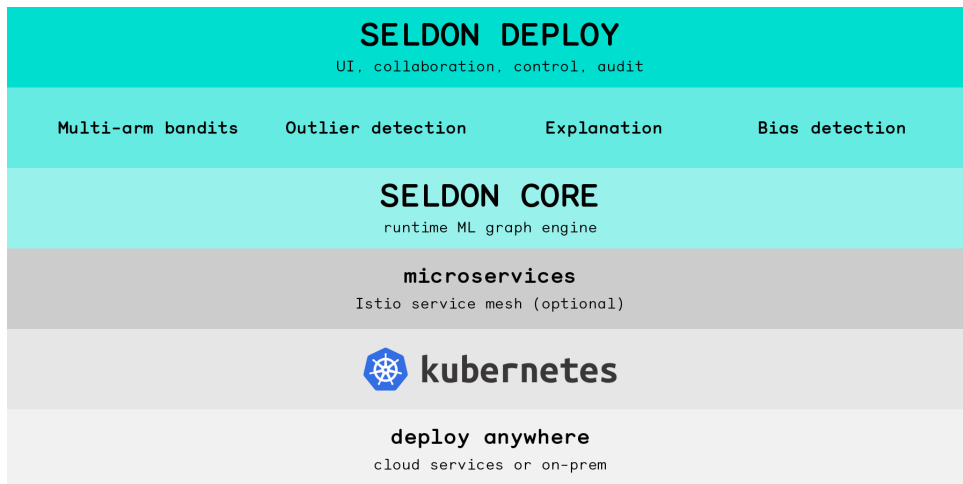
## A.6 GitOps

GitOps[gitb], a term coined in 2017, is the idea of having a git repository, although the same can be achieved with other versioning systems, containing declarative definitions of the deployed systems & infrastructure. A required component for this concept is having the automation to deploy this definition to the environment. It is the practice of *Infrastructure as Code* extended with the software engineering automation and tooling.

GitOps practices provide streamlined deployments. The requirements for GitOps, having the environment defined declaratively, lead to easier environment replication and self-documenting description, as everything available in the deployment needs to be specified, strongly pushing away from manual changes to the deployed environments.

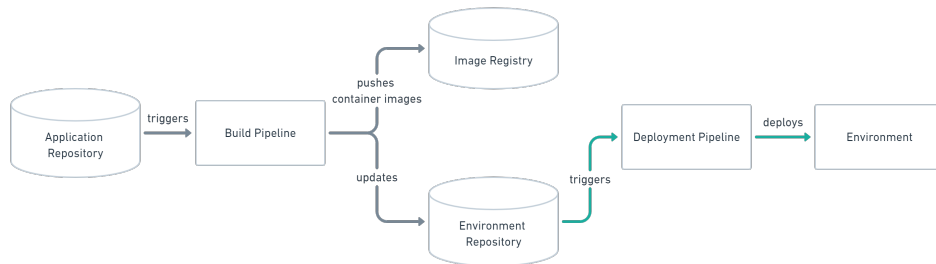
<sup>10</sup><https://www.seldon.io>





**Figure A.3:** Seldon Core stack

Source: <https://www.seldon.io/tech/products/core>



**Figure A.4:** GitOps workflow diagram for push-based deployment

Source: <https://www.gitops.tech>

In Figure A.4 we can see an example development workflow. First, the developer pushes changes to a repository. That triggers the build pipeline, which releases a new version of the application and makes the appropriate changes in the environment repository. That, in turn, triggers its deployment pipeline leading to the update environment.

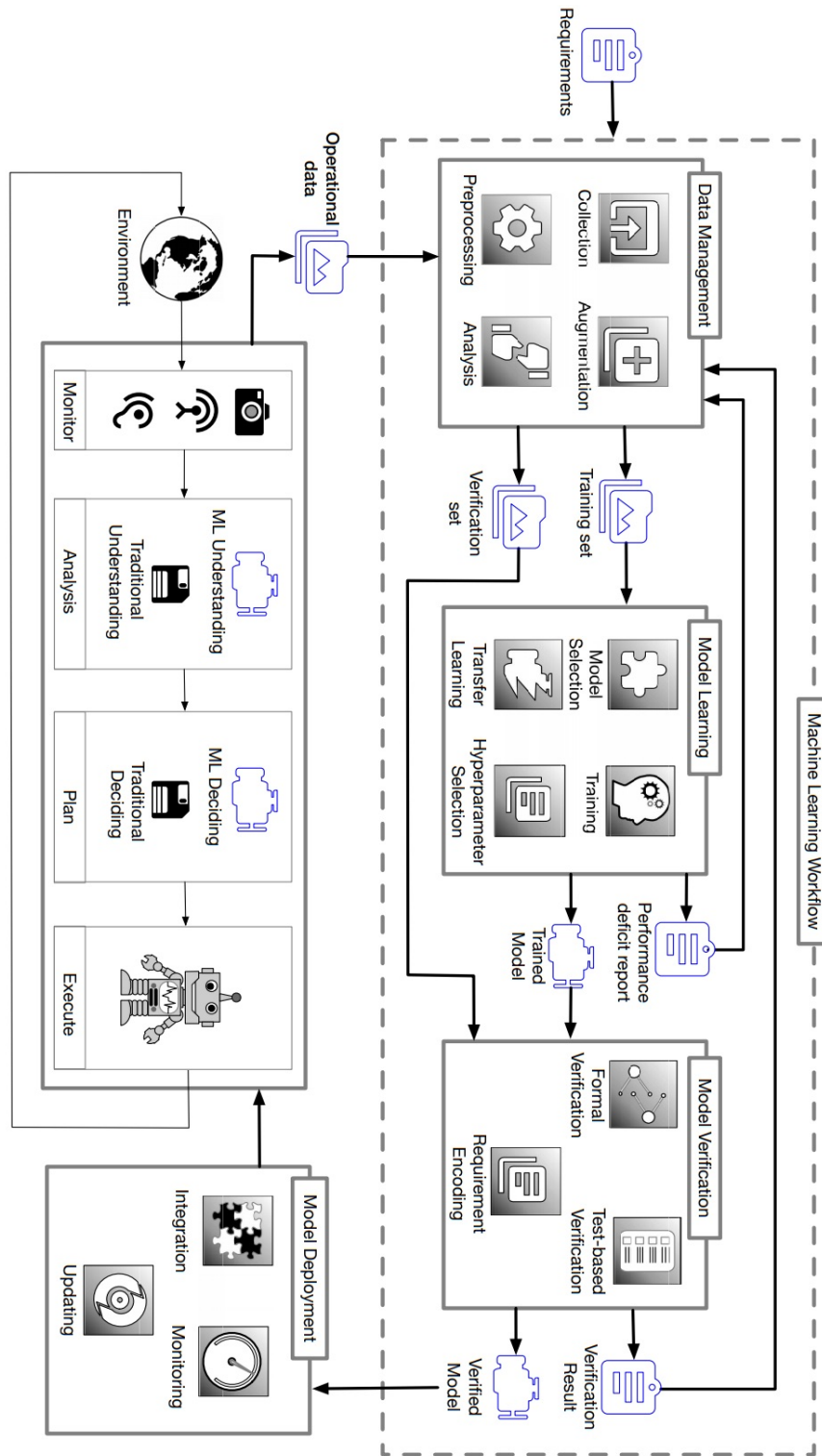




## **Appendix B**

### **Large figures and originals**

This chapter contains the original figures that were redrawn for print clarity and illegible figures.

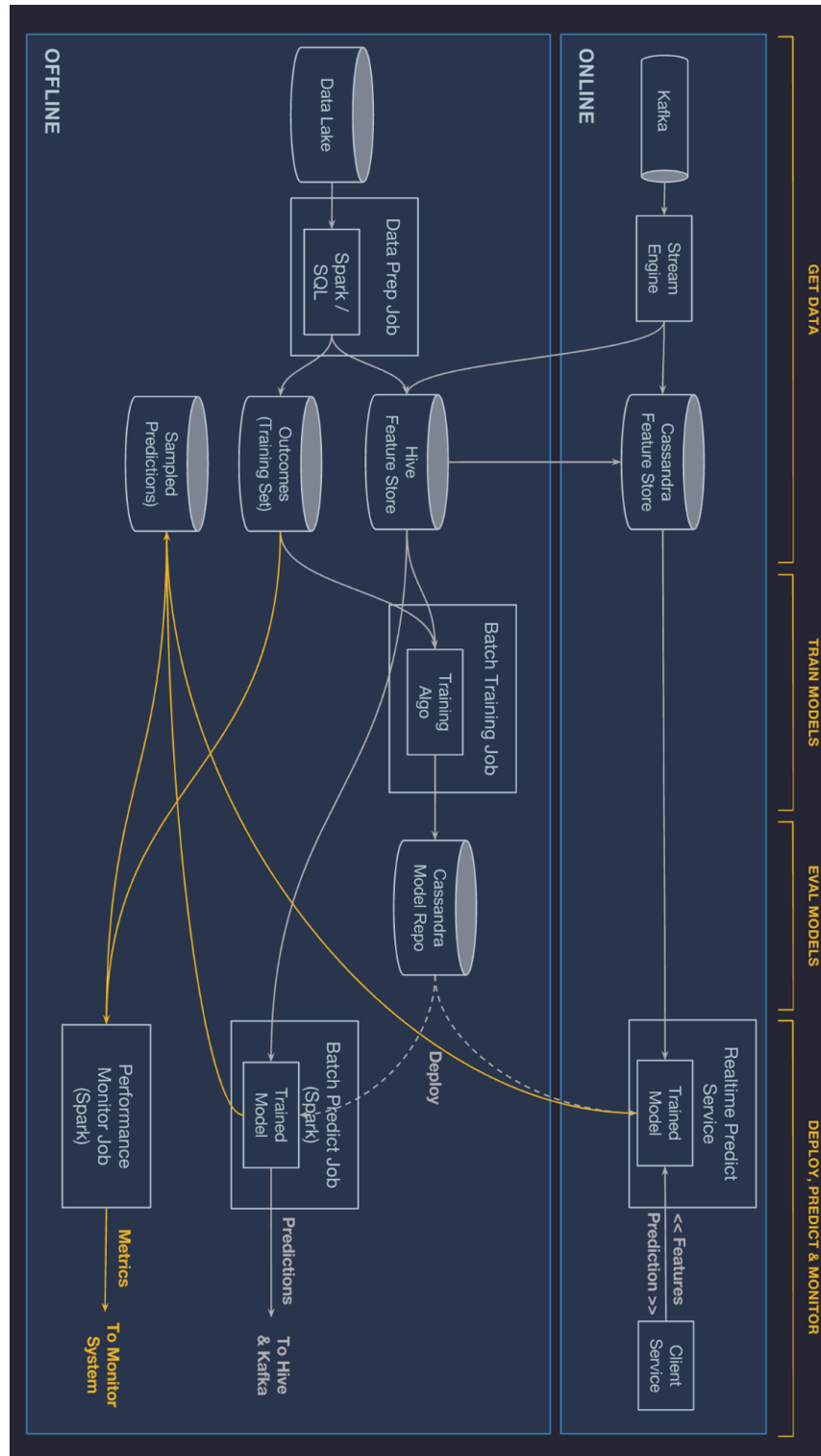


**Figure B.1:** Machine Learning Lifecycle Diagram

Source: Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges [ACP19]



**Figure B.2:** Functionality offered by the Amazon SageMaker  
 Source: <https://aws.amazon.com/sagemaker>



**Figure B.3:** Architecture of Uber's Machine learning platform Michelangelo. Source: Meet Michelangelo: Uber's Machine Learning Platform [HB]

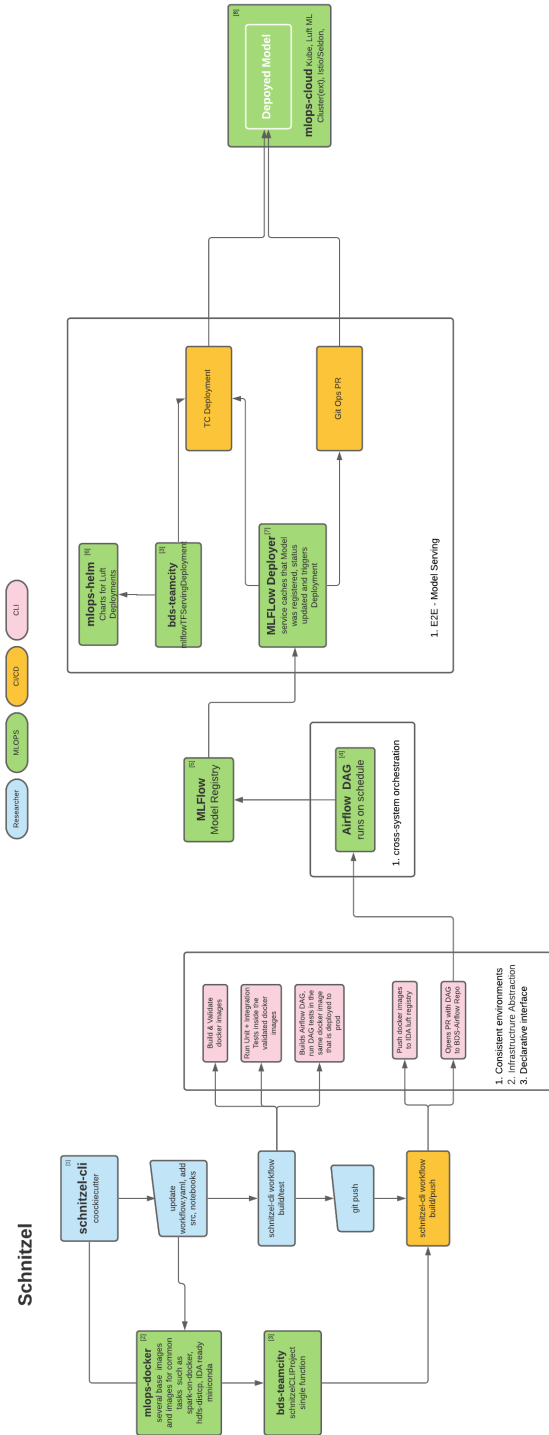
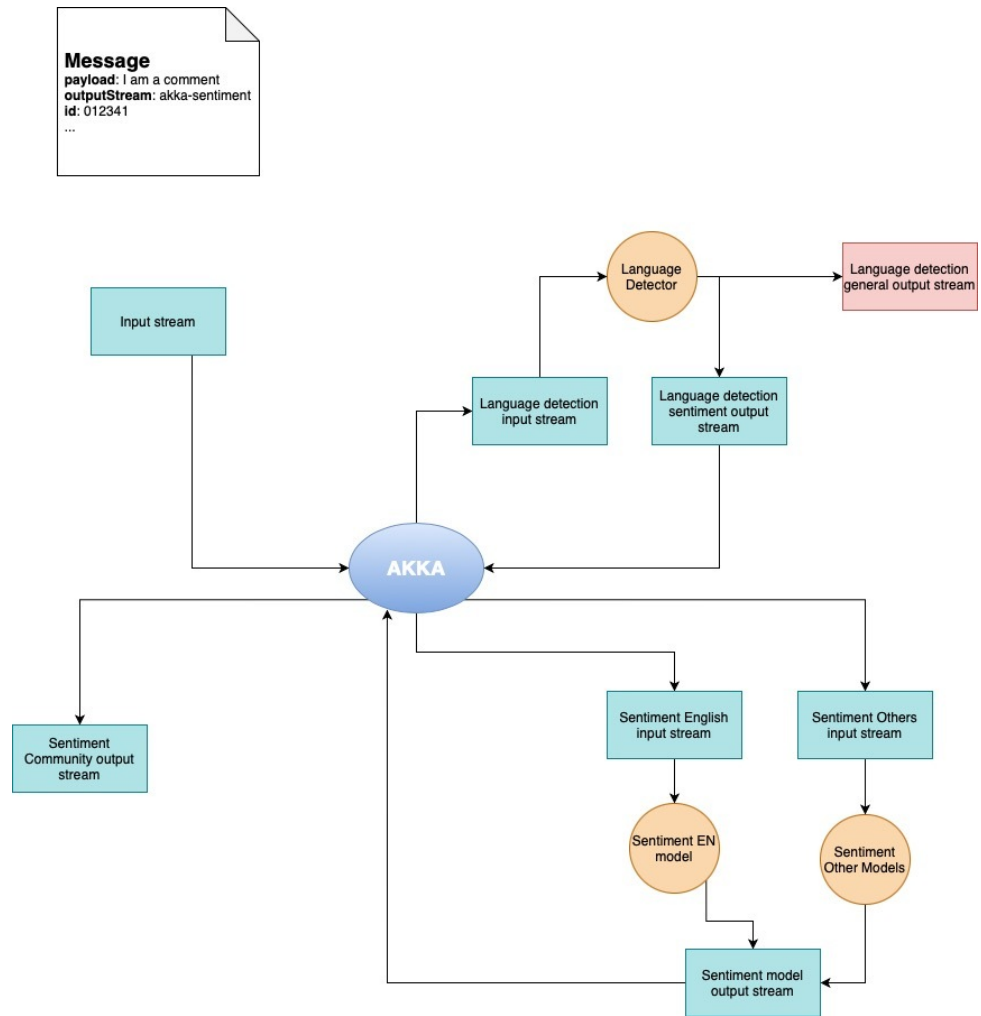


Figure B.4: Avast Schnitzel original diagram. Author: João Da Silva, Avast



**Figure B.5:** Sociabakers sentiment analysis pipeline original diagram. Author: Alex Hagerf, Socialbakers



## Appendix C

### Attachment

```
/
├── thesis ..... the source files of this thesis in LATEX
│   ├── img ..... figures
│   ├── tex ..... content files
│   └── zav_prace.pdf ..... thesis assignment
├── mlops_survey.tsv ..... cleaned survey responses
└── thesis.pdf ..... this thesis
```



## Appendix D

### Bibliography

- [ACP19] Rob Ashmore, Radu Calinescu, and Colin Paterson. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. 2019.
- [aim20] Worldwide spending on artificial intelligence is expected to double in four years, reaching \$110 billion in 2024, according to new idc spending guide. <https://www.idc.com/getdoc.jsp?containerId=prUS46794720>, Aug 2020. [Online; accessed May-2021].
- [ALT<sup>+</sup>14] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, page 173–182, 2014.
- [BLZ<sup>+</sup>] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. <https://github.com/onnx/onnx>. [Online; accessed March-2021].
- [DPL15] Andrej Dyck, Ralf Penners, and Horst Lichter. Towards definitions for release engineering and devops. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 3–3, 2015.
- [DSK] João Da Silva and Yury Kasimov. Mats stack (mlflow, airflow, tensorflow, spark) for cross-system orchestration of machine learning pipelines. [https://databricks.com/session\\_eu20/mats-stack-mlflow-airflow-tensorflow-spark-for-cross-system-orchestration-of-machine-learning-pipelines](https://databricks.com/session_eu20/mats-stack-mlflow-airflow-tensorflow-spark-for-cross-system-orchestration-of-machine-learning-pipelines).
- [ent19] 2020 state of enterprise machine learning: Cost reduction is the top priority. <https://jaxenter.com/2020-state-of-enterprise-machine-learning-165649.html>, Dec 2019. [Online; accessed May-2021].
- [fla] Flask framework. <https://palletsprojects.com/p/flask>. [Online; accessed April-2021].

- [Fow01] M. Fowler. Reducing coupling. *IEEE Software*, 18(4):102–104, 2001.
- [gita] Gitlab project forking workflow. [https://docs.gitlab.com/ee/user/project/repository/forking\\_workflow.html](https://docs.gitlab.com/ee/user/project/repository/forking_workflow.html). [Online; accessed May-2021].
- [gitb] Gitops. <https://www.gitops.tech>. [Online; accessed May-2021].
- [gra] Organizing gradle projects. [https://docs.gradle.org/current/userguide/organizing\\_gradle\\_projects.html](https://docs.gradle.org/current/userguide/organizing_gradle_projects.html). [Online; accessed April-2021].
- [Gro] Data Mining Group — PFA Working Group. Pfa: Portable format for analytics. <http://dmg.org/pfa/index.html>. [Online; accessed March-2021].
- [HB] Jeremy Hermann and Mike Del Balso. Meet michelangelo: Uber’s machine learning platform. <https://eng.uber.com/michelangelo-machine-learning-platform>.
- [Hela] Ian (a.k.a. Christian) Hellström. Machine learning platforms in 2021. <https://databaseline.tech/ml-platforms-in-2021>.
- [Helb] Ian (a.k.a. Christian) Hellström. A tour of end-to-end machine learning platforms. <https://www.kdnuggets.com/2020/07/tour-end-to-end-machine-learning-platforms.html>.
- [lea] learnitguide.net — what is kubernetes. <https://www.learnitguide.net/2018/08/what-is-kubernetes-learn-kubernetes.html>. [Online; accessed May-2021].
- [mav] Maven standard directory layout. <http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html>. [Online; accessed April-2021].
- [MKF<sup>+</sup>17] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. Tfx: A tensorflow-based production-scale machine learning platform. In *KDD 2017*, 2017.
- [mlfa] Mlflow models supported libraries. [https://www.mlflow.org/docs/latest/python\\_api/mlflow.models.html#module-mlflow.models](https://www.mlflow.org/docs/latest/python_api/mlflow.models.html#module-mlflow.models). [Online; accessed April-2021].

- [mlfb] Mlflow models serve. <https://www.mlflow.org/docs/latest/models.html#local-model-deployment>. [Online; accessed April-2021].
- [mlo] Machine learning operations maturity model. <https://docs.microsoft.com/en-us/azure/architecture/example-scenario/mlops/mlops-maturity-model>. [Online; accessed April-2021].
- [MVM10] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Apache Maven*. Alpha Press, 2010.
- [onn] Onnx support for scikit-learn models. <http://onnx.ai/sklearn-onnx/supported.html>. [Online; accessed March-2021].
- [pmm] Pmml: Predictive model markup language. [https://en.wikipedia.org/wiki/Predictive\\_Model\\_Markup\\_Language](https://en.wikipedia.org/wiki/Predictive_Model_Markup_Language). [Online; accessed March-2021].
- [pol] Principle of least privilege. [https://en.wikipedia.org/wiki/Principle\\_of\\_least\\_privilege](https://en.wikipedia.org/wiki/Principle_of_least_privilege). [Online; accessed May-2021].
- [pro] The procfile heroku. <https://devcenter.heroku.com/articles/procfile>. [Online; accessed May-2021].
- [PVG<sup>+</sup>11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [pyt] Pep 3119 – introducing abstract base classes. <https://www.python.org/dev/peps/pep-3119/>. [Online; accessed May-2021].
- [Ros95] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.
- [sag17] Introducing amazon sagemaker. <https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-amazon-sagemaker>, 2017.
- [sbt] sbt directory structure. <https://www.scala-sbt.org/1.x/docs/Directories.html>. [Online; accessed April-2021].
- [SHG<sup>+</sup>15] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’15, page 2503–2511, Cambridge, MA, USA, 2015. MIT Press.

- [Sta21] Radek Starosta. Phishing detection using natural language processing. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2021.
- [wsg] Pep 3333 – python web server gateway interface v1.0.1. <https://www.python.org/dev/peps/pep-3333>. [Online; accessed May-2021].