Towards Maintainable and Explainable AI Systems with Dataflow

Andrei Paleyes
Churchill College

This dissertation is submitted on October 2023 for the degree of Doctor of Philosophy
Abstract

Machine learning is enjoying rapid growth both as a thriving academic discipline and as a technology that has the potential to transform many aspects of our everyday lives. We have already witnessed breakthroughs in speech generation, drug discovery, recommendation algorithms, and more, all achieved with the help of machine learning. It is vital to realise that any practical application of machine learning is not limited to just creating an accurate model based on a sanitised dataset. Such real-life applications are complex software systems, in which the model is only one, albeit important, component. A significant effort is also spent on creating data collection and cleaning pipelines, quality assurance, model updating workflows, monitoring and operational maintenance of these systems. The experience of numerous practitioners shows that the translation of a well-performing machine learning model to a well-performing machine learning system is not easy. This thesis embarks on a quest to understand the pain points of this translation process and explore software architecture paradigms well suited for the needs of modern data-driven systems.

We begin by surveying existing reports on ML deployment and the difficulties they describe. The identified issues and concerns are matched against a typical ML deployment workflow, and we show that there is no single bottleneck, and the entire deployment pipeline is riddled with challenges. We argue that a lot of these challenges are caused by existing software infrastructure and that more data-oriented approaches to software architecture are needed to tackle them. This observation leads us to the second contribution of this thesis, in which we examine data-oriented architecture (DOA) as a promising software architecture paradigm that machine learning systems can benefit from. We focus on measuring the level of adoption of DOA in practical deployments of machine learning and show that even though the paradigm itself is relatively unknown, its principles widely permeate the modern engineering of ML systems. Specifically, we identify dataflow architecture as one of the patterns that realise all DOA principles.

We proceed to evaluate the benefits of the dataflow for the deployment of machine learning. The evaluation is presented in two parts. In the first part, we compare the process of deploying an ML model within the functionally equivalent codebases of applications implemented with dataflow and service-oriented approaches, the latter being used as a baseline. We identify some benefits of dataflow, such as higher discoverability and simpler data collection in the
system. We also identify the limitations of the paradigm. We then present Seldon Core v2, an open-source model inference platform we designed following the dataflow architecture. We present a detailed discussion on how DOA principles can be implemented in practice, discuss the data observability features of the platform, and quantify the performance trade-offs involved.

The last contribution of the thesis points out another benefit of dataflow architecture for software development: a strong relationship between dataflow software and graphical causal models. We identify a connection between dataflow graphs and causal graphs and argue that this relationship allows a straightforward application of causal inference to dataflow software. We use fault localisation as a concrete example of this idea and showcase it in a variety of dataflow systems and scenarios.

The thesis closes with a discussion on research avenues that can further develop the community’s understanding and adoption of Data-Oriented Architectures and dataflow for machine learning systems.
Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60,000 words.

Andrei Paleyes
April 2024
Acknowledgements

Over the entire period of postgraduate studies I was asked the same question many a time: “Do you like being a student again? Any regrets about going for it and breaking your career in industry?” To which I always honestly replied that these are the best years of my professional life. So here is to many people without whom that would not be impossible.

First of all I would like to thank my supervisor, Neil Lawrence. His hands-off-yet-caring style of supervision allowed me to understand my own interests and passions, and to stick to them when things did not work out. His exceptional ability to connect people provided me with so many open doors and brilliant opportunities I frankly lost count of them. Choice of a supervisor is probably the most important one a PhD student can make, and I am absolutely sure I made the right one. Hopefully our relationship will continue to grow for the years to come.

I am grateful to my colleagues at Amazon Research Cambridge for all the inspirational examples they were and still are. I would probably not even think about doing a PhD if not for that amazing group of people. Special thanks to Javier Gonzalez for acting as my informal de-facto supervisor while I made my first cautious academic steps, teaching me all I know about Bayesian optimisation, and being an unshakably positive human being. Thanks to Cliff McCollum, Tom Diethe, Eno Thereska, Andreas Damianou, and Maren Mahsereci for giving me advice when I sought it.

Academia can be a solitary place, but it’s much more fun when shared with the right people. I enjoyed being a part of the ML@CL research group, and am thankful to Diana Robinson, Aditya Ravuri, Cilie Feldager, Eric Meissner, Ferenc Huszar, Carl Henrik Ek, Jess Montgomery, Markus Kaiser, Challenger Mishra, Tomasina Dada, Siyuan Guo for a friendly and fun environment, inspiring discussions, and feedback. Special thanks to Christian Cabrera-Jojoa for all the work we did together, and for his consistency and objectivity.

COVID-19 pandemic has hit the UK when I was only two months into the PhD, and has impacted just about everyone. The definite highlight of that troublesome and uncertain period of our lives is being a part of the energetic and hardworking team alongside Bobby He, Sheheryar Zaidi, Michael Hutchinson, Bryn Elesedy and Yee Whye Teh, under the umbrella of DELVE group. I truly believe our work at that time had a positive impact on people’s livelihoods.

After having a decade-long career in industry it was a weird feeling becoming an intern.
Luckily, my internship at Secondmind stopped being weird very quickly, and I would like to thank everyone at the company for the welcoming atmosphere they have created. Every morning I was looking forward to the day ahead. Special thanks goes to Victor Picheny for being my mentor and for his masterful explanations of multi-objective optimisation, and to Henry Moss for his support without which the internship would never have happened in the first place.

Our collaboration with Seldon plays a important role in this thesis, and I am grateful to Clive Cox and Sherif Akoush for welcoming me to the company, giving me the freedom to explore, and bringing to life our research ideas. Also thanks to Alex Rakowski for sharing with me the fun of preparing for and speaking at Kafka Summit London. Finally, thanks to Lucien Carata who softly but convincingly showed what systems research is really about.

Of course most important of all is the gratitude I feel to my family. Thanks to my parents Yauheni and Ala for always finding the ways to support and help me, making sure I had the best education and upbringing possible. Thanks for my sister Sasha for encouragements and positivity. Thanks to my daughter Alisa for never failing to put a smile on my face, you have helped me more than you probably realise. And thanks to my beautiful wife Nadya for her love and for sharing with me all the lows and all the highs of life.
# Contents

1 **Introduction**  
1.1 Thesis structure .............................................. 15

2 **Challenges in Deploying Machine Learning**  
2.1 Machine Learning Deployment Workflow .......................... 19  
2.2 Data Management .............................................. 21  
  2.2.1 Data collection ........................................... 21  
  2.2.2 Data preprocessing ........................................ 22  
  2.2.3 Data augmentation ........................................ 22  
  2.2.4 Data analysis ............................................. 24  
2.3 Model Learning ................................................ 24  
  2.3.1 Model selection ........................................... 24  
  2.3.2 Training .................................................... 26  
  2.3.3 Hyper-parameter selection ................................ 27  
2.4 Model Verification .............................................. 28  
  2.4.1 Requirement encoding ..................................... 28  
  2.4.2 Formal Verification ........................................ 28  
  2.4.3 Test-based Verification ................................... 29  
2.5 Model Deployment .............................................. 30  
  2.5.1 Integration ................................................ 30  
  2.5.2 Monitoring ............................................... 31  
  2.5.3 Updating ............................................... 32  
2.6 Cross-cutting aspects ........................................... 34  
  2.6.1 Ethics ..................................................... 34  
  2.6.2 Law ....................................................... 35  
  2.6.3 End users’ trust .......................................... 36  
  2.6.4 Security .................................................. 38  
2.7 Summary and Conclusions ........................................ 39
3 Data-oriented architectures for real-world machine learning systems

3.1 Data management in SOA systems

3.2 Data-Oriented Architectures Paradigm and ML Challenges

3.2.1 Data as a First Class Citizen

3.2.2 Prioritise Decentralisation

3.2.3 Openness

3.3 Survey Methodology

3.3.1 Planning Stage

3.3.2 Conducting Stage

3.4 ML Applications Survey from DOA Principles Perspective

3.4.1 Data as a First Class Citizen

3.4.2 Prioritise Decentralisation

3.4.3 Openness

3.4.4 Summary

3.4.5 Threats to validity

3.5 Open questions

3.6 Related Work

3.7 Summary and Conclusions

4 Introduction to dataflow architecture

4.1 Dataflow and von Neumann architectures

4.2 Modern adoption of dataflow

4.3 Flow-based programming

4.4 Data streaming

4.5 Summary and Conclusions

5 An Empirical Evaluation of dataflow architecture

5.1 Experiment design and implementation

5.1.1 Metrics Definition

5.1.2 Applications

5.1.3 Implementation notes

5.2 Evaluation

5.2.1 Data collection task

5.2.1.1 How much additional code is required to implement a data collection task?

5.2.1.2 How does the system’s maintainability change when data collection tasks are implemented?

5.2.1.3 How complex does the application become after implementation of the data collection task?
7.3.3 Experimentation ........................................ 130
7.4 First example .............................................. 131
7.5 Demonstrations ............................................ 133
  7.5.1 Dataflow frameworks ................................ 133
  7.5.2 Experiments .......................................... 134
7.6 Limitations and next steps ............................... 137
7.7 Summary and Conclusions ................................. 139

8 Conclusions .................................................. 141
  8.1 Summary of contributions ............................... 141
  8.2 Research agenda ......................................... 142
    8.2.1 Systems Monitoring and Shadow Systems .......... 142
    8.2.2 End-to-end Systems Optimisation .................... 143
    8.2.3 Self-adaptive and Continual Learning Systems .. 144
    8.2.4 Systems FITness .................................... 145
    8.2.5 Security and Privacy ................................ 145
  8.3 Final remarks ........................................... 146

Bibliography .................................................. 149
Chapter 1

Introduction

Developments in machine learning (ML) have led to breakthroughs in multiple research areas, such as achieving a superhuman level in games [Silver et al., 2018], generating human-like conversations [Brown et al., 2020] and predicting protein structures at an unmatched level of accuracy [Jumper et al., 2021]. Consequently, there is a growing realisation that artificial intelligence (AI) powered by ML algorithms can bring practical benefits to businesses, institutions and government agencies. For example, it is estimated that the manufacturing sector in the UK alone can generate nearly £200 billion in value through the use of AI [Department for Business, Energy and Industrial Strategy, 2017]. Similarly, the European Commission recognises AI as one of the key technologies behind the Green Deal – an initiative to build a sustainable economy and help the EU achieve carbon neutrality [European Commission, 2022]. Goldman Sachs estimates that the private sector will double the investments into AI adoption by 2025, reaching $200 billion globally compared to $92 billion in 2022 [Goldman Sachs Research, 2023].

These ambitious plans require an ability to translate cutting-edge ML research into production-ready systems. Our ability to leverage ML for practical purposes at scale requires a set of high-quality tools and tested practices that would guide the process of deploying ML models into production. Unfortunately, expectations from the promise of AI systems often do not match the reality. This fact was clearly illustrated during the COVID-19 pandemic when hundreds of AI tools were developed to detect COVID-19 in CT scans and chest radiographs, but they all were found unreliable for clinical use due to identified flaws and biases [Roberts et al., 2021]. The pandemic showed our inability to discover, collect and harness relevant data, which made it impossible to rapidly build AI systems for decision-making in critical situations, despite the vast amount of human and compute resources dedicated to solving the problem [DELVE Initiative, 2020]. In the same vein, a survey of AI adoption in manufacturing in China found that 91% of AI projects failed to meet expectations [Deloitte China, 2020].

Moreover, in those cases when AI systems are deployed to production a new challenge arises — the need to understand and explain their behaviour. Modern software applications are complex and consist of a large number of components. In AI systems some of these components
are ML models, which tend to be treated as opaque black-boxes, even if they exhibit high accuracy in finding correlations and patterns in the data. It is often necessary yet challenging to explain decisions made by these components, as well as trace causal relationships between them to effectively use the data available to answer business and technical questions. The absence of such understanding can harm developers’ ability to comprehend, improve and even use software in the long term. This effect is known as “intellectual debt” [Zittrain, 2019], and was coined by Jonathan Zittrain in analogy with “technical debt”, a term well known in the engineering community. While technical debt prevents engineers from maintaining the system, intellectual debt prevents us from understanding and explaining it. Avoiding indebtedness like this is particularly important for AI systems, because as Zittrain himself puts it “A world of knowledge without understanding becomes, to those of us living in it, a world without discernible cause and effect, and thus a world where we might become dependent on our own digital concierges to tell us what to do and when”. It is therefore imperative to design AI systems in a way that ensures we can understand and explain their inner workings.

Trustworthy AI is central to policy ambitions for AI, as articulated through the UK Government’s National AI Strategy. Characteristics such as explainability, robustness, and transparency have been identified as important components of such trustworthiness [UK Office for Artificial Intelligence, 2021]. Delivering explainable and transparent AI in practice is therefore vital to the Government’s AI governance and assurance agenda. Such issues are also the focus of major research efforts. For example, European Learning and Intelligent Systems Excellence (ELISE), a pan-European consortium of AI research hubs, dedicates a large portion of its research agenda to building transparent and secure AI systems, to designing novel software architectures that provide data access without compromising privacy, and to bridging the overall gap between research and commercial use of AI [ELISE Network, 2021]. This need for alignment between technical developments and societal interests is increasingly the focus of individual research institutions. For example University of Cambridge’s vision for expanding AI capabilities, titled “AI@Cam”, acknowledges the need for alignment between technical developments and societal interests in AI systems, with deployment of AI technologies called out as a challenge that requires careful system design and testing, domain expertise and stakeholder engagement [University of Cambridge, 2022].

To fully understand the potential impact of ML models they cannot be considered separately from the systems in which they are going to be embedded. This observation emphasises the role software and systems engineering plays in building AI systems. Motivated by the rising demand for AI-powered solutions, this thesis aims to identify software architectures that would bridge the growing gap between the successes of AI reported by the research community and the safe, robust and reliable systems that practitioners require. We seek to characterise the challenges modern AI systems face in deployment, identify new approaches to AI system design that can help the community resolve them, and find ways to build transparent AI systems with
clearly traceable causal links between components. We thus formulate the following research questions:

- **Research question 1.** What are the challenges associated with deploying ML, and which parts of the deployment process are affected?

- **Research question 2.** Are there software architectures that can address or mitigate the ML deployment challenges?

- **Research question 3.** How can software architectures aid developers’ ability to apply causal reasoning to system components?

The present thesis seeks answers to these questions.

### 1.1 Thesis structure

Here we provide an overview of the thesis structure and the contributions on which it is based. Everywhere below authors marked with “*” contributed equally to the corresponding work.

Chapter 2 surveys reports of real-life ML deployment experiences and the issues they describe. The identified issues are matched against a typical ML deployment workflow to show that the entire deployment pipeline is riddled with challenges and to argue that more data-oriented approaches to software architecture are needed to tackle them. The material in this chapter has been published as Paleyes, A., Urma, R., & Lawrence, N.D., Challenges in Deploying Machine Learning: A Survey of Case Studies, *ACM Computing Surveys*, 55, 2022 [Paleyes et al., 2022c]. Its earlier version was presented under the same title at The ML-Retrospectives, Surveys & Meta-Analyses Workshop, NeurIPS, 2020.

Having identified the need for data-oriented approaches as a way to improve the ML deployment process, Chapter 3 examines data-oriented architecture (DOA) as an existing, relatively unknown but promising software architecture paradigm. We discuss the benefits that DOA brings to ML deployment and show that even though DOA is not widely known among the software engineering community, its principles are already implicitly used for building ML-powered systems. Dataflow architecture is identified as a methodology that embodies the DOA principles. The material in this chapter is currently under review at ACM Computing Surveys journal and is available as a preprint [Cabrera et al., 2023].

Chapter 4 gives the reader an introduction to dataflow architecture, its history, principles and examples. Flow-based programming (FBP) is introduced as a particular flavour of dataflow paradigm with added abstractions relevant to later discussions. The chapter also discusses the closely related concept of data streaming.

Chapters 5 and 6 explore the benefits of dataflow architecture for the deployment of ML. Chapter 5 compares the process of deploying an ML model within the functionally equivalent
codebases of applications implemented with FBP and SOA, the latter paradigm being used as a baseline. We identify some benefits of FBP, such as higher discoverability and simpler collection of data in the system. We also identify the current limitations of the paradigm. Material in this chapter has been published as Paleyes, A., Cabrera, C., & Lawrence, N.D., An Empirical Evaluation of Flow Based Programming in the Machine Learning Deployment Context, *IEEE/ACM 1st International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2022 [Paleyes et al., 2022a]. Its preliminary version was presented as Paleyes, A., Cabrera, C., & Lawrence, N.D., Towards better data discovery and collection with flow-based programming, *Data-centric AI workshop, NeurIPS*, 2021 [Paleyes et al., 2021].

Recognising the fact that the previous chapter only explored the potential of dataflow on small synthetic examples, Chapter 6 describes a real-life open source system, Seldon Core v2 (SCv2), built following the dataflow paradigm using Apache Kafka streams. The chapter dives deep into implementation details, showcasing how dataflow and streaming can be used to create a production-ready system for ML operations that follows all DOA concepts, and why it is beneficial to do so. It also exposes some of the performance trade-offs required to provide users with data observability features. Some of the motivational material in this chapter was presented as Akoush, S. *, Paleyes, A. *, van Looveren, A., & Cox, C., Desiderata for next generation of ML model serving, *Workshop on Challenges in Deploying and Monitoring Machine Learning Systems, NeurIPS*, 2022 [Akoush et al., 2022]. Technical details described in this chapter were presented as Paleyes, A. *, & Rakowski, A. *, Dataflows for machine learning operations, *Kafka Summit*, 2023.

In Chapter 7 we explore yet another benefit of the dataflow approach to software system design, including AI systems. Namely, we identify a connection between dataflow graphs and causal graphs and argue that this relationship allows a straightforward application of causal inference to dataflow software. We use fault localisation as a concrete example of this idea and showcase it in a variety of dataflow systems and scenarios. The material in this chapter has been published as Paleyes, A. *, Guo, S. *, Schölkopf, B., & Lawrence, N.D. (2023). Dataflow graphs as complete causal graphs, *IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2023 [Paleyes et al., 2023a], and Paleyes, A., & Lawrence, N.D., Causal fault localisation in dataflow systems, *3rd Workshop on Machine Learning and Systems (EuroMLSys), EuroSys*, 2023 [Paleyes and Lawrence, 2023].

Chapter 8 concludes the thesis, summarises the main findings and discusses possibilities for further research.
Chapter 2
Challenges in Deploying Machine Learning

Machine learning is rapidly evolving from an area of academic research to an applied discipline. According to a recent global survey conducted by McKinsey, ML is increasingly adopted in standard business processes with nearly 25 per cent year-over-year growth [Cam et al., 2019]. ML enjoys growing interest from the general public [Tiwari et al., 2023], business leaders [Davenport and Ronanki, 2018] and governments [Royal Society (Great Britain), 2017].

This shift comes with challenges. Just as with any other field, there are significant differences between what works in an academic setting and what is required by a real-world system. Certain bottlenecks and invalidated preconceptions should always be expected in the course of that process. As more solutions are developed and deployed, practitioners report their experience in various forms, including academic publications, public talks and blog posts. Motivated by these reports and our personal experiences, in this chapter we aim to survey the challenges in deploying ML in production¹ with the objective of understanding what parts of the deployment process cause the most difficulties. We will provide an overview of the machine learning deployment workflow and review case studies to extract problems and concerns practitioners have at each particular deployment stage. We will also discuss cross-cutting aspects that affect every stage of the deployment workflow: ethical considerations, law, end users’ trust and security.

This is not the first survey of machine learning in production. A decade ago, such surveys were already conducted, albeit with a different purpose in mind. ML was mainly a research discipline, and it was uncommon to see ML solutions deployed for a business problem outside of “Big Tech” companies. So, the purpose of such surveys was to show that ML can be used to solve practical problems, and illustrate it with examples, as was done by Pěchouček and Mařík [2008]. Nowadays the focus has changed: ML is commonly adopted in many industries, and

---

¹By production we understand the setting where a product or a service is made available for use by its intended audience.
the question becomes not “Where is it used?”, but rather “How difficult is it to use?”

One approach to assess the current state of ML deployment for businesses is a survey conducted among professionals. Such surveys are primarily undertaken by consulting firms and encompass a variety of topics. Algorithmia’s report [Wiggers, 2019a, Hecht, 2019] goes deep into the deployment timeline, with the majority of companies reporting between 8 and 90 days to deploy a single model, and 18% taking even more time. A report by IDC [Wiggers, 2019b] surveyed 2,473 organisations and found that a significant portion of their attempted AI deployments fail, quoting lack of expertise, bias in data and high costs as primary reasons. O’Reilly has conducted an interview study that focused on ML practitioners’ work experience and the tools they use [Lorica and Paco, 2018]. Broader interview-based reports have also been produced by DotsScience [2019] and Dimensional Research [2019].

While there are several business reports on the topic, the challenges of the entire machine learning deployment pipeline are not covered nearly as widely in the academic literature. There is a burgeoning field of publications focusing on specific aspects of deployed ML, such as Bhatt et al. [2020] which focuses on explainable ML or Amershi et al. [2019] which discusses software engineering aspects of deploying ML. A large number of industry-specific surveys also exist, and we review some of these works in the appropriate sections below. However, the only general-purpose survey we found is Baier et al. [2019], which combines a literature review and interviews with industry partners. In contrast with that paper, which concentrates on the experiences of the information technology sector, we aim to cover case studies from a wide variety of industries, thus increasing the breadth of our survey. We also discuss the most commonly reported challenges in the literature in much greater detail.

Our survey in this chapter considers three main types of papers:

- Case study papers that report experience from a single ML deployment project. Such works usually go deep into discussing each challenge the authors faced and how it was overcome.

- Review papers that describe applications of ML in a particular field or industry. These reviews normally give a summary of challenges that are most commonly encountered during the deployment of the ML solutions in the reviewed area.

- “Lessons learned” papers where authors reflect on their past experiences of deploying ML in production.

To ensure that this survey focuses on the current challenges, only papers published after 2016 are considered, with a few exceptions to this rule. We also refer to other types of papers where it is appropriate, e.g. practical guidance reports, interview studies and regulations. We have not conducted any interviews as a part of this study.
2.1 Machine Learning Deployment Workflow

For the purposes of this work, we are using the ML deployment workflow definition suggested by Ashmore et al. [2021], however, it is possible to conduct a similar review with any other ML pipeline description, such as CRISP-DM [Shearer, 2000] or TDSP [Severtson et al., 2017]. In this section, we give a brief overview of the definition we are using.

According to Ashmore et al. [2021], the process of developing an ML-based solution in an industrial setting consists of four stages:

- **Data management**, which focuses on preparing data that is needed to build a machine learning model;
- **Model learning**, where model selection and training happens;
- **Model verification**, the main goal of which is to ensure the model adheres to certain functional and performance requirements;
- **Model deployment**, which is about the integration of the trained model into the software infrastructure that is necessary to run it. This stage also covers questions about model maintenance and updates.

Each of these stages is broken down further into smaller steps. It is important to highlight that the apparent sequence of this description is not necessarily the norm in a real-life scenario. It is perfectly normal for these stages to run in parallel to a certain degree and inform each other via feedback loops. Therefore, this or any similar breakdown should be considered not as a timeline, but rather as a useful abstraction to simplify references to concrete parts of the deployment pipeline. We have chosen this representation of the workflow because compared to alternatives, it defines a larger number of deployment steps, which allows for a finer classification of the challenges.

For the remainder of this chapter, we discuss common issues practitioners face at each step. We also discuss cross-cutting aspects that can affect every stage of the deployment pipeline. Where appropriate, the concrete class of ML problems is specified (such as supervised learning or reinforcement learning), although the majority of the challenges can be encountered during the deployment of many types of learning tasks. Table 2.1 provides a summary of the issues and concerns we discuss. By providing illustrative examples for each step of the workflow, we show how troublesome the whole deployment experience can be. Note that the order in which challenges appear in that table and the chapter does not necessarily reflect their severity. The impact that a particular issue can have on a deployment project depends on a large variety of...

---

There is a requirements formulation stage that Ashmore et al. do not consider a part of the ML workflow. Similarly, we see this as a challenge across many domains and thus out of the scope of this study. We refer interested readers to the work by Takeuchi and Yamamoto [2020].
factors, such as business area, availability of resources, team experience, and more. For that reason, we do not attempt to prioritise the challenges discussed in the survey.

Table 2.1: All considerations, issues and concerns explored in this study. Each is assigned to the stage and step of the deployment workflow where it is commonly encountered.

<table>
<thead>
<tr>
<th>Deployment Stage</th>
<th>Deployment Step</th>
<th>Considerations, Issues and Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data management</td>
<td>Data collection</td>
<td>Data discovery</td>
</tr>
<tr>
<td></td>
<td>Data preprocessing</td>
<td>Data dispersion, Data cleaning</td>
</tr>
<tr>
<td></td>
<td>Data augmentation</td>
<td>Labeling of large volumes of data, Access to experts, Lack of high-variance data</td>
</tr>
<tr>
<td></td>
<td>Data analysis</td>
<td>Data profiling</td>
</tr>
<tr>
<td>Model learning</td>
<td>Model selection</td>
<td>Model complexity, Resource-constrained environments, Interpretability of the model</td>
</tr>
<tr>
<td></td>
<td>Training</td>
<td>Computational cost, Environmental impact, Privacy-aware training</td>
</tr>
<tr>
<td></td>
<td>Hyper-parameter selection</td>
<td>Resource-heavy techniques, Unknown search space, Hardware-aware optimisation</td>
</tr>
<tr>
<td>Model verification</td>
<td>Requirement encoding</td>
<td>Performance metrics, Business driven metrics</td>
</tr>
<tr>
<td></td>
<td>Formal verification</td>
<td>Regulatory frameworks</td>
</tr>
<tr>
<td></td>
<td>Test-based verification</td>
<td>Simulation-based testing, Data validation routines, Edge case testing</td>
</tr>
<tr>
<td>Model deployment</td>
<td>Integration</td>
<td>Operational support, Reuse of code and models, Software engineering anti-patterns, Mixed team dynamics</td>
</tr>
<tr>
<td></td>
<td>Monitoring</td>
<td>Feedback loops, Outlier detection, Custom design tooling</td>
</tr>
<tr>
<td></td>
<td>Updating</td>
<td>Concept drift, Continuous delivery</td>
</tr>
<tr>
<td>Cross-cutting aspects</td>
<td>Ethics</td>
<td>Aggravation of biases</td>
</tr>
</tbody>
</table>
Table 2.1: Continued: All considerations, issues and concerns explored in this study. Each is assigned to the stage and step of the deployment workflow where it is commonly encountered.

<table>
<thead>
<tr>
<th>Deployment Stage</th>
<th>Deployment Step</th>
<th>Considerations, Issues and Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fairness and accountability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Authorship</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decision making</td>
</tr>
<tr>
<td>Law</td>
<td></td>
<td>Country-level regulations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abiding by existing legislation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Focus on technical solution only</td>
</tr>
<tr>
<td>End users’ trust</td>
<td></td>
<td>Involvement of end users</td>
</tr>
<tr>
<td></td>
<td></td>
<td>User experience</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Explainability score</td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td>Data poisoning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Model stealing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Model inversion</td>
</tr>
</tbody>
</table>

2.2 Data Management

Data is an integral part of any machine learning solution. The overall effectiveness of the solution depends on the training and test data as much as on the algorithm. The process of creating quality datasets is usually the very first stage in any production ML pipeline. Unsurprisingly, practitioners face a range of issues while working with data as reported by Polyzotis et al. [2018]. Consequently, this stage consumes time and energy that is often not anticipated beforehand. In this section, we describe issues concerning four steps within data management: data collection, data preprocessing, data augmentation and data analysis.

2.2.1 Data collection

Data collection involves activities that aim to discover and understand what data is available, as well as how to organise convenient storage for it. The task of discovering what data exists and where it is can be a challenge by itself, especially in large production environments typically found within organisations [DELVE Initiative, 2020]. Finding data sources and understanding their structure is a major task, which may prevent data scientists from even getting started on the actual application development. As explained by Lin and Ryaboy [2013], at Twitter this situation often happens as a result of the same entity (e.g. a Twitter user) being processed by multiple services. Internally Twitter consists of multiple services calling each other, and every service is responsible for a single operation. This approach, known in software engineering as the “single responsibility principle” [Martin, 2002], results in an architecture that is very
flexible in terms of scalability and modification. However, the flip side of this approach is that at a large scale it is very hard to keep track of what data related to the entity is being stored by which service, and in which form. Some data may only exist in the form of logs, which by their nature are not easily parsed or queried. An even worse case is the situation when data is not stored anywhere, and to build a dataset one needs to generate synthetic service API calls. Such dispersion of data creates major hurdles for data scientists because, without a clear idea of what data is available or can be obtained, it is often impossible to understand what ML solutions can achieve.

### 2.2.2 Data preprocessing

The preprocessing step normally involves a range of data cleaning activities: identification of a schema, imputation of missing values, reduction of data into an ordered and simplified form, and mapping from raw form into a more convenient format. Methods for carrying out data manipulations like this are an area of research that goes beyond the scope of this study. We encourage readers to refer to review papers on the topic, such as Abedjan et al. [2016], Patil and Kulkarni [2012], Ridzuan and Zainon [2019].

A lesser known but important problem that can also be considered an object of the preprocessing step is data dispersion. It often turns out that there can be multiple relevant separate data sources that may have different schemas, different conventions, and their own way of storing and accessing the data. Joining this information into a single dataset suitable for machine learning can be a complicated task in its own right, known as the data integration process [Nazabal et al., 2020]. An example of this is what developers of Firebird faced [Madaio et al., 2016]. Firebird is an advisory system in the Atlanta Fire Department, that helps identify priority targets for fire inspections. As a first step towards developing Firebird data was collected from 12 datasets including the history of fire incidents, business licenses, households and more. These datasets were combined to give a single dataset covering all relevant aspects of each property monitored by the Fire Department. Authors particularly highlight data joining as a difficult problem. Given the fact that buildings can be identified through their geospatial location, each dataset contained spatial data specifying the building’s address. Spatial information can be presented in different formats and sometimes contains minor differences such as different spellings. All this needed to be cleaned and corrected. These corrections can be highly time consuming and proved to be such in the Firebird project.

### 2.2.3 Data augmentation

There are multiple reasons why data might need to be augmented, and in practice one of the most problematic ones is the absence of labels. A label is a value that the ML model seeks to predict from the input data in a classic supervised learning setting. Real-world data is often
unlabeled, thus labelling turns out to be a challenge in its own right. We discuss three possible factors for the lack of labelled data: limited access to experts, absence of high-variance data, and sheer volume.

Label assignment is difficult in environments that tend to generate large volumes of data, such as network traffic analysis. To illustrate the scale of this volume, a single 1-GB/s Ethernet interface can deliver up to 1.5 million packets per second. Even with a huge downsampling rate, this is still a significant number, and each sampled packet needs to be traced to be labelled. This problem is described by Pacheco et al. [2019], which surveys applications of ML to network traffic classification, with tasks such as protocol identification or attack detection. There are two main ways of acquiring data in this domain, and both are complicated for labelling purposes:

- **Uncontrolled**, collecting real traffic. This approach requires complex tracking flows belonging to a specific application. Due to this complexity, very few works implement reliable ground truth assignment for real traffic.

- **Controlled**, emulating or generating traffic. This approach is very sensitive to the choice of tooling and its ability to simulate the necessary traffic. Studies have shown that existing tools for label assignment can introduce errors into collected ML datasets of network traffic data, going as high as almost 100% for certain applications [Dusi et al., 2011]. Moreover, these tools’ performance degrades severely for encrypted traffic.

Access to experts can be another bottleneck for collecting high-quality labels. It is particularly true for areas where expertise mandated by the labeling process is significant, such as medical image analysis [Budd et al., 2021]. Normally multiple experts are asked to label a set of images, and then these labels are aggregated to ensure quality. This is rarely feasible for big datasets due to experts’ availability. A possible option here is to use noisy label oracles [Du and Ling, 2010] or weak annotations [Peyre et al., 2017], however these approaches provide imprecise labels, which can lead to a loss in quality of the model [Ren et al., 2020]. Such losses are unacceptable in the healthcare industry, where even the smallest deviation can cause catastrophic results (this is known as The Final Percent challenge according to Budd et al. [2021]).

Lack of access to high-variance data (data that covers large parts of the feature space) can be among the main challenges one faces when deploying machine learning solutions from the lab environment to the real world. Dulac-Arnold et al. [2021] explain that this is the case for Reinforcement Learning (RL). RL is an area of ML that focuses on intelligent agents that learn by taking actions and receiving feedback from their environment. Agents make decisions on what action to take next to maximize the reward based on their previous experience. It is common practice in RL research to have access to separate environments for training and evaluation of an agent. However, in practice, all data comes from the real system, and the agent can no longer have a separate exploration policy - this is simply unsafe. Therefore the data
available becomes low-variance - very little of the state space is covered. While this approach ensures safety, it means that the agent is not trained to recognise an unsafe situation and make the right decision in it. A practical example of this issue can be seen in the area of autonomous vehicle control [Kuutti et al., 2020]. There simulations are often used for training, but complex interactions, such as friction, can be hard to model, and small variations in simulation may result in the agent not being transferable to the real world.

Budd et al. [2021] show that interface design directly impacts the quality of applications built to collect annotations for unlabeled data. They discuss a range of projects that collected labels for medical images, all of which benefited from a well-designed user interface. The authors conclude that the end user interface plays a large part in the overall success of the annotation applications.

2.2.4 Data analysis

Data needs to be analysed to uncover potential biases or unexpected distribution shifts in it. The availability of high-quality tools is essential for conducting any kind of data analysis. One area that practitioners find particularly challenging in that regard is visualisation for data profiling [Kandel et al., 2012]. Data profiling refers to all activities associated with troubleshooting data quality, such as missing values, inconsistent data types and verification of assumptions. Despite obvious relevance to the fields of databases and statistics, there are still too few tools that enable the efficient execution of these data mining tasks. The need for such tools becomes apparent considering that, according to the survey conducted by Microsoft [Kim et al., 2017], data scientists think data issues are the main reason to doubt the quality of the overall work.

2.3 Model Learning

Model learning is the stage of the deployment workflow that enjoys the most attention within the academic community. As an illustration of the scale of the field’s growth, the number of submissions to NeurIPS, a primary conference on ML methods, has quadrupled in six years, going from 1678 submissions in 2014 to 6743 in 2019 [Charrez, 2019]. Nevertheless, there are still plenty of practical considerations that affect the model learning stage. In this section, we discuss issues concerning three steps within model learning: model selection, training and hyper-parameter selection.

2.3.1 Model selection

In many practical cases, the selection of a model is decided by one key characteristic of a model: complexity. Despite areas such as deep learning and reinforcement learning gaining in

---

3For further discussion of the role user interface can play in the adoption of an ML system, see Section 2.6.3.
popularity with the research community, in practice, simpler models are often chosen. Such models include shallow neural network architectures, simple approaches based on Principal Component Analysis (PCA), decision trees and random forests.

Simple models can be used as a way to prove the concept of the proposed ML solution and get the end-to-end setup in place. This approach reduces the time to get a deployed solution, allows the collection of important feedback and also helps avoid overcomplicated designs. This was the case reported by Haldar et al. [2019]. In the process of applying machine learning to AirBnB search, the team started with a complex deep learning model. The team was quickly overwhelmed by its complexity and ended up consuming development cycles. After several failed deployment attempts the neural network architecture was drastically simplified: a single hidden layer NN with 32 fully connected ReLU activations. Even such a simple model had value, as it allowed the building of a whole pipeline of deploying ML models in a production setting while providing reasonably good performance\(^4\). Over time the model evolved, with a second hidden layer being added, but it remained fairly simple, never reaching the initially intended level of complexity.

Another advantage that less complex models can offer is their relatively modest hardware requirements. This becomes a key decision point in resource-constrained environments, as shown by Wagstaff et al. [2019]. They worked on deploying ML models to a range of scientific instruments onboard the Europa Clipper spacecraft. Spacecraft design is always a trade-off between the total weight, robustness and the number of scientific tools onboard. Therefore computational resources are scarce and their usage has to be as small as possible. These requirements naturally favour the models that are light on computational demands. The team behind Europa Clipper used machine learning for three anomaly detection tasks, some models took time series data as input and some models took images, and on all three occasions, simple threshold or PCA-based techniques were implemented. They were specifically chosen because of their robust performance and low demand for computational power.

A further example of a resource-constrained environment is wireless cellular networks, where energy, memory consumption and data transmission are very limited. Most advanced techniques, such as deep learning, are not considered yet for practical deployment, despite being able to handle high dimensional mobile network data [Challita et al., 2020].

The ability to interpret the output of a model into understandable business domain terms often plays a critical role in model selection, and can even outweigh performance considerations. For that reason decision trees (DT) [Quinlan, 1986], which can be considered a fairly basic ML algorithm, are widely used in practice. DT presents its branching logic in a transparent way that resembles a human decision process, which allows for interpretation and auditing. Hansson et al. [2016] describe several cases in manufacturing that adopt DT because of their interpretability.

\(^4\)We discuss more benefits of setting up the automated deployment pipeline in Section 2.5.3.
Banking is another industry where DT finds extensive use. As an illustrative example, it is used by Keramati et al. [2016] where the primary goal of the ML application is predicting churn in an interpretable way through if-then rules. While it is easy to imagine more complicated models learning the eventual input-output relationship for this specific problem, interpretability is a key requirement here because of the need to identify the features of churners. The authors found DT to be the best model to fulfil this requirement.

Nevertheless, deep learning (DL) is commonly used for practical background tasks that require analysis of a large amount of previously acquired data. This notion is exemplified by the field of unmanned aerial vehicles (UAV) [Carrio et al., 2017]. Data collected by image sensors is the most common UAV data modality being exploited by DL, because of the sensors’ low cost, low weight, and low power consumption. DL offers excellent capabilities for processing and presentation of raw images acquired from sensors, but computational resource demands still remain the main blocker for deploying DL as an online processing instrument on board UAVs.

### 2.3.2 Training

Model training is the process of feeding the chosen model with a collected dataset to learn certain patterns or representations of the data. One of the biggest concerns with the model training stage is the economic cost associated with carrying out the training procedure due to the computational resources required. This is often true in the field of natural language processing (NLP), as illustrated by Sharir et al. [2020]. The authors observe that while the cost of individual floating-point operations is decreasing, the overall cost of training NLP is only growing. They took one of the state-of-the-art models in the field, BERT [Devlin et al., 2019], and found out that depending on the chosen model size full training procedure can cost anywhere between $50k and $1.6m in cloud computing resources, which is unaffordable for most research institutions and many companies. The authors observe that training dataset size, number of model parameters and number of operations used by the training procedure are all contributing towards the overall cost. Of particular importance here is the second factor: novel NLP models are already using billions of parameters, and this number is expected to increase further in the near future [Benaich and Hogarth, 2020].

A related concern is raised by Strubell et al. [2019] regarding the impact the training of ML models has on the environment. By consuming more and more computational resources, ML model training is driving up energy consumption and greenhouse gas emissions. According to the estimates provided in the paper, one full training cycle utilizing neural architecture search emits an amount of CO₂ comparable to what four average cars emit in their whole lifetime. The authors stress how important it is for researchers to be aware of such impact of model training, and argue that the community should give higher priority to computationally efficient hardware and algorithms. Similar concerns around environmental impact are voiced by Bender et al. [2021].
As more businesses start using ML techniques on their users’ data, concerns are being raised over the privacy of data and how well individuals’ sensitive information is preserved over the course of the model training process [Papernot et al., 2018]. Illustrating the gravity of this concern, Shokri et al. [2017] constructed an attack for membership inference, that is to determine if a given input record was a part of the model’s training dataset. They verified it on models trained on leading ML-as-a-service providers, achieving anywhere from 70% up to 94% accuracy of membership inference. Consequently, companies have to consider privacy-aware approaches, which most likely come at the cost of the model’s accuracy. Navigating the trade-off between privacy and utility is considered an open challenge for practitioners dealing with sensitive data [Avent et al., 2020]. Some of the ways this trade-off is resolved in practice are differential privacy that explicitly corrupts the data [Dwork, 2006], homomorphic encryption that restricts the class of learning algorithm but allows for training on encrypted data [Gentry, 2009], and federated learning that distributes training across personal devices to preserve privacy, but thereby constrains the algorithms that can be used for model fitting [Konečný et al., 2015].

2.3.3 Hyper-parameter selection

In addition to parameters that are learned during the training process, many ML models also require hyperparameters. Examples of such hyper-parameters are the depth of a decision tree, the number of hidden layers in a neural network or the number of neighbours in the k-Nearest Neighbours classifier. Hyper-parameter optimisation (HPO) is the process of choosing the optimal setting of these hyper-parameters. Most HPO techniques involve multiple training cycles of the ML model. This is computationally challenging because in the worst case, the size of the HPO task grows exponentially: each new hyper-parameter adds a new dimension to the search space. As discussed by Yang and Shami [2020], these considerations make HPO techniques very expensive and resource-heavy in practice, especially for applications of deep learning. Even approaches like Hyperband [Li et al., 2017] or Bayesian optimisation [Snoek et al., 2012], which are specifically designed to minimise the number of training cycles needed, are not yet able to deal with the high dimensional searches that emerge when many hyper-parameters are involved. Large datasets complicate matters by leading to long training times for each search.

Many hyper-parameter tuning approaches require the user to define a complete search space, i.e. the set of possible values each of the hyper-parameters can take. Unfortunately, in practical use cases, this is often impossible due to insufficient knowledge about the problem at hand. Setting the hyper-parameter optimisation bounds remains one of the main obstacles preventing wider use of the state-of-the-art HPO techniques [Shahriari et al., 2016].

HPO often needs to take into account specific requirements imposed by the environment where the model will run. This is exemplified by Marculescu et al. [2018] in the context of
hardware-aware ML. In order to deploy models to embedded and mobile devices, one needs to be aware of energy and memory constraints imposed by such devices. This creates a need for customised hardware-aware optimisation techniques that efficiently optimise the accuracy of the model and the hardware jointly.

2.4 Model Verification

Verification is considered an essential step in any software development cycle as it ensures the quality of the product and reduces maintenance costs. As is the case with any software, ML models should generalise well to unseen inputs, demonstrate reasonable handling of edge cases and overall robustness, as well as satisfy all functional requirements. In this section, we discuss issues concerning three steps within model verification: requirement encoding, formal verification and test-based verification.

2.4.1 Requirement encoding

Defining requirements for a machine learning model is a crucial prerequisite of testing activities. It often turns out that an increase in model performance does not translate into a gain in business value, as Booking.com discovered after deploying 150 models into production [Bernardi et al., 2019]. One particular reason they highlight is the failure of proxy metrics (e.g. clicks) to convert to the desired business metric (e.g. conversion). Therefore, alongside accuracy measures, additional domain-specific metrics need to be defined and measured. Depending on the application these may be inspired by KPIs and other business-driven measures. In the case of Booking.com such metrics included conversion, customer service tickets or cancellations. A cross-disciplinary effort is needed to even define such metrics, as understanding from modelling, engineering and business angles is required. Once defined, these metrics should also be used for monitoring the production environment and for quality control of model updates.

Besides, simply measuring the accuracy of the ML model is not enough to understand its performance. Performance metrics should also reflect audience priorities. For instance Sato et al. [2019] recommend validating models for bias and fairness, while in the case described by Wagstaff et al. [2019] controlling for consumption of spacecraft resources is crucial.

2.4.2 Formal Verification

The formal verification step verifies that software functionality follows the requirements defined within the scope of the project. For ML models such verification could include mathematical proofs of correctness or numerical estimates of output error bounds, but as Ashmore et al. [2021] point out this rarely happens in practice. More often, quality standards are being formally set
via extensive regulatory frameworks that define what quality means and how models can be shown to meet them.

An example of where ML solutions have to adhere to regulations is the banking industry [Ananth et al., 2019]. This requirement was developed in the aftermath of the global financial crisis, as the industry realised that there was a need for heightened scrutiny towards models. As a consequence an increased level of regulatory control is now being applied to the processes that define how the models are built, approved and maintained. For instance, official guidelines have been published by the UK Prudential Regulation Authority [2018] and European Central Bank [2017]. These guidelines require ML model risk frameworks to be in place for all business decision-making solutions, and implementation of such frameworks requires developers to have extensive test suites in order to understand the behaviour of their ML models. The formal verification step in that context means ensuring that the model meets all criteria set by the corresponding regulations.

Regulatory frameworks share similarities with country-wide policies for governing the use of ML-powered technology, which we discuss in greater detail in Section 2.6.1.

2.4.3 Test-based Verification

In the context of ML, test-based verification is intended to ensure that the model generalises well to previously unseen data. While collecting a validation dataset is usually not a problem, as it can be derived from splitting the training dataset, it may not be sufficient for production deployment.

In an ideal setting, testing is done in a real-life setting, where business-driven metrics can be observed, as we discussed in Section 2.4.1. Full scale testing in a real-world environment can be challenging for a variety of safety, security and scale reasons, and is often substituted with testing in simulation. That is the case for models for autonomous vehicle control [Kuutti et al., 2020]. Simulations are cheaper, faster to run, and provide flexibility to create situations rarely encountered in real life. Thanks to these advantages, simulations are becoming prevalent in this field. However, it is important to remember that simulation-based testing hinges on assumptions made by simulation developers, and therefore cannot be considered a full replacement for real-world testing. Even small variations between simulation and the real world can have drastic effects on the system behavior, and therefore the authors conclude that validation of the model and simulation environment alone is not enough for autonomous vehicles. This point is emphasised further by the experiences from the field of reinforcement learning [Dulac-Arnold et al., 2021], where the use of simulations is a de facto standard for training agents.

Hackett et al. [2018] presented an instructive use case of how limited simulation-based testing can be. The authors were part of a team that conducted an experiment that explored a reinforcement learning based cognitive engine (CE) for running a software-defined radio unit on board the International Space Station (ISS). Preparation for the experiment included
extensive ground testing in an emulated environment that informed many hyper-parameter choices and the computational setup. Nevertheless, when the software was deployed on ISS, the actual conditions of the testing environment were so harsh the team was able to test only a subset of all planned experiments. The authors observed that despite extensive preparation CE was unable to cope with these emergency scenarios.

In addition, the dataset itself also needs to be constantly validated to ensure data errors do not creep into the pipeline and do not affect the overall quality. Data issues that go unnoticed can cause problems down the line that are difficult to troubleshoot. Breck et al. [2019] argue that such issues are common in the setup where data generation is decoupled from the ML pipeline. Data issues can originate from bugs in code, feedback loops, and changes in data dependencies. They can propagate and manifest themselves at different stages of the pipeline, therefore it is imperative to catch them early by including data validation routines in the ML pipeline.

2.5 Model Deployment

Machine learning systems running in production are complex software systems that have to be maintained over time. This presents developers with another set of challenges, some of which are shared with running regular software services, and some are unique to ML.

There is a separate discipline in engineering, called DevOps, that focuses on techniques and tools required to successfully maintain and support existing production systems. Consequently, there is a necessity to apply DevOps principles to ML systems. However, even though some of the DevOps principles apply directly, there are also a number of challenges unique to productionising machine learning. This is discussed in detail by Dang et al. [2019] which uses the term AIOps for DevOps tasks for ML systems. Some of the challenges mentioned include a lack of high-quality telemetry data as well as no standard way to collect it, and difficulty in acquiring labels which makes supervised learning approaches inapplicable and lack of agreed best practices around handling of machine learning models. In this section, we discuss issues concerning three steps within model deployment: integration, monitoring and updating.

2.5.1 Integration

The model integration step consists of two main activities: building the infrastructure to run the model and implementing the model itself in a form that can be consumed and supported. While the former is a topic that belongs almost entirely in systems engineering and therefore lies out of the scope of this work, the latter is of interest for our study, as it exposes important

---

5Readers might have also encountered term MLOps (https://ml-ops.org/).
6Please refer to Section 2.2.3 for a detailed discussion about data labelling.
aspects at the intersection of ML and software engineering. Many concepts that are routinely used in software engineering are now being reinvented in the ML context.

Code reuse is a common topic in software engineering, and ML can benefit from adopting the same mindset. Reuse of data and models can directly translate into savings in terms of time, effort or infrastructure. An illustrative case is an approach Pinterest took towards learning image embeddings [Zhai et al., 2019]. There are three models used in Pinterest internally which use similar embeddings, and initially, they were maintained completely separately, to make it possible to iterate on the models individually. However, this created engineering challenges, as every effort in working with these embeddings had to be multiplied by three. Therefore the team decided to investigate the possibility of learning a universal set of embeddings. It turned out to be possible, and this reuse ended up simplifying their deployment pipelines as well as improving performance on individual tasks.

A broad selection of engineering problems that machine learning practitioners now face is given by Sculley et al. [2015]. Many of them are known anti-patterns in engineering\(^7\), but are currently widespread in machine learning software. Some of these issues, such as abstraction boundary erosion and correction cascades, are caused by the fact that ML is used in cases where the software has to take an explicit dependency on external data. Others, such as glue code or pipeline jungles, stem from the general tendency in the field to develop general-purpose software packages. Yet another source of problems discussed in the paper is the configuration debt: in addition to all configurations, a regular software system may require ML systems to add a sizable number of ML-specific configuration settings that have to be set and maintained.

Researchers and software engineers often find themselves working together on the same project aiming to reach a business goal with a machine learning approach. On the surface there seems to be a clear separation of responsibilities: researchers produce the model while engineers build the infrastructure to run it. In reality, their areas of concern often overlap when considering the development process, model inputs and outputs and performance metrics. Contributors in both roles often work on the same code. Thus it is beneficial to include researchers in the whole development journey, making sure they own the product codebase along with the engineers, use the same version control and participate in code reviews. Despite obvious onboarding and slow-start challenges, this approach was seen to bring long-term benefits in terms of speed and quality of product delivery [Amershi et al., 2019].

\subsection{2.5.2 Monitoring}

Monitoring is one of the issues associated with maintaining machine learning systems as reported by Sculley et al. [2015]. While monitoring is crucial for the maintenance of any software service, the ML community is in the early stages of understanding what are the key

\(^7\)In software engineering an anti-pattern is understood as a common response to a recurring problem that is considered ineffective or counterproductive.
metrics of data and models to monitor and how to trigger system alarms when they deviate from normal behaviour. Monitoring of evolving input data, prediction bias and overall performance of ML models is an open problem. Another maintenance issue highlighted by this paper that is specific to data-driven decision-making is feedback loops. ML models in production can influence their own behaviour over time via regular retraining. While making sure the model stays up to date, it is possible to create a feedback loop where the input to the model is being adjusted to influence its behaviour. This can be done intentionally, as well as inadvertently, which is a unique challenge when running live ML systems.

Klaise et al. [2020] point out the importance of outlier detection as a key instrument to flag model predictions that cannot be used in a production setting. The authors name two reasons for such predictions to occur: the inability of the models to generalise outside of the training dataset and overconfident predictions on out-of-distribution instances due to poor calibration. Deployment of the outlier detector can be a challenge in its own right, because labeled outlier data is scarce, and the detector training often becomes a semi-supervised or even an unsupervised problem.

Additional insight on monitoring of ML systems can be found in Ackermann et al. [2018]. This paper describes an early intervention system (EIS) for two police departments in the US. On the surface their monitoring objectives seem completely standard: data integrity checks, anomaly detection and performance metrics. One would expect to be able to use out-of-the-box tooling for these tasks. However, the authors explain that they had to build all these checks from scratch to maintain good model performance. For instance, the data integrity check meant verifying updates of a certain input table and checksums on historical records, the performance metric was defined in terms of the number of changes in top $k$ outputs, and anomalies were tracked on rank-order correlations over time. All of these monitoring tools required considerable investigation and implementation. This experience report highlights a common problem with currently available end-to-end ML platforms: the final ML solutions are usually so sensitive to a problem’s specifics that out-of-the-box tooling does not fit their needs well.

As a final remark, we note that there is an overlap between the choice of metrics for monitoring and validation. The latter topic is discussed in Section 2.4.1.

### 2.5.3 Updating

Once the initial deployment of the model is completed, it is often necessary to be able to update the model later on to make sure it always reflects the most recent trends in data and the environment. There are multiple techniques for adapting models to new data, including scheduled regular retraining and continual learning [Diethe et al., 2018]. Nevertheless in the production setting model updating is also affected by practical considerations.

A particularly important problem that directly impacts the quality and frequency of the
model update procedure is the concept drift, also known as dataset shift [Quiñonero-Candela et al., 2009]. Concept drift in ML is understood as changes observed in joint distribution \( p(X, y) \), where \( X \) is the model input and \( y \) is the model output. Such changes can occur discretely, for example after some outside event that affects the input data distribution, or continuously, when data is gradually changing over time. Undetected, this phenomenon can have major adverse effects on model performance, as is shown by Jameel et al. [2020] for classification problems or by Celik and Vanschoren [2021] in the AutoML context. Concept drift can arise due to a wide variety of reasons. For example, the finance industry faced turbulent changes as the financial crisis of 2008 was unfolding, and if advanced detection techniques were employed they could have provided additional insights into the ongoing crisis, as explained by Masegosa et al. [2020]. Changes in data can also be caused by an inability to avoid fluctuations in the data collection procedure, as described in the paper by Langenkämper et al. [2020] which studies the effects of slight changes in marine images on deep learning models’ performance. Data shifts can have noticeable consequences even when occurring at a microscopic scale, as Zernike et al. [2019] show in their research on predictive maintenance for wear and tear of industrial machinery. Even though concept drift has been known for decades [Schlimmer and Granger, 1986], these examples show that it remains a critical problem for applications of ML today. Consequently, detection of concept drift becomes a growing concern for teams that maintain ML models in production [Soemers et al., 2018, Sun et al., 2020], which makes this challenge directly related to monitoring discussed in the previous section.

On top of the question of when to retrain the model to keep it up to date, there is an infrastructural question on how to deliver the model artefact to the production environment. In software engineering, such tasks are commonly solved with continuous delivery (CD), which is an approach for accelerating the development cycle by building an automated pipeline for building, testing and deploying software changes. CD for machine learning solutions is complicated because, unlike in regular software products where changes only happen in the code, ML solutions experience change along three axes: the code, the model and the data. An attempt to formulate CD for ML as a separate discipline can be seen in Sato et al. [2019]. This work describes the pieces involved and the tools that can be used at each step of building the full pipeline. A direct illustration of the benefits that a full CD pipeline can bring to the real-life ML solution can be found in the work of Wider and Deger [2017].

While updating is necessary for keeping a model up to date with recent fluctuations in the data, it may also inflict damage on users or downstream systems because of the changes in the model’s behaviour, even without causing obvious software errors. To study updates in teams where AI is used to support human decision-making, Bansal et al. [2019] introduced the notion of compatibility of an AI update. Authors define an update as compatible only if it does not violate the user’s trust characterised via model behaviour. The authors proceeded to show that updating an AI model to increase accuracy, at the expense of compatibility, may degrade
overall AI-Human team performance. This line of research is continued by Srivastava et al. [2020], who provide a detailed empirical study of backward compatibility issues in ML systems. They show how ML model updates may become backward incompatible due to optimisation stochasticity or noisy training datasets. These findings motivate the need for de-noising and compatibility-aware training methods as the means to ensure reliable updates of deployed ML models.

2.6 Cross-cutting aspects

In this section, we describe additional aspects that ML projects have to consider: ethics, law, end users’ trust, and security. These aspects can affect every stage of the deployment pipeline.

2.6.1 Ethics

Ethical considerations should always inform data collection and modeling activities. As stated in the report on ethical AI produced by the Alan Turing Institute [Leslie, 2019], “it is essential to establish a continuous chain of human responsibility across the whole AI project delivery workflow”. If researchers and developers do not follow this recommendation, complications may come up due to a variety of reasons, some of which we illustrate in this section.

Since ML models use previously seen data to make decisions, they can rely on hidden biases that already exist in data - a behaviour that is hard to foresee and detect. This effect is discussed in detail by O’Neil [2016] in the field of criminal justice. Models that calculate a person’s criminal “risk score” are often marketed as a way to remove human bias. Nevertheless, they use seemingly neutral demographic information like a neighbourhood that often ends up serving as a proxy for more sensitive data such as race. As a result, people are disadvantaged on the basis of race or income. Machine translation is another example of an area where such hidden biases exist in data and can be exploited via an ML system. Prates et al. [2020] show a strong tendency towards male defaults in popular online translation services in particular for fields typically associated with unbalanced gender distribution, such as STEM.

Likewise, Soden et al. [2019] mention the aggravation of social inequalities through the use of biased training datasets as one of the main hurdles in applying ML to Disaster Risk Management (DRM). It is argued that ML causes privacy and security concerns through a combination of previously distinct datasets. Reducing the role of both experts and the general public is also seen as an ethical issue by DRM professionals because they feel it increases the probability of error or misuse. Similar worries about unintentional or malign misuse of ML decision-making systems are expressed by Muthiah et al. [2016]. Their software for predicting

---

8Here by “ethics” we understand moral principles and techniques that inform the responsible development and use of ML or AI solutions.

9We discuss related cross-cutting security concerns in Section 2.6.4.
civil unrest, called EMBERS, is designed to be used as a forecasting and communication tool, however, authors remark that it can also be potentially misused by governments, either due to a misunderstanding of its role in society, or deliberately.

ML models for facial analysis often become subjects to criticism due to their unethical behaviour. For example, Buolamwini and Gebru [2018] analysed popular datasets for facial analysis and discovered them to be imbalanced on the basis of skin colour. They have also evaluated 3 commercial classification systems and showed darker-skinned females to be the most misclassified group. The authors conclude that urgent attention towards gender and skin type is needed for businesses that want to build genuinely fair facial analysis algorithms. Such questions about fairness and accountability of ML algorithms and data are studied by the branch of machine learning known as Fairness in ML [Barocas et al., 2017].

An interesting ethical aspect arises in the usage of ML in the field of creative arts, discussed by Anantrasirichai and Bull [2021]. When a trained model is used to create a piece of visual art, it is not entirely clear where the authorship of this piece resides. The questions of originality therefore require special attention. Closely related to this question is the growing concern of fake content being generated with ML, such as deepfake images and video, which can be easily used for the wrong purposes [Mirsky and Lee, 2021].

2.6.2 Law

As ML grows its influence on society’s everyday life, it is natural to expect more regulations to govern how ML models should function and how businesses, governments and other bodies can use them. Such legal frameworks can sometimes be used to guide decisions on ethics, although in general ethics and legal should be considered separate aspects.

Various countries have produced regulations to protect personal data rights. Typically, the more sensitive the information collected from the individual, the stronger the regulations governing its use. Examples of such regulations include the General Data Protection Regulation in the European Union [Rumbold and Pierscionek, 2017] and ethical screening laws in a range of Asian countries [Aljunid et al., 2012]. One domain that deals with some of the most sensitive information is healthcare. According to Han et al. [2020], many countries have strict laws in place to protect the data of patients, which makes the adoption of ML in healthcare particularly difficult. On one hand, there is no doubt that these rules are absolutely necessary to make sure people are comfortable with their data being used. On the other hand, the number of reviews, software updates and cycles of data collection/annotation that are required make it exceptionally hard to keep up with technical advances in ML, as Han et al. [2020] explain following their experience deploying ML solutions in the healthcare sector in Japan.

Legislation takes time to develop, and often cannot keep up with the speed of progress in ML. This phenomenon is well known in policymaking and has been discussed in the context of ML as well as other technological advances by Marchant [2011] or more recently by the
World Economic Forum [Malan et al., 2016, Malan, 2018]. As Malan [2018] explains, by the
time regulations are written they can already be out of date, resulting in a cat-and-mouse
game that is wasteful of resources and causes legal framework abuses. Additionally, it is generally
challenging to formulate specific and unambiguous laws for such a rapidly developing area
as ML. For example, Wachter et al. [2017a] show that GDPR lacks precise language as well
as explicit and well-defined rights and safeguards, therefore failing to guarantee the ‘right to
explanation’. As a result of these challenges, ML applications often have to abide by the existing
laws of the area where they are deployed. Minssen et al. [2020] analyse the current regulatory
approaches to medical ML in the US and Europe and discuss how existing laws evaluate ML
applications. At the same time, governments have to provide ML adoption plans. For instance,
the US Food and Drug Administration released an action plan outlining steps the agency plans
to take to produce a regulatory framework for medical ML solutions [Stephens, 2021].

Companies should not be focusing solely on the technological side of their solutions, as
DeepMind and Royal Free NHS Foundation Trust discovered while working on Streams, an
application for automatic review of test results for serious conditions. Their initial collabora-
tion was not specific enough on the use of patient data and on patient involvement overall,
which triggered an investigation into their compliance with data protection regulations. The
revised collaboration agreement was far more comprehensive and included a patient and public
engagement strategy to ensure data is being used ethically [Suleyman and King, 2017].

2.6.3 End users’ trust

Machine learning is often met cautiously by the end users [Laï et al., 2020, Royal Society
(Great Britain), 2017, Ramamoorthy et al., 2019]. On their own accord, models provide minimal
explanations, which makes it difficult to persuade end users of their utility [Han et al., 2020].
To convince users to trust ML-based solutions, time has to be invested to build that trust. In
this section, we explore ways in which that is done in practice.

If an application has a well-defined accessible audience, getting that audience involved early
in the project is an efficient way to foster their confidence in the end product. This approach is
very common in medicine because the end product is often targeted at a well-defined group of
healthcare workers and/or patients. One example is the project called Sepsis Watch [Sendak
et al., 2020]. In this project, the goal was to build a model that estimates a patient’s risk of
developing sepsis. It was not the first attempt at automating this prediction, and since previous
attempts were considered failures, medical personnel were sceptical about the eventual success
of Sepsis Watch. To overcome this scepticism, the team prioritised building trust, with strong
communication channels, early engagement of stakeholders, front-line clinicians and decision-
makers, and established accountability mechanisms. One of the key messages of this work is
that model interpretability has limits as a trust-building tool and other ways to achieve high
credibility with the end users should be considered. This aligns with conclusions made by
“Project explAIn” which found that the relative importance of explanations of AI decisions varies by context [Information Commissioner’s Office, 2019]. A similar argument is made by Soden et al. [2019], who explore the impact ML has on disaster risk management (DRM). Due to the growing complexity of the ML solutions deployed, it is becoming harder for the public to participate and consequently to trust the ML-based DRM services, such as flooding area estimates or prediction of damage from a hurricane. As a mitigation measure the authors recommend making the development of these solutions as transparent as possible, by taking into account the voice of residents in the areas portrayed by models as “at risk” and relying on open software and data whenever possible. The importance of strong communication and engagement with early adopters is also emphasised by Mutembesa et al. [2018] as they analysed their experience of deploying a nationwide cassava disease surveillance system in Uganda.

While the projects described above focused on engagement and accountability, in other circumstances explainability is the key to building the trust of the target audience. This is often the case when the users have experience and an understanding of ML. Rudin [2019] calls the ML community to stop using black box models and explaining their behavior afterwards, and instead design models that are inherently interpretable. Bhatt et al. [2020] analysed explainability as a feature of machine learning models deployed within enterprises, and found that it is a must-have requirement for most stakeholders, including executives, ML engineers, regulators, and others. Moreover, their survey showed that explainability score is a desired model metric, along with measures of fairness and robustness. Explainability is also necessary in cases where it is demanded by the existing regulations\textsuperscript{10}, and users will not trust decisions that are made automatically without provided explanations. Wang et al. [2020] describe such a requirement in the context of credit risk scoring. They observed that the XGBoost algorithm outperforms traditional scorecard approaches, but lacks the necessary explainability component. This prompted the authors to develop a custom loan decision explanation technique for XGBoost, subsequently deployed by QuickBooks Capital.

A poorly designed user interface can be one of the main obstacles to the adoption of any new technology. While this problem is not specific to ML, it is nevertheless worth mentioning as a challenge ML applications face. For example, Wang et al. [2021a] studied the deployment of the AI-powered medical diagnosis tool “Brilliant Doctor” in rural China and discovered that the majority of doctors could not use it productively. One of the main reasons quoted was the UX design that did not take into account the particularities of the environment (screen sizes, interaction with other software in the clinic) where it was installed, often resulting in unusable software. On the contrary, investing time in specialised user interfaces with tailored user experience can pay off with fast user adoption. Developers of Firebird [Madaio et al., 2016], a system that helps identify priority targets for fire inspection in the city of Atlanta, USA, found that the best way to avoid resistance from the end users while transitioning to an ML solution

\textsuperscript{10}Effects of regulations on ML deployment are also discussed in Section 2.4.2
as a replacement for the previously used pen-and-paper method was to develop a user interface that presented the results of modelling in a way that the end users (fire officers and inspectors in the Fire dept) found most useful and clear. Similarly, authors of EMBERS [Muthiah et al., 2016], a system that forecasts population-level events (such as protests), in Latin America, noticed that their users have two modes of using the system: (a) high recall: obtain most events and then filter them using other methods; (b) high precision: focus on a specific area or a specific hypothesis. To improve the user experience and thus increase their confidence in the product, the user interface was improved to easily support both modes. This case study emphasises the importance of context-aware personalisation for ML systems’ interfaces, one of the key observations delivered by “Project explAIn” [Information Commissioner’s Office, 2019].

2.6.4 Security

Machine Learning opens up opportunities for new types of security attacks across the whole ML deployment workflow [Kumar et al., 2019]. Specialised adversarial attacks for ML can occur on the model itself, the data used for training and also the resulting predictions. The field of adversarial machine learning studies the effect of such attacks against ML models and how to protect against them [Biggio and Roli, 2018, Kurakin et al., 2017]. Recent work from Siva Kumar et al. [2020] found that industry practitioners are not equipped to protect, detect and respond to attacks on their ML systems. In this section, we describe the three most common attacks reported in practice which affect deployed ML models: data poisoning, model stealing and model inversion. We focus specifically on adversarial machine learning and consider other related general security concerns in deploying systems such as access control and code vulnerabilities beyond the scope of our work.

In data poisoning, the goal of the adversarial attack is to deliberately corrupt the integrity of the model during the training phase to manipulate the produced results. In data poisoning scenarios an attacker is usually assumed to have access to the data that will ultimately be used for training, for example by sending emails to victims to subvert their spam filter [Nelson et al., 2008]. Poisoning attacks are particularly relevant in situations where the machine learning model is continuously updated with new incoming training data. Jagielski et al. [2018] reported that in a medical setting using a linear model, the introduction of specific malicious samples with an 8% poisoning rate in the training set resulted in incorrect dosage for half of the patients.

Data poisoning can also occur as a result of a coordinated collective effort that exploits feedback loops we have discussed in Section 2.5.2, as happened with Microsoft’s Twitter bot Tay [Schwartz, 2019]. Tay was designed to improve its understanding of the language over time but was quickly inundated with a large number of deliberately malevolent tweets. Within 16 hours of its release, a troubling percentage of Tay’s messages were abusive or offensive, and the bot was taken down.

Another type of adversarial attack is reverse engineering a deployed model by querying its
inputs (e.g. via a public prediction API) and monitoring the outputs. The adversarial queries are crafted to maximise the extraction of information about the model to train a substitute model. This type of attack is referred to as model stealing. In a nutshell, this attack results in the loss of intellectual property which could be a key business advantage for the defender. Tramèr et al. [2016] have shown that it is possible to replicate models deployed in production from ML services offered by Google, Amazon and Microsoft across a range of ML algorithms including logistic regression, decision trees, SVMs and neural networks. In their work, they report the number of queries ranging from 650 to 4013 to extract an equivalent model and in time ranging from 70s to 2088s.

A related attack is that of model inversion where the goal of the adversarial attack is to recover parts of the private training set, thereby breaking its confidentiality. Fredrikson et al. [2015] have shown that they could recover training data by exploiting models that report confidence values along with their predictions. Veale et al. [2018] emphasise the importance of protecting against model inversion attacks as a critical step towards compliance with data protection laws such as GDPR.

### 2.7 Summary and Conclusions

In this chapter, we examined problems, issues and difficulties that arise over the course of deploying ML into production, and mapped them to steps of the deployment workflow. We showed that practitioners deal with challenges at every step of the ML deployment process. We discussed challenges that arise during the data management, model learning, model verification and model deployment stages, as well as considerations that affect the whole deployment pipeline including ethics, end users’ trust, law and security. We illustrated them with examples across different fields and industries by reviewing case studies, experience reports and the academic literature.

It is worth the academic community’s time and focus to think about these problems, rather than expect each applied field to figure out its own approaches. We believe that ML researchers can drive improvements to the ML deployment experience by exploring holistic approaches and taking into account practical considerations. Additionally, ML shares a lot of similarities with traditional computer science disciplines, and consequently faces similar challenges, albeit with its peculiarities. Therefore ML as a field would benefit from a cross-disciplinary dialogue with such fields as software engineering, human-computer interaction, systems, and policymaking. Many pain points we have described were already experienced by communities in these fields, and the ML community should turn to them for solutions and inspiration.

To make the ML deployment scalable and accessible to every business that may benefit from it, it is important to understand the most critical pain points and to provide tools, services and best practices that address those points. As we observed multiple times in this chapter a
significant amount of challenges we saw, particularly those around data management and model
deployment, originate from the underlying software systems. Modern software architectures
do not fit well with the requirements of ML deployment projects. In the next chapter, we will
understand why this is the case, what design decisions were made in existing ML deployment
projects to overcome those challenges, and what architectural paradigm is better suited for ML
deployment.
Chapter 3

Data-oriented architectures for real-world machine learning systems

In the previous chapter we saw that, despite all the attention machine learning (ML) received recently, practical applications of ML are still riddled with challenges and complications. Nevertheless, such applications are being done in large numbers. Naturally ML projects involve a significant software engineering effort, and in this chapter, we will focus on the software architecture paradigm that can improve developer’s experience of deploying ML.

Artificial intelligence (AI) solutions are receiving a lot of attention, and have already been used to solve challenging problems in domains as diverse as healthcare, agriculture, robotics, physics, and transportation [LeCun et al., 2015]. Their success has been driven by the growth of available data, increasingly powerful hardware, and the development of novel ML models [Došilović et al., 2018]. Many of these models were originally developed in academic environments, but their demonstrated practical value has led to their rapid adoption in real-world software systems. The design, implementation, deployment, and maintenance of real-world systems that integrate ML models present developers with additional challenges due to the contrast between real-world environments and the more controlled environments from which ML models originate. In particular, real-world environments produce large amounts of heterogeneous, dynamic, high-dimensional, and sometimes sensitive data, which might need to be processed in real-time [Joshi, 2007, Cabrera and Clarke, 2019]. In addition, as any other modern system ML-based systems deployed in the real world must be scalable, fast, adaptable, secure, and autonomous while enabling data availability, reuse, monitoring, and trust [Polyzotis et al., 2018, Paleyes et al., 2022c, Lwakatare et al., 2020].

Currently, the most common software design paradigm for the development of distributed enterprise-level systems is Service-Oriented Architecture (SOA) and its variant known as microservices [O’Reilly, 2020, Aniche et al., 2019]. These paradigms are formulated around services that separate systems’ functionalities and communicate through well-defined programming interfaces (APIs) [Taibi et al., 2018]. Services are often deployed in flexible cloud
platforms, which makes systems highly available and scalable [Karabey Aksakalli et al., 2021]. SOA and microservices are well equipped to address some, but not all, of the requirements that the real world poses to ML-based systems. For example, separation of concerns facilitates maintenance by following the divide-and-conquer principle [Killalea, 2016, Munaf et al., 2019], but data observability requirements are hard to satisfy because services hide the systems’ data behind their programming interfaces. This situation is known as “The Data Dichotomy”: while high-quality data management requires exposing systems’ data, services are designed to hide it [Stopford, 2016]. Traditionally in software engineering data and logic are kept separate, to allow them to evolve independently. However, ML algorithms require data to learn, leading to a tight coupling between data and logic, thus violating some of the conventional principles of software engineering. Consequently, data-related tasks require additional efforts from developers and data scientists when working with systems built with existing paradigms [Paleyes et al., 2022a].

Data-Oriented Architecture (DOA) is an emerging software engineering paradigm that aims to support data-related tasks by design while creating loosely coupled, decentralised, scalable and open systems. DOA proposes to achieve these goals by considering data as the common denominator between disparate system components [Joshi, 2007, Ning et al., 2019]. The components in DOA are distributed, autonomous, and communicate with each other at the data level (i.e. data coupling) using asynchronous message exchange protocols [Vorhemus and Schikuta, 2017]. These design principles allow DOA-based systems to achieve desirable properties such as data availability, reusability, and monitoring, as well as systems adaptability, scalability, and autonomy [Joshi, 2007, Vorhemus and Schikuta, 2017, Ning et al., 2019]. Systems that possess these properties are better equipped for adopting and integrating ML models in production environments.

This chapter presents a survey of real-world systems based on ML from a data-oriented software architecture perspective. Even though the majority of existing reports on ML deployments do not mention DOA explicitly, their authors had to resolve the same challenges that DOA aims to solve, and thus implicitly embedded some of the DOA principles in their projects to achieve successful delivery. By observing commonalities among the existing deployed ML applications, we evaluate to what extent DOA principles are implemented in practice and distil a set of DOA best practices. We first contextualise the DOA principles through the ML-based systems challenges. Then we survey current deployed ML-based systems from a data-oriented architecture perspective. We analyse the commonalities between these systems and discuss why, how and to what extent DOA principles have been adopted in practice. Finally, we formulate practical advice, open challenges, and future research directions for academics and industry experts interested in DOA.
3.1 Data management in SOA systems

Before presenting DOA, its principles and our findings on their adoption, we shall explain in greater detail why SOA presents data scientists with problems.

Deployment of ML in production faces a variety of challenges that affect every step of the workflow. Data discovery and collection is one of the primary areas of concern [Polyzotis et al., 2018, Lwakatare et al., 2020]. In this section, we explain how these problems manifest in the currently most prevalent enterprise software design approach.

Modern systems are built around a “service-client” relationship. If a software client, perhaps a utility or a service or an object, needs an output of a certain operation, and there is a service capable of performing this operation, the client makes a call to that service passing in some input data through the service’s API, and receives a reply with the output data. An architectural paradigm that is built entirely on this type of relationship is known as service-oriented architecture (SOA) [Perrey and Lycett, 2003]. Service orientation gives developers a range of important benefits: encapsulation, loose coupling, modularity, scalability and ease of integration with other systems, so it is a reasonable choice for those who need to build scalable software [Papazoglou and Georgakopoulos, 2003]. In recent years an SOA derivative known as microservices gained substantial popularity and can be considered a de facto standard distributed software design approach [O’Reilly, 2020].

Approaches that rely on API calls can often make data ephemeral. That hinders efforts to re-use this data, for example in model training or validation, presenting a critical challenge for systems that are focused on data-driven decision-making or data analysis. To illustrate why this is the case, let us consider a system of two services, A and B, where B is making a call to A, A does some computations internally, and then returns output to B, as shown in Figure 3.1. Imagine we would like to collect a dataset that allows us to study the behaviour of service A. That could be for the purposes of measuring its performance, running a business analysis task, or training a machine learning model. For that, we would need a set of \((X, y)\) pairs, where each \(X\) is an input that B sent to A, and each \(y\) is the corresponding output that A sent back to B. In the current architecture, there is nothing that guarantees such pairs were ever recorded. Even more so, there is no guarantee that \(X\) or \(y\) are available anywhere as separate data entries.

Consider a more complex system with a third service C (Figure 3.2). In this system, before replying to service B, service A also makes an interim call to service C to collect some additional input it needs to complete the computation. Here service A introduces a hidden state, which complicates the task of collecting all the data required to describe A’s behavior. Not only do we need to store and match inputs from B to the outputs, but we also need to match each such pair to the corresponding exchange of calls with C. The problem quickly grows with the number of

---

1By service, or software service, we understand a software functionality or a set of software functionalities provided via an interface also known as an API, that different clients can reuse for different purposes by calling the named API.
Figure 3.1: System of two software services. Service B sends a request with some input to service A, and receives a response with some output. Input and output data are passed over the network and are not necessarily stored anywhere.

Figure 3.2: System of three software services. Service A makes an interim call to service C before generating a response to B’s request. All inputs and outputs are passed over the network, and without separate effort, it is difficult to match the response from C to the corresponding request from B.

In practice, some of the necessary data can be found in logs and databases. However, the decision on what data to store and where is left to developers of services A, B and C. Moreover, developers of different services within one software system may choose different technologies and formats for their logs and databases. In fact, one of the main features of services is their ability to completely hide such implementation details from the rest of the system. In Confluent, the software company behind Apache Kafka, this situation is known as “The Data Dichotomy”: while high-quality data management is about exposing data, services are about hiding it [Stopford, 2016].

When data scientists are presented with a business problem, their first step is to understand what data is available within the system and to collect it into a dataset suitable for ML model training. With the majority of data hidden behind services, data scientists have to spend significant time discovering this data. This involves talking to service owners, examining databases, parsing logs, merging multiple existing data sources, and partnering with software engineering teams to collect additional data. These struggles are described in detail by Lin and Ryaboy [2013], who describe how the scalable service-oriented architecture in Twitter becomes a source of multiple problems for data science projects inside the company. Authors point out that while each service outputs rich logging information which could, in theory, serve as a dataset source, in practice these logs are usually inconsistent, incomplete and difficult to
parse. Furthermore, the single responsibility principle that drives SOA architectures means it is difficult for data scientists to collect a complete dataset, and many data science projects get stuck at the data discovery stage. Similar observations are made by Nazabal et al. [2020], who collectively call these issues "data organization" and recognise them as one of the major challenges data scientists have to solve.

The main point is that there is nothing embedded in the SOA design paradigm that simplifies data discovery and collection. Any data-driven task within the boundaries of the system requires separate efforts from service developers and data scientists. Organisational measures, such as team structures and conventions, can address this to a certain extent. However, organisational measures on their own scale poorly and tend to break in the long term, and shall be supported by strong and agile architectural framework [Larsson et al., 2007, Nord et al., 2014]. This motivated our search for an alternative architecture paradigm that considers data a first-class citizen of a software system, thus fulfilling the growing demand for creating data-driven solutions.

Given the popularity of SOA, many attempts have been made to re-think this software design approach and enhance it with better data handling capabilities. Götz et al. [2018] proposes a data-driven approach towards designing microservices, which requires a complete data picture of a business as a prerequisite - something that can be difficult to acquire for any medium-to-large-sized business because of a sheer number of data flows and no built-in solutions to track them. Uber Engineering’s Domain-Oriented Microservice Architecture [Gluck, 2020] can be seen as a step forward in this line of thinking, as this design approach only requires a high-level understanding of main entities and the domains a business operates with. On a different note, Safina et al. [2016] extends Jolie, a programming language based on the microservices paradigm, in a way that allows to build services with Jolie in a data-driven fashion. Dehghani [2019] from technology consultancy ThoughtWorks proposed data meshes - a distributed data storage architecture that replaces monolithic data lakes, which improves data ownership and decouples data processing pipelines but still hides data behind services. Contrary to this strand of software engineering research, rather than incrementally improving SOA, we wanted to consider using radically different approaches for better data management in software development, which led us to the discovery of data-oriented architecture, which we now describe.

### 3.2 Data-Oriented Architectures Paradigm and ML Challenges

Having an intuitive understanding of the shortcomings of service-oriented approaches to software engineering, we now turn our attention to data-oriented architecture, an alternative paradigm that has the potential to mitigate many of the issues of ML deployment.

Data-Oriented Architecture (DOA) is an emerging software architecture paradigm for
building systems that aims to close the same gap. Compared with SOA, DOA places strong emphasis on data that is highly available by design. Such availability facilitates data monitoring and adaptation to guarantee its quality in the whole system. DOA proposes a set of principles for building data-oriented software: considering data as a first-class citizen, decentralisation as a priority, openness [Joshi, 2007, Schuler et al., 2015, Vorhemus and Schikuta, 2017, Ning et al., 2019]. Systems that follow these principles achieve high data availability while treating it in a reusable and maintainable way. They also display scalability, resilience, and autonomy. These properties can help software developers mitigate the challenges that real-world environments pose to the deployment of ML, and that are difficult to address or require additional efforts.
when following current paradigms (e.g., SOA). Figure 3.3 maps ML deployment challenges and DOA principles addressing them. The left side of the figure shows the challenges of the ML workflow at deployment discussed in the previous chapter, while the right side shows the DOA principles described in the literature [Joshi, 2007, Schuler et al., 2015, Vorhemus and Schikuta, 2017, Ning et al., 2019].

The remainder of this section discusses each DOA principle in greater detail, while in Section 3.4 we analyse to what extent these principles have been applied in practice.

3.2.1 **Data as a First Class Citizen**

For the absolute majority of software systems, their outputs and overall performance depend on the quality of the data that flows through the system’s components [Fisler et al., 2021], and the magnitude of that dependency increases with the addition of ML models into the system. Data collection, augmentation, preprocessing, analysis, and monitoring become critical. Performing these activities in real-world environments can be challenging as they usually generate high volumes of variable data, which needs to be processed in real time to create value for organisations and individuals. The properties of such data are also known as “the five Vs” of big data: volume, variability, velocity, veracity, and value [Nazabal et al., 2020]. This data must be monitored at run time to evaluate the system’s performance, identify or predict failures, and trigger adaptations in its components. It means that data management influences the model selection, parametrisation, training, testing, verification, and updating the ML-based components [Paleyes et al., 2022c]. The location of the data and the way it flows between systems components impact the privacy, transparency, and security properties of systems at deployment. Components in current software design paradigms (e.g., web services or microservices) usually fall into “The Data Dichotomy” as they hide their data behind service interfaces, while data management requires exposing data [Stopford, 2016]. The dichotomy does not suit data management tasks as additional efforts are needed to access the data that flows through the system. Such data unavailability complicates systems’ monitoring and the tasks that depend on it (e.g., learning models management) [Paleyes et al., 2022a].

DOA proposes to treat the **data as a first-class citizen** of a software system, understanding data as the common denominator between disparate components [Joshi, 2007]. It means that the data in a DOA-based system is primary and the operations on data are secondary. This principle makes systems **data-driven** by design, which matches the nature of ML-based components. DOA-based systems rely on a **invariant shared data model** which is processed and nourished by multiple system components [Schuler et al., 2015]. The shared data model is a single data structure equivalent to the one that data engineers build into the data management stage of the ML workflow. The key difference is that this data model is automatically built from the system components’ interactions. Systems components do not expose any APIs, and instead interact via data mediums, where input is listened to and output is offloaded [Schuler et al., 2015]. Such
interactions enable DOA-based systems to achieve data coupling, considered as the loosest form of coupling [Olsson, 2014, Offutt et al., 1993]. The shared data model stores the history of the system’s state during its whole life cycle. The systems’ data is fully available, describing the systems’ current and past states. Data availability facilitates data management tasks, learning model management, systems monitoring, failure detection, and adaptation. Components’ behaviour is programmatically observable, traceable, and auditable. Such transparency helps the responsible design of ML-based systems to resolve questions regarding ethics, trust, and security.

3.2.2 Prioritise Decentralisation

Big tech companies drove recent ML breakthroughs thanks to the increasingly powerful hardware and the efforts in building massive data sets [LeCun et al., 2015, Došilović et al., 2018]. These enablers are not always present in systems deployed in the real world. Smaller organisations are using ML to solve challenging problems around the world (e.g. sustainable farming, epidemics, climate change, etc.) with limited budgets and resources. Most organisations worldwide cannot afford expensive cloud computing resources or the effort of building and maintaining massive learning models. Such a reality threatens the practical and democratic adoption of ML models as different stages of their workflow are computationally expensive. For example, the training stage is an iterative process that solves an optimisation problem to find learning model parameters. This process is expensive in complex models (e.g., large language models) that learn from non-quadratic, non-convex, and high dimensional data sets [Judd, 1990, Orr and Müller, 2003, Goodfellow et al., 2017]. Hyperparameters improve the efficiency of the training process as well as the accuracy of the learning models [Goodfellow et al., 2015]. However, the selection of these hyperparameters is also a resource-demanding optimisation problem [Paine et al., 2020, Bischl et al., 2021]. In addition, expensive and centralised cloud deployments can fail to meet low-latency data management requirements from critical applications. The physical distance between end users and cloud data centres impacts systems’ end-to-end response time [Cabrera et al., 2022]. Data ownership, security, and trust requirements are also affected when systems are centralised. The ownership of the data changes from end users to cloud providers, which can entail privacy and security issues as end users are unaware of where and how their data is stored, processed, and used [Akbar et al., 2023].

The simplest way to deliver a shared data model (Principle 3.2.1) would seem to be to centralise it, but in practice scalability, resource constraints, low latency, and security requirements mean that in DOA we **prioritise decentralisation** [Schuler et al., 2015, Vorhemus and Schikuta, 2017]. Such decentralisation should be logical and physical. Logical decentralisation enables organisations to scale when developing ML-based systems as different development  

---

2 An example of such organisation is Data Science Africa (DSA): http://www.datascienceafrica.org/aboutus/
teams focus on smaller systems’ components. Logical decentralisation is a way to achieve a clear separation of concerns, avoid central bottlenecks in the computational process, and increase flexibility in system development. Physical decentralisation enables the deployment of ML models in constrained environments where there is no access to expensive computational resources and improves privacy and ownership of data. Components of ML-based systems should be deployed as decentralised entities that store data chunks of the shared information model described in the previous principle. These entities first perform their operations with their local resources (i.e. local first, Kleppmann et al. [2019]). If local resources (i.e., data, computing time, or storage) are not enough, entities can connect temporally with other participants to share resources. Entities first scan their local environment for potential resources they need. They prioritise interactions with nodes in the close vicinity to share or ask for data and computing resources (i.e., peer-to-peer first). Cloud servers are used as fallback mechanisms. The decentralisation principle enables the system’s data replication by design as different entities can store the same data chunk. Such replication provides data availability because if one entity fails, its information is not lost. Similarly, replication provides scalability as different entities can respond to concurrent data requests. Prioritising decentralisation also alleviates the high demand for resources of ML-based systems as they can perform their data-related tasks (e.g., ML model training) in data sets that are partitioned by design. In addition, decentralisation creates a flexible ecosystem where resources from different devices can be used on demand. This DOA principle advocates for a more sustainable and democratic approach that prioritises the computational power available in everyday devices over the expensive cloud resources. It is important to note that full decentralisation, logical or physical, is not always necessary, for example, there is no need to perform local first computations when large centralised computational resources are available. However, data replication, partitioned data sets, and flexible resource management are DOA-enabled properties that can still benefit even partial decentralisation of ML-based systems.

3.2.3 Openness

The development of ML components often involves the automation of various stages of data processing, both at the training and inference stages. Such automation is mainly required because of the amounts of data ML-based systems manage, the complex processes they perform, and the fact that their users are usually experts in domains different from ML (e.g. healthcare, physics, etc.) [Waring et al., 2020]. AutoML emerged as a recent sub-field that aims to automate the whole ML training life cycle including processes such as data augmentation, model selection, and hyperparameter optimisation [Escalante, 2020, Vaccaro et al., 2021]. Nevertheless, real-world environments set automation requirements for adopting ML models that are not covered by AutoML. ML-based systems rely on the interaction of a significant number of heterogeneous components. These components must be integrated, composed, monitored, and adapted in a
Figure 3.4: Survey process that depicts the steps from the review need identification to the full-text reading of the selected papers. This process is based on the methodology proposed by Kitchenham and Charters [2007], Kitchenham and Brereton [2013].

way that satisfies end-to-end system quality requirements. The scale and dynamic nature of real-world environments make human intervention infeasible when performing such tasks [Cabrera and Clarke, 2019].

DOA proposes openness as a principle that ML-based systems should follow. Systems components should be autonomous, asynchronous and communicate with each other using a message exchange protocol. This principle creates open environments where systems’ components interact autonomously [Joshi, 2007]. Systems can take advantage of such environments when adopting ML models as their components could perform the integration, composition, monitoring, and adaptation tasks in an autonomous and decentralised manner. Asynchronous entities produce their outputs and can subscribe to inputs at any time [Schuler et al., 2015]. These steps are public and explicit in favour of data trust, traceability, and transparency. Similarly, the system’s components are autonomous in deciding which data to store, which data to make public, and which data to hide for security and privacy [Vorhemus and Schikuta, 2017]. The message exchange protocol between asynchronous components replaces interface dependencies in service-oriented or microservices architectures with asynchronous messages between data producers and consumers. Asynchronous communication protocols enable data coupling, considered to be the loosest form of coupling [Olsson, 2014, Offutt et al., 1993].

3.3 Survey Methodology

The primary goal of this chapter is to assess to which extent DOA principles are already adopted, perhaps implicitly, in existing deployed ML systems. We therefore want to survey research works that have used ML to solve problems in different domains and have been deployed and tested in real-world settings. We are particularly interested in works that report the software architectures behind these systems and, if possible, the design decisions authors made toward such architectures. The selection of these works is not straightforward as a myriad of papers that apply ML in different domains have been published in recent years, making manual selection unfeasible. For this reason, we developed a semi-automatic framework based on a well-known methodology for systematic literature reviews (SLRs) in software engineering [Kitchenham and Charters, 2007, Kitchenham and Brereton, 2013]. The framework is available.
Table 3.1: Search query format used to retrieve papers. The query is composed of three search terms in conjunction. Each of them is replaced by the values in the second column.

<table>
<thead>
<tr>
<th>Search Query</th>
<th>&lt;search_term_1&gt;AND&lt;search_term_2&gt;AND&lt;search_term_3&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;search_term_1&gt;</td>
<td>&quot;autonomous vehicle&quot; OR &quot;health&quot; OR &quot;industry&quot; OR &quot;smart cities&quot; OR &quot;multimedia&quot; OR &quot;science&quot; OR &quot;robotics&quot; OR &quot;oceanology&quot; OR &quot;finance&quot; OR &quot;space&quot; OR &quot;e-commerce&quot;</td>
</tr>
<tr>
<td>&lt;search_term_2&gt;</td>
<td>&quot;machine learning&quot;</td>
</tr>
<tr>
<td>&lt;search_term_3&gt;</td>
<td>&quot;real world&quot; AND &quot;deploy&quot;</td>
</tr>
</tbody>
</table>

publicly on GitHub\(^3\) to allow the reproducibility of this work, as well as the reusability of this framework in other surveys. The framework queries the search APIs from different digital libraries to retrieve papers’ metadata (e.g., title, abstract, and citations) in an automatic fashion. It then applies syntactic and semantic filters over the retrieved papers to reduce the search space. The filtered list of papers is manually examined to select the papers to be surveyed. Figure 3.4 depicts the stages of the survey process. It has two principal stages described in the next section.

### 3.3.1 Planning Stage

The first stage consists of the review plan definition as follows:

a. **Need identification**: Section 3.2 introduced the DOA principles and how these can support software developers when addressing challenges of ML deployment in the real world. Despite these potential benefits, and several surveys in ML and its applications (see Section 3.6), it is not clear yet to what extent current ML-based systems have adopted these principles. We want to conduct a survey of deployed ML-based systems from a DOA perspective to fill this gap and to identify best practices, and open research directions toward the development of the next generation of DOAs for ML systems in the real world.

b. **Research questions**: The main research question we want to answer with this survey is *to what extent current ML-based systems have adopted the DOA principles?* The answer to this question will allow us to identify the research gaps and directions to develop the next generations of DOAs. The long-term goal of our work is to establish DOA as a mature and competitive paradigm for designing, developing, implementing, deploying, monitoring, and adapting ML-based systems.

c. **Search terms**: We want to search for papers that present ML-based systems deployed in real-world environments in different domains. Table 3.1 shows the query format and the search terms we use to retrieve such papers. The query is composed of three search terms in conjunction. Each of them is replaced by the values in the second column.

\(^3\)Semi-automatic Literature Survey: https://github.com/cabrerac/semi-automatic-literature-survey
Table 3.2: Synonyms to extend queries. Search terms in the Word column are expanded using their respective synonyms.

<table>
<thead>
<tr>
<th>Word</th>
<th>Synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;health&quot;</td>
<td>&quot;healthcare&quot;, &quot;health care&quot;, &quot;health-care&quot;, &quot;medicine&quot;, &quot;medical&quot;, &quot;diagnosis&quot;</td>
</tr>
<tr>
<td>&quot;industry&quot;</td>
<td>&quot;industry 4&quot;, &quot;manufacture&quot;, &quot;manufacturing&quot;, &quot;factory&quot;, &quot;manufactory&quot;, &quot;industrial&quot;</td>
</tr>
<tr>
<td>&quot;smart cities&quot;</td>
<td>&quot;sustainable city&quot;, &quot;smart city&quot;, &quot;digital city&quot;, &quot;urban&quot;, &quot;city&quot;, &quot;cities&quot;, &quot;mobility&quot;, &quot;transport&quot;, &quot;transportation system&quot;</td>
</tr>
<tr>
<td>&quot;science&quot;</td>
<td>&quot;physics&quot;, &quot;physiology&quot;, &quot;chemistry&quot;, &quot;biology&quot;, &quot;geology&quot;, &quot;social&quot;, &quot;maths&quot;, &quot;materials&quot;, &quot;astronomy&quot;, &quot;climatology&quot;, &quot;oceanology&quot;, &quot;space&quot;</td>
</tr>
<tr>
<td>&quot;autonomous vehicle&quot;</td>
<td>&quot;self-driving vehicle&quot;, &quot;self-driving car&quot;, &quot;autonomous car&quot;, &quot;driverless car&quot;, &quot;driverless vehicle&quot;, &quot;unmanned car&quot;, &quot;unmanned vehicle&quot;, &quot;unmanned aerial vehicle&quot;</td>
</tr>
<tr>
<td>&quot;networking&quot;</td>
<td>&quot;computer network&quot;, &quot;intranet&quot;, &quot;internet&quot;, &quot;world wide web&quot;</td>
</tr>
<tr>
<td>&quot;e-commerce&quot;</td>
<td>&quot;marketplace&quot;, &quot;electronic commerce&quot;, &quot;shopping&quot;, &quot;buying&quot;</td>
</tr>
<tr>
<td>&quot;robotics&quot;</td>
<td>&quot;robot&quot;</td>
</tr>
<tr>
<td>&quot;finance&quot;</td>
<td>&quot;banking&quot;</td>
</tr>
<tr>
<td>&quot;machine learning&quot;</td>
<td>&quot;ML&quot;, &quot;deep learning&quot;, &quot;neural network&quot;, &quot;reinforcement learning&quot;, &quot;supervised learning&quot;, &quot;unsupervised learning&quot;, &quot;artificial intelligence&quot;, &quot;AI&quot;</td>
</tr>
<tr>
<td>&quot;deploy&quot;</td>
<td>&quot;deployment&quot;, &quot;deployed&quot;, &quot;implemented&quot;, &quot;implementation&quot;, &quot;software&quot;</td>
</tr>
<tr>
<td>&quot;real world&quot;</td>
<td>&quot;reality&quot;, &quot;real&quot;, &quot;physical world&quot;</td>
</tr>
</tbody>
</table>
Table 3.3: Categories and keywords for Lbl2Vec algorithm [Schopf et al., 2021]

<table>
<thead>
<tr>
<th>Category</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;system&quot;</td>
<td>&quot;architecture&quot;, &quot;framework&quot;, &quot;platform&quot;, &quot;tool&quot;, &quot;prototype&quot;.</td>
</tr>
<tr>
<td>&quot;software&quot;</td>
<td>&quot;develop&quot;, &quot;engineering&quot;, &quot;methodology&quot;, &quot;architecture&quot;, &quot;design&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;implementation&quot;, &quot;open&quot;, &quot;source&quot;, &quot;application&quot;.</td>
</tr>
<tr>
<td>&quot;deploy&quot;</td>
<td>&quot;production&quot;, &quot;real&quot;, &quot;world&quot;, &quot;embedded&quot;, &quot;physical&quot;, &quot;cloud&quot;, &quot;edge&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;infrastructure&quot;.</td>
</tr>
<tr>
<td>&quot;simulation&quot;</td>
<td>&quot;synthetic&quot;, &quot;simulate&quot;.</td>
</tr>
</tbody>
</table>

terms in conjunction (i.e., AND operator). The first term refers to popular domains where ML has been applied. The second term filters papers that apply machine learning in these domains, and the third term filters the papers that actually deploy their solution in the real world. Search engines in scientific databases use different matching algorithms. Some of them search for exact words in the papers’ attributes (e.g., title or abstract), which can be too restrictive. We expand these queries by including synonyms for the different words in the search terms (Table 3.2). Synonyms extend queries using inclusive disjunction (i.e., OR operator) with their respective words.

d. **Source selection:** We search in the most popular scientific repositories using the APIs they offer. They are IEEEXplore⁴, Springer Nature⁵, ScienceDirect⁶, Semantic Scholar⁷, CORE⁸, and ArXiv⁹. Some popular repositories, such as the ACM digital library, could not be used as they do not provide an API to query. Nevertheless, because of the significant overlap with other sources (papers can be published in multiple libraries, or indexed by meta-repositories such as Semantic Scholar), we are confident in the sufficient coverage of our search.

### 3.3.2 Conducting Stage

The second stage consists of the review execution as follows:

a. **Automatic search:** We implemented clients that consume the APIs exposed by the selected repositories as part of our semi-automatic framework. In this step, each client parses the query in the format the respective API understands, submits the request, and stores the search results in separate CSV files. The search results are the metadata of the retrieved papers (e.g., title, abstract, publication data).

---

⁴IEEEXplore API: https://developer.ieee.org/
⁵Springer Nature API: https://dev.springernature.com/
⁶ScienceDirect API: https://www.elsevier.com/solutions/sciencedirect/librarian-resource-center/api
⁷Semantic Scholar: https://www.semanticscholar.org/product/api
⁸CORE API: https://core.ac.uk/services/api
⁹ArXiv API: https://arxiv.org/help/api/
b. **Preprocessing of Retrieved Data:** Each API provides papers metadata in its own format, there are duplicated papers between repositories, and some records can be incomplete (e.g., a paper missing an abstract). The preprocessing step prepares the data for the following steps in our semi-automated framework. It joins the papers’ metadata in a single file, cleaning the data, and removing repeated and incomplete papers. A total of 34,932 papers were selected after this step.

c. **Syntactic and Semantic Filters:** All the data of the retrieved papers are stored in a single file after the preprocessing step. However, the number of papers is still too large for manual processing. We reduce the search space by applying two filters. A syntactic filter selects the papers that talk in the abstract about real-world deployments. In particular, this filter searches for the "real world" and "deploy" words and their synonyms (Table 3.2) in the papers’ abstracts. We found that the selected papers can be classified into four categories, after the syntactic filtering. The first category includes the papers that present architectures of deployed ML-based systems, the second category includes papers that present software engineering approaches to build ML-based systems in practice, the third category includes papers that present physical implementations (e.g., edge architectures) of ML-based systems with a special focus on the infrastructure, and the final category includes papers that experiment and evaluate ML algorithms and systems based on synthetic data and simulated environments. We used an unsupervised-learning algorithm Lbl2Vec to semantically classify the selected papers in these four groups following the work proposed by Schopf et al. [2021]. Lbl2Vec requires as inputs a set of texts to classify (i.e., selected papers abstracts) and a set of predefined categories (Table 3.3). The algorithm assigns papers to the most relevant category. We use this semantic classification to select the papers that belong to the first three categories (i.e., system, software, and deploy). These filters produced a total of 5,559 papers.

d. **Semi-automatic filtering:** The syntactic filters in the previous step reduce the set of papers to a number that is more feasible to be manually explored. Our framework in this step shows the paper information to the user in a centralised interface where papers are selected as included or excluded. Such manual selection has two stages following the methodology defined by Kitchenham and Brereton [2013]. We select the papers by reading their abstracts in the first stage. Then, we filter the selected papers by skimming the full text. The papers that pass these two filters are part of the final set of selected papers. Manual filters produced a total of 101 papers and were performed by one researcher in our team.

e. **Snowballing:** We use the API from Semantic Scholar to retrieve metadata of the papers that cite the selected papers (i.e., 101 papers) from the previous stage. We apply the preprocessing stage as well as the syntactic, semantic, and manual filters to the resulting
papers from this snowballing process. The papers that pass these filters are added to the final set of selected papers after removing repeated papers. This process resulted in 2 more papers added to the selection, resulting in a total of 103 papers for full-text reading.

f. **Full-text reading:** Two researchers of our team read the 103 papers and selected 45 papers to report in this survey. We agreed on the selection criteria based on the DOA principles definition and made annotations in a centralised repository about to what extent these papers adopt the DOA principles (Section 3.2). Selection conflicts were solved by group discussions. The survey of the 45 papers from a data-oriented perspective is presented in Section 3.4.

g. **Exclusion criteria:** We exclude the following works during our semi-automatic process:

- Papers that report experiments on synthetic data or simulated environments.
- Papers that propose systems architectures or methodologies without a proper real-world deployment and evaluation.
- Papers that present isolated ML algorithms which are not part of larger systems.
- Papers with missing metadata that cannot be analysed by our framework (i.e., a paper without abstract).
- Papers that are duplicates of already included papers.
- Papers that are not written in English.
- Survey and review papers.
- Thesis and report documents.

### 3.4 ML Applications Survey from DOA Principles Perspective

This section presents the survey of ML-based systems deployed in real-world environments. The main goal of this survey is to understand to what extent and how the DOA principles (Section 3.2) have been applied in practice. The answer to this question allows us to identify the systems’ requirements that DOA principles satisfy, and the practical approaches for implementing DOA, as well as to define a research agenda to facilitate the deployment of ML-based systems. Table 3.4 shows the final list of reviewed papers against the DOA principles. We found that the ML-based systems reported in these papers can adopt the principles fully, partially, or not at all. The level of adoption depends on the requirements these systems need to satisfy, the nature of the data they handle, and the environments where they are deployed. The rest of this section quantifies and describes the adoption of each DOA principle and sub-principle.
Table 3.4: All papers reviewed in our survey. We show whether each paper adopts (fully or partially) each of the DOA sub-principles discussed in Section 3.2.

<table>
<thead>
<tr>
<th>Research work</th>
<th>Data as a First Class Citizen</th>
<th>Prioritise Decentralisation</th>
<th>Openness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data driven</td>
<td>Invariant and shared data mode</td>
<td>Data coupling</td>
</tr>
<tr>
<td>Lebofsky et al. [2019]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Herrero and Zorrilla [2022]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Zhang et al. [2016]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dai et al. [2019]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jiang et al. [2017]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Karageorgou et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Herrero and Zorrilla [2022]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Zhang et al. [2021]</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Sultana et al. [2021]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Santana et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Alonso et al. [2020]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Sarabia-Jácome et al. [2020]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Xu et al. [2018]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Alves et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Conroy et al. [2022]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Belloccioho et al. [2016]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Shih et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Gallagher et al. [2019]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Salhaoui et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Shi et al. [2019]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Nguyen et al. [2021]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Schumann et al. [2012]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Schubert et al. [2021]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Agarwal et al. [2016]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Habibi Gharakheili et al. [2019]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Müller and Salathé [2019]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Lu et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Calancea et al. [2019]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Quintero et al. [2019]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Franklin et al. [2014]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Brumbaugh et al. [2019]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Hegemier and Kelley [2021]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Gorkin et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Barachi et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Niu et al. [2017]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Qui et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Cabanes et al. [2019]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gao et al. [2016]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Amrollahi et al. [2020]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Kemsaram et al. [2020]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Bayerl et al. [2020]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Johny and Madhusoodanan [2021]</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Falcao et al. [2021]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Hawes et al. [2017]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Ali et al. [2016]</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

✓ = Adopted, – = Partially adopted, = Not adopted


3.4.1 Data as a First Class Citizen

Figure 3.5 shows the degree of adoption of the reviewed papers for the data as a first-class citizen principle. We found that all reviewed papers report data-driven systems. This result is expected as we are reviewing ML-based systems that are data-driven by nature. All the systems contain one or multiple learning models to accomplish the data tasks they are designed for. These systems’ performance depends on the data that flow through them. For example, Schumann et al. [2012] use a Bayesian network for health monitoring of space vehicles (e.g., rovers), Sarabia-Jácome et al. [2020] build a DL model for fall detection in Ambient Assisted Living (AAL) environments, Jiang et al. [2017] propose Pytheas as a data-driven approach that optimises the Quality of Experience (QoE) of applications based on network metrics, Agarwal et al. [2016] present an RL-based decision-making system that Microsoft products use for different optimisation tasks, such as content recommendation system in MSN and virtual machine management in Azure Compute.

We found that 60% of the reviewed systems fully adopt the sub-principle of handling their information using shared data models, which are implemented either using data streams or database schemes. Our review shows that stream data structures fit better data continuously produced from different sources. That is the case of environments that require systems to
process multimedia streams (e.g., video) [Gao et al., 2016, Jiang et al., 2017, Habibi Gharakheili et al., 2019, Dai et al., 2019, Falcao et al., 2021], sensors data [Schumann et al., 2012, Cabanes et al., 2019, Lu et al., 2020, Shih et al., 2020, Kemsaram et al., 2020, Zhang et al., 2021, Nguyen et al., 2021, Herrero and Zorrilla, 2022, Conroy et al., 2022], social media data [Zhang et al., 2016, Xu et al., 2018, Müller and Salathé, 2019, Shan et al., 2022], and network metrics [Santana et al., 2020, Sultana et al., 2021]. We observed that streams are particularly common in systems that are built with dataflow architecture [Culler, 1986, Paleyes et al., 2022a], where data inputs are transformed by different components while they flow through the system. A good example of a system architected in that way is described by Herrero and Zorrilla [2022]. They propose a data-intensive platform for Industry 4.0 based on the RAI4.0 reference architecture [López Martínez et al., 2021] where software and hardware components are modelled as stream producers and consumers. This platform must adapt to real-time changes in a water pump and its environment to provide predictive maintenance. Similarly, Sultana et al. [2021] present a software framework to detect Distributed Denial of Service (DDoS) attacks in real-time network traffic. This software is based on a Support Vector Machine (SVM) model deployed using the Acumos and ONAP platforms. The framework is separated into decoupled components that exchange data between them via Kafka streams. Databases are more appropriate as shared data models when the system’s functionalities require making online decisions based on individual historical data records [Bellocchio et al., 2016, Agarwal et al., 2016, Hawes et al., 2017, Lebofsky et al., 2019, Gallagher et al., 2019, Alves et al., 2020, Schubert et al., 2021]. A shared data model in such cases facilitates the storage and management of historical records and provides an efficient communication medium between systems’ components compared to synchronous API calls or Remote Procedure Calls (RPCs). For example, Alves et al. [2020] propose a system for industrial predictive maintenance based on historical data collected from IoT devices. These devices produce and write monitoring data in a database. The system’s components read such data for the respective maintenance decision-making. Schubert et al. [2021] introduce a deep learning-based system used by the Texas Spacecraft Laboratory. The system’s components store and retrieve data from AWS S3 buckets to generate large datasets of synthetic images that support on-orbit spacecraft operations. Hawes et al. [2017] present a robotic platform that stores the Robotic Operative System (ROS) messages in a document-oriented database (i.e., MongoDB). These messages describe the status of the robot and its interactions with the environment. A monitoring component reads this history to predict future environment states.

Over thirteen per cent of the reviewed papers partially adopt a shared data model between their systems’ components. That is the case for systems that combine different storage technologies in data models that are shared but not unique [Brumbaugh et al., 2019, Amrollahi et al., 2020, Karageorgou et al., 2020]. These systems must store, monitor, and trace the data status at different stages of its processing. Systems combine different storage technologies and models because the data structure and format change after each processing stage. For example, the
AIDEx platform [Amrollahi et al., 2020] queries an electronic medical records (EMR) database and passes such data to a set of microservices that predict the risk of patients’ infection sepsis based on their own data models. AIDEx creates patients’ health data streams based on the prediction results which are stored in a MongoDB instance that is then used for visualisation purposes. Karageorgou et al. [2020] propose a system for multilingual sentiment analysis of Twitter streams. This platform uses RabbitMQ queues to collect tweets in different languages, and Kafka streams to analyse them. Other systems that partially adopt a *shared data model* are the ones that must collect data from spatially distributed sources [Sarabia-Jácome et al., 2020, Bayerl et al., 2020, Alonso et al., 2020]. Systems distribute such data to satisfy low latency, resource constraints, and privacy requirements. Distributed nodes use common models to store local data, but this data is not shared between them. Subsection 3.4.2 reports more details on distributed and decentralised deployments.

Just over a quarter of the reviewed papers report systems whose components do not have shared data models. Some systems do not need shared data models because they are small and not data-intensive. These are either implemented as monoliths in the cloud [Niu et al., 2017, Gorkin et al., 2020] or edge architectures that use edge servers as gateways for data transmission and inference. Such edge solutions deploy trained learning models on embedded devices like in the monitoring and prediction systems proposed by Quintero et al. [2019], Salhaoui et al. [2020], Hegemier and Kelley [2021], Johny and Madhusoodanan [2021]. Monolithic deployments follow a client-server pattern to collect sensor data and respond to users’ requests. For example, Niu et al. [2017] deploy a recognition system of indoor daily activities for one-person household apartments. There are also systems that do not have shared data models but are complex, large and process big volumes of data. These rely on cloud data centres and follow well-known software architectures (e.g., microservices) to satisfy such requirements [Franklin et al., 2014, Ali et al., 2016, Shi et al., 2019, Calancea et al., 2019, Qiu et al., 2020, Barachi et al., 2020]. Functionalities and data of systems’ subcomponents are encapsulated and hidden behind interfaces (e.g., service APIs). Transparency and traceability requirements are not prioritised in these systems while monitoring tasks are performed by third-party cloud components (e.g., middleware solutions, Cabrera et al. [2017]).

Systems’ components in 31.1% of the reviewed papers are designed following the *data coupling* sub-principle. It means that these systems’ components interact with each other by reading from and writing to data mediums. Some of these systems adopt this principle to handle real-time data that needs to be processed and monitored continuously. We found that components of real-time systems [Schumann et al., 2012, Jiang et al., 2017, Cabanes et al., 2019, Dai et al., 2019, Karageorgou et al., 2020, Nguyen et al., 2021, Sultana et al., 2021, Herrero and Zorrilla, 2022, Brumbaugh et al., 2019, Conroy et al., 2022] act as subscribers and publishers of data to streams that represent the state of the data at different stages in a workflow. Streams-based systems make use of different technologies such as Apache Kafka [Jiang et al., 2017,
Sultana et al., 2021, Herrero and Zorrilla, 2022] or Spark Streaming [Jiang et al., 2017, Brumbaugh et al., 2019], sometimes adopting the underlying stream-based programming model for the entire system [Dai et al., 2019]. Message queues (e.g., RabbitMQ) are also used in systems to collect heterogeneous data from different sources or enable interaction between components that act in parallel [Karageorgou et al., 2020, Nguyen et al., 2021]. Systems also adopt the data coupling sub-principle to handle large amounts of data that cannot be sent through APIs or RPCs [Lebofsky et al., 2019, Schubert et al., 2021] or when they analyse historical data [Zhang et al., 2016, Agarwal et al., 2016, Alves et al., 2020, Shan et al., 2022]. Components read and write data in shared databases that reflect the data states and enable batch processing. The system proposed by Zhang et al. [2016] illustrates these types of data coupling by combining streams and databases. Social media data from Weibo and Chinese forums are stored in distributed Apache Kafka streams that are processed using Apache Storm to enable real-time sentiment analysis. The system also offers batch processing where social media is stored using the Hadoop Distributed File System (HDFS) and an HBase database. The Apache Spark machine learning library (MLlib) is used to analyse the data in a distributed fashion.

Systems that partially adopt data coupling (i.e., 15.5%) [Gao et al., 2016, Gallagher et al., 2019, Müller and Salathé, 2019, Habibi Gharakheili et al., 2019, Amrollahi et al., 2020, Shih et al., 2020, Brumbaugh et al., 2019] are the ones where some components communicate through data mediums and others using traditional process calls. APIs or RPCs are used to communicate with distributed components or external entities, while the centralised system’s components interact through data mediums. For example, systems can use API calls to collect data from heterogeneous data sources [Müller and Salathé, 2019, Amrollahi et al., 2020] or to interact with end users [Gao et al., 2016, Gallagher et al., 2019]. More than half of the reviewed papers do not adopt data coupling. These systems mostly use REST [Niu et al., 2017, Quintero et al., 2019, Gorkin et al., 2020, Santana et al., 2020, Lu et al., 2020, Barachi et al., 2020, Kemsaram et al., 2020, Johny and Madhusoodanan, 2021] or RPCs [Franklin et al., 2014, Bellochio et al., 2016, Hawes et al., 2017, Xu et al., 2018, Shi et al., 2019, Calancea et al., 2019, Bayerl et al., 2020, Qiu et al., 2020, Sarabia-Jácome et al., 2020, Alonso et al., 2020, Salhaoui et al., 2020, Falcao et al., 2021, Zhang et al., 2021, Hegemier and Kelley, 2021] for communication. These systems’ components hide the data behind their interfaces which causes the data dichotomy issue impacting data management tasks (e.g., monitoring) [Stopford, 2016].

We found that not all the systems where components share a common data model necessarily follow the data coupling sub-principle. This is the case of distributed systems where components have common local data models, but there is no interaction between them. These systems are designed to address low latency and data privacy requirements but do not need to aggregate data from geographically distributed sources for training their models. Some of these distributed systems create independent deployments that are in charge of the functionality of the whole system in a given geographical region or coverage area [Bellochio et al., 2016, Hawes et al.,
Figure 3.6: Adoption of prioritising decentralisation principle. While approximately half of the reviewed works follow “local data chunks” and “local first” principles, less than 20% use peer-to-peer type of communication.

2017, Lu et al., 2020, Kemsaram et al., 2020]. Other distributed systems are based on edge devices that play the role of intermediate nodes that collect, preprocess, and transmit data to centralised servers as well as perform inference tasks when cloud-trained learning models are deployed on them [Xu et al., 2018, Sarabia-Jácome et al., 2020, Alonso et al., 2020, Bayerl et al., 2020, Santana et al., 2020, Zhang et al., 2021, Falcao et al., 2021, Shan et al., 2022].

3.4.2 Prioritise Decentralisation

Figure 3.6 presents to what extent the reviewed papers prioritise decentralisation when deploying their ML-based systems. Almost eighteen per cent of the papers report systems where decentralised entities store and share local data chunks with minimal presence of centralised servers. Decentralised approaches [Zhang et al., 2016, Xu et al., 2018, Dai et al., 2019, Lebofsky et al., 2019, Karageorgou et al., 2020, Zhang et al., 2021, Herrero and Zorrilla, 2022, Shan et al., 2022] rely on local data chunks, which provide partitioning and replication by design. Data partition and replication enable systems to efficiently manage computing resources and be fault tolerant. That is the case of the Breakthrough Listen program presented by Lebofsky et al. [2019]. The aim of the program is to collect data from radio and optical telescopes to
search for extraterrestrial intelligence (SETI). The search is based on different strategies which include machine learning models for modulation scheme classification and outlier detection. The paper reports the software and hardware architecture behind the data collection, reduction, archival, and public dissemination pipeline. This architecture consists of clusters of storage and compute nodes that handle raw data volumes averaging 1PB per day. Decentralised storage nodes use Hierarchical Data Format version 5 (HDF5) to organise the large data sets in groups that facilitate data management along the pipeline. Another example of decentralisation is Poligraph [Shan et al., 2022], a system that detects fake news by combining ML models and human knowledge. Users send fake news detection requests to decentralised servers that run different ML models and collect news reviews from experts. Users’ requests and news are replicated using the Byzantine Fault Tolerant (BFT) protocol to mitigate experts’ unavailability (i.e., fault tolerance). Decentralised systems are also well-equipped to process intensive and sparse data sources [Zhang et al., 2016, Xu et al., 2018, Karageorgou et al., 2020, Herrero and Zorrilla, 2022]. For example, Zhang et al. [2016] propose a system for sentiment analysis of tweets from three Chinese cities (i.e., Beijing, Shanghai, Guangzhou and Chengdu). The decentralised storage and management of data is based on HBase and Kafka. Decentralised approaches must rely on distributed computing protocols and technologies to mitigate the challenges that arise from the absence of a central control entity. Karageorgou et al. [2020] use Spark for scalable sentiment analysis on multilingual data from social media, Xu et al. [2018] uses Distributed Hash Tables (DHTs) to facilitate the retrieval of social data stored by topic for spam detection, and Herrero and Zorrilla [2022] integrate Zookeeper, Kafka, and Apache Cassandra in a decentralised platform for a predictive maintenance industrial service.

Some systems partially decentralise the storage of data (17.8% of the papers). These approaches exploit decentralisation properties (e.g., closeness to data owners) while having central control of the system [Patel et al., 2014, Tabatabaee Malazi et al., 2022]. Partial decentralisation creates federated networks where collected data from sensor devices (e.g., IoT sensors, smartphones, wearables, etc.) is stored in local databases [Bellochio et al., 2016, Shi et al., 2019, Alonso et al., 2020, Hegemier and Kelley, 2021] or encoded in local learning models trained in more powerful servers (e.g., the cloud) [Jiang et al., 2017, Santana et al., 2020, Salhaoui et al., 2020, Sarabia-Jácome et al., 2020, Hegemier and Kelley, 2021]. In both cases, edge servers preprocess and filter the collected data before transmitting it to back-end servers. Such central entities have a complete view of the system’s state and perform more complex tasks (e.g., learning models training, job scheduling, resource allocation etc.). Partial distribution of data favours data ownership, privacy and security as devices and edge nodes can decide when and how to transmit the information to the cloud. Santana et al. [2020] propose a crowd management system based on Wi-Fi frames originating from people’s smartphones in the context of the SmartSantander project [Sanchez et al., 2014]. The system collects Wi-Fi frames as streams, which are preprocessed by edge nodes. This preprocessing includes data
anonymisation to protect people’s identity; and dimensionality reduction to extract the key features from the streams. A centralised processing tier stores and uses the preprocessed and anonymised data to train ML models (logistic regression classifier, naive Bayesian classifier, and random forest classifier) for crowdsensing estimation. Local data storage also facilitates ML-based solutions deployment in resource-constrained environments. Alonso et al. [2020] implement a real-time monitoring edge system for smart farming, where farmers want to reduce storage and compute costs while monitoring the state of dairy cattle and feed grain. Edge servers in the system collect, buffer, and filter data from IoT devices to eliminate possible noise and discard duplicated frames. Such filtering also reduces transmission and storage costs.

The majority of reviewed papers report ML-based systems that rely on centralised storage (i.e., 64.4%). Some of these systems are deployed on single cloud servers [Franklin et al., 2014, Ali et al., 2016, Agarwal et al., 2016, Gao et al., 2016, Niu et al., 2017, Calancea et al., 2019, Habibi Gharakheili et al., 2019, Brumbaugh et al., 2019, Müller and Salathé, 2019, Gallagher et al., 2019, Lu et al., 2020, Barachi et al., 2020, Gorkin et al., 2020, Amrollahi et al., 2020, Qiu et al., 2020, Shih et al., 2020, Sultana et al., 2021, Schubert et al., 2021, Conroy et al., 2022], where all data is stored. Cloud servers are flexible and can handle Big Data requirements like the ones described by Conroy et al. [2022]. This work presents a system that monitors and predicts COVID-19 infections to aid military workforce readiness. This system monitored personnel of the US Department of Defense during the pandemic. It had around 10,000 users in ten months, collected 201 million hours of data, and delivered 599,174 total user-days of predictive service. Similarly, Calancea et al. [2019] use the Google Cloud Datastore service to store the data behind iAssistMe, a platform to assist people with eye disabilities. In both systems, scalability is assured by increasing or reducing the number of server instances based on the number of requests. Other systems are deployed on single resource-constraint devices like mobile phones [Quintero et al., 2019, Bayerl et al., 2020], robots [Schumann et al., 2012, Hawes et al., 2017, Kemsaram et al., 2020], or small processing units (e.g., Raspberry Pi) [Cabanés et al., 2019, Alves et al., 2020, Nguyen et al., 2021, Johny and Madhusoodanan, 2021, Falcao et al., 2021]. This type of deployment fits data problems that require online and fast inference and actuation but where data storage is infeasible due to hardware or bandwidth constraints. For example, Nguyen et al. [2021] present the implementation of a neuroprosthetic hand with embedded deep learning-based control. The neural decoder is designed based on the recurrent neural network (RNN) architecture and deployed on the NVIDIA Jetson Nano. This enables the implementation of the neuroprosthetic hand as a portable and self-contained unit with real-time control of individual finger movements. Data is not stored as it is processed online.

We found that 17.8% of the reviewed systems prioritise local processing. Learning models like other systems’ components are deployed on distributed computing nodes that process users’ requests, make decisions, and provide systems’ functionalities. Decentralised processing helps to satisfy scalability and low latency requirements in data-intensive applications [Zhang
et al., 2016, Jiang et al., 2017, Lebofsky et al., 2019, Dai et al., 2019, Karageorgou et al., 2020, 
Zhang et al., 2021, Shan et al., 2022, Herrero and Zorrilla, 2022]. Dai et al. [2019] introduce 
BigDL, a distributed deep learning framework for big data platforms and workflows. Intel 
developed the BigDL framework to ease the application of deep learning in real-world data 
pipelines for its industrial users: Mastercard, World Bank, Cray, Talroo, UCSF, JD, UnionPay, 
Telefonica, GigaSpaces, and more. Data that these companies handle is dynamic, messy, and 
requires complex, iterative, and recurrent processing that is challenging to implement efficiently 
in centralised architectures. BigDL distributes both training and inference being built on top 
of the scalable architecture of the Spark compute model. The architecture of Spark partitions 
data across workers that form computing clusters. Workers apply map, filter, and reduce 
operations in a parallel fashion. The data partitioning and parallel processing enable the design 
and implementation of fault-tolerant and efficient systems based on learning models built 
from large data sets. There are few cases where distributed nodes cooperate to offer systems’ 
functionalities. These systems also follow the peer-to-peer first sub-principle and correspond 
to the 20% of the reviewed papers [Zhang et al., 2016, Jiang et al., 2017, Xu et al., 2018, Dai 
et al., 2019, Lebofsky et al., 2019, Karageorgou et al., 2020, Zhang et al., 2021, Shan et al., 2022, 
Herrero and Zorrilla, 2022]. Peer-to-peer cooperation enables systems to satisfy scalability, 
lateness, fault-tolerance, and resource management requirements of systems that handle data 
from intensive and distributed sources. That is the case with systems that analyse social media 
data, for example, sentiment analysis of tweets or spam detection [Zhang et al., 2016, Xu et al., 
2018, Karageorgou et al., 2020], or the Breakthrough Listen program [Lebofsky et al., 2019] in 
the search for extraterrestrial intelligence (SETI). The adoption of the peer-to-peer sub-principle 
requires the use of distributed computing technologies and protocols to handle the complexity 
of decentralised processing. Systems also use these technologies when they adopt the local 
first sub-principle. Examples of the use of these technologies are Zhang et al. [2016], Dai et al. 
[2019], Karageorgou et al. [2020] using Spark, Jiang et al. [2017] based on Kafka, Xu et al. [2018] 
using DHTs, Lebofsky et al. [2019] storing data on Hierarchical Data Format (HDF5), Shan 
et al. [2022] using the Byzantine Fault Tolerant (BFT) protocol, and Herrero and Zorrilla [2022] 
based on Zookeeper.

Over thirty-one per cent of the reviewed systems partially adopt the local first sub-principle. 
Different nodes are in charge of different functionalities in these systems. Centralised servers 
are in charge of computationally expensive processing such as model training and updating 
[Xu et al., 2018, Quintero et al., 2019, Shi et al., 2019, Kemsaram et al., 2020, Bayerl et al., 2020, 
Sarabia-Jácome et al., 2020, Johny and Madhusoodanan, 2021, Salhaoui et al., 2020, Hegemier 
and Kelley, 2021], global decision making [Calancea et al., 2019, Alonso et al., 2020, Santana et al., 
2020], and systems’ orchestration [Brumbaugh et al., 2019, Sultana et al., 2021]. Centralised 
servers execute these tasks because they have enough computing power and a global view of 
the system. Central processing enables easier control of the components of the system, which
can be challenging in fully decentralised systems as autonomous entities must self-govern. Such central control makes systems robust as they rely on central servers to offer scalable and highly available solutions. The works of Sultana et al. [2021] and Brumbaugh et al. [2019] are good examples of central control. They encapsulate systems’ functionalities in logically distributed containers, but their control is centralised. Kubernetes is used as a central platform to manage container instantiation, resource allocation, scheduling, and execution. Systems that partially adopt the local first sub-principle use edge nodes or mobile devices to offer data collection and preprocessing functionalities, as well as inferences when trained models are deployed on them. Edge architectures enable local inference to fit low latency and data ownership requirements as learning models are executed closer to end users [Tabatabae Malazi et al., 2022, Cabrera et al., 2022]. For example, Sarabia-Jácome et al. [2020] deploy an edge-based fall detection system for Ambient Assisted Living (AAL) environments. AAL systems process highly sensible patient data and require very low processing time. The authors propose a 3-layer fog-cloud architecture composed of medical devices, fog nodes, and a cloud server. Deep learning models are deployed in fog nodes to detect patients’ falls. The authors prove that the proposed platform provides better performance than a cloud baseline in terms of efficiency and response time. Each patient is attended by one or more fog nodes that create their own local area network and preprocess (e.g., filtering) the patient’s data. Dedicated fog nodes enable data privacy and security by design. Kemsaram et al. [2020] present the architecture design and development of an onboard stereo vision system for cooperative automated vehicles. The platform is based on a stereo camera that captures left and right images which are processed by pre-trained DNNs to perform object perception, lane perception, and free space perception. An object tracker then estimates different metrics over the classified images such as depth and radial distance, relative velocity, and azimuth and elevation angle. The system is based on a 4-layer architecture deployed in each vehicle where all its components interact to provide timely responses to guide autonomous navigation in real time.

We found that over half of the papers report centralised systems in which all their components and functionalities are deployed in central nodes. Some of these systems work in extreme resource constraint scenarios that demand deployment in single and isolated devices [Schumann et al., 2012, Hawes et al., 2017, Cabanes et al., 2019, Nguyen et al., 2021, Falcao et al., 2021]. Such deployments suit environments that require autonomous and self-contained systems because communication with back-end servers is limited or impossible. That is the case of the work proposed by Schumann et al. [2012] that reports an ML-based system deployed in rovers to enable automatic vehicle monitoring in spatial missions. The Spirit and Opportunity Mars Exploration Rovers (MER) were autonomously functional for 6 and 14 years respectively on the surface of Mars\textsuperscript{10}. These rovers performed data collection and navigation

\textsuperscript{10}NASA's MER: \url{https://mars.nasa.gov/resources/spirit-and-opportunity-by-the-numbers/}
The STRANDS Core System [Hawes et al., 2017] is designed for long-term autonomy (LTA) robot applications in security and care environments. LTA refers to the capability of robots to be in continuous operation for multiple weeks. STRANDS supported over 100 days of autonomous operations for robots performing navigation, human behaviour prediction, and activity recognition tasks. We also found that most of the centralised systems are deployed in cloud servers to exploit their flexibility and address scalability and availability requirements [Franklin et al., 2014, Agarwal et al., 2016, Ali et al., 2016, Bellochio et al., 2016, Gao et al., 2016, Niu et al., 2017, Gallagher et al., 2019, Müller and Salathé, 2019, Habibi Gharakheili et al., 2019, Barachi et al., 2020, Lu et al., 2020, Amrollahi et al., 2020, Qiu et al., 2020, Gorkin et al., 2020, Shih et al., 2020, Alves et al., 2020, Schubert et al., 2021, Conroy et al., 2022]. That is the case of iTelescope [Habibi Gharakheili et al., 2019], which is an ML-based platform for real-time video classification. This platform collects data streams from the network traffic and processes them with two learning models for video identification and resolution classification. iTelescope was deployed in a campus network, served several hundreds of users, and demonstrated good performance with tens of thousands of concurrent streams. Centralised entities rely on third-party entities to satisfy additional requirements such as monitoring and security. For example, AIDEx [Amrollahi et al., 2020] is a platform to predict patients’ risk of developing sepsis in the next 4 to 6 hours. AIDEx fetches patients’ records from a real-time EMR database and displays hourly sepsis risk scores for each patient. This platform is based on microservices for preprocessing data, executing the prediction learning model, and storing and visualising the prediction outcomes. The privacy and security of patients’ data are key requirements of the system, which are delegated to configurable security mechanisms such as firewalls and virtual private clouds (VPCs).

Systems that follow the peer-to-peer first sub-principle correspond to the 20% of the reviewed papers as we mentioned above. The rest of the papers (i.e., 80%) report systems that do not follow this sub-principle. Some of these are implemented as centralised architectures in the cloud [Franklin et al., 2014, Agarwal et al., 2016, Ali et al., 2016, Gao et al., 2016, Müller and Salathé, 2019, Habibi Gharakheili et al., 2019, Brumbaugh et al., 2019, Shih et al., 2020, Barachi et al., 2020, Amrollahi et al., 2020, Qiu et al., 2020, Sultana et al., 2021, Schubert et al., 2021]. Central entities in these deployments orchestrate the interactions between systems’ components. Such centralised control enables robust and flexible systems that are easy to scale and highly available as mentioned before. For example, Ali et al. [2016] present the architecture of ID-Viewer, which is a surveillance system for identifying infectious diseases in Pakistan. Data collection, analysis, and visualisation components process data from different areas of the country. These components are deployed as microservices in a central server to enable spatio-temporal analysis of the causes and evolution of infectious diseases. Other systems are structured as federated architectures [Bellochio et al., 2016, Niu et al., 2017, Shi et al., 2019, Calancea et al., 2019, Quintero et al., 2019, Gallagher et al., 2019, Gorkin et al.,
without horizontal interactions between nodes at the same layer. Examples of these architectures are IoT deployments in different domains such as health care [Niu et al., 2017, Quintero et al., 2019, Sarabia-Jácome et al., 2020, Conroy et al., 2022], smart buildings [Lu et al., 2020, Santana et al., 2020], and industry [Alonso et al., 2020, Alves et al., 2020]. The architectures of these platforms follow a layered pattern where there is a vertical interaction between IoT devices, edge servers, and cloud servers. Lower layers (i.e., IoT devices) are in charge of data collection, middle layers (i.e., edge servers) are in charge of data preprocessing, filtering, and transmission, and upper layers (i.e., cloud servers) are in charge of data aggregation, decision making, visualisation, and users’ interaction. There are also self-contained systems deployed in single constraint devices [Schumann et al., 2012, Hawes et al., 2017, Cabanes et al., 2019, Bayerl et al., 2020, Kemsaram et al., 2020, Nguyen et al., 2021, Falcao et al., 2021, Johny and Madhusoodanan, 2021]. These are embedded systems that combine hardware and software components for specific functions in environments that require fast reactions (e.g., neuroprosthetic devices, Nguyen et al. [2021]), or have limited connectivity with back end servers (e.g., NASA’s Mars rovers, Schumann et al. [2012]).

### 3.4.3 Openness

Figure 3.7 shows to what extent the openness principle is adopted. We found that 22.2% of systems are based on flexible architectures where new components can join automatically. These new components join and cooperate with the existing ones to achieve the systems’ goals. It creates flexible, scalable, and highly available systems where storage and computing capabilities can be extended on demand to address dynamic requirements [Franklin et al., 2014, Zhang et al., 2016, Jiang et al., 2017, Xu et al., 2018, Lebofsky et al., 2019, Dai et al., 2019, Karageorgou et al., 2020, Sultana et al., 2021, Herrero and Zorrilla, 2022, Shan et al., 2022]. For example, Shan et al. [2022] propose a decentralised architecture for Poligraph based on Byzantine Fault Tolerant (BFT) protocol. Servers and reviewers can be added to process requests for fake news detection in parallel. The number of servers changes to respond to the variability in the number of requests that Poligraph receives which enables scalability, while parallelism enables low latency. The Sifter platform [Xu et al., 2018] is an online spam detection system for social networks. It uses a recurrent neural network (RRN) for spam detection and Distributed Hash Tables (DHT) to address the fast-changing nature of topics and events. Social data is stored by decentralised servers and grouped by topics using the DHT structure to enable fast retrieval. Each server responds to spam detection requests using its local neural network. New servers can join by following the DHT protocol which extends the storage capability of Sifter. Local spam detection fits low latency requirements as requests are solved closer to end users. We observe that the technologies and protocols used to handle decentralised storage and
peer-to-peer cooperation (Section 3.4.2) also support flexible architectures where entities can join autonomously. That is the case protocols such as DHT [Xu et al., 2018], HDF5 [Lebofsky et al., 2019], Spark [Dai et al., 2019], or BFT [Shan et al., 2022], and architectural patterns such as dataflow [Sultana et al., 2021, Jiang et al., 2017], the observer [Franklin et al., 2014], or publish/subscribe [Zhang et al., 2016, Jiang et al., 2017, Karageorgou et al., 2020, Sultana et al., 2021, Herrero and Zorrilla, 2022].

There are systems based on federated [Bellocchio et al., 2016, Calancea et al., 2019, Shi et al., 2019, Alonso et al., 2020, Santana et al., 2020, Sarabia-Jácome et al., 2020, Salhaoui et al., 2020, Zhang et al., 2021, Hegemier and Kelley, 2021] and centralised [Ali et al., 2016, Niu et al., 2017, Quintero et al., 2019, Gallagher et al., 2019, Alves et al., 2020, Gorkin et al., 2020, Qiu et al., 2020, Shih et al., 2020, Lu et al., 2020, Barachi et al., 2020, Johny and Madhusoodanan, 2021, Conroy et al., 2022] architectures that partially (i.e., 46.7% of the papers) adopt the autonomous entities sub-principle. These architectures have flexible mechanisms to add new devices for sensing or integrate new users at any time. However, they are not flexible enough to include more storage or compute servers at the edge or cloud layers. Systems add new sensors to expand their data collection capabilities and new users to expand their service coverage by using interoperable

<table>
<thead>
<tr>
<th></th>
<th>Autonomous entities</th>
<th>Asynchronous entities</th>
<th>Message protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>28.261%</td>
<td>32.609%</td>
<td>32.609%</td>
</tr>
<tr>
<td>Partial</td>
<td>39.130%</td>
<td>4.348%</td>
<td>21.739%</td>
</tr>
<tr>
<td>No</td>
<td>32.609%</td>
<td>63.043%</td>
<td>45.652%</td>
</tr>
</tbody>
</table>

Figure 3.7: Adoption of openness principles. Most reviewed systems use autonomous entities, but only a third use them asynchronously. More than 50% utilise message protocols.
communication technologies. For example, Alonso et al. [2020] use Fiware\textsuperscript{11} as a middleware to manage devices that transmit the collected data using the ZigBee wireless standard. The collected data is then used to make automated real-time decisions about the state of dairy cattle and the feed grain. Niu et al. [2017] developed the SensorBox device that integrates different IoT sensors to measure environmental variables. Supervised learning models are trained with such data to classify daily activities. Barachi et al. [2020] deploys a crowdsensing platform based on DL to identify accidents and generate reports. Users act as data consumers and collectors that can be added to the network by installing a mobile application. Similarly, Calancea et al. [2019] require users to install an Android application to use the NLP-based platform that supports visually challenged people.

A third of the papers report ML-based systems whose components are not autonomous. It is mainly because they are designed as self-contained embedded systems [Schumann et al., 2012, Hawes et al., 2017, Cabanes et al., 2019, Kemsaram et al., 2020, Bayerl et al., 2020, Falcao et al., 2021, Nguyen et al., 2021] or are based on cloud-based architectures [Agarwal et al., 2016, Gao et al., 2016, Habibi Gharakheili et al., 2019, Müller and Salathé, 2019, Brumbaugh et al., 2019, Amrollahi et al., 2020, Schubert et al., 2021]. Embedded systems are deployed to perform specific tasks with limited resources. A change in the tasks of these systems implies their redesign including their hardware components. That is the case of the robots of the STRANDS project [Hawes et al., 2017] which are specifically designed for navigation, human behaviour prediction, and activity recognition tasks. Similarly, the embedded system proposed by Cabanes et al. [2019] is particularly designed to detect parking events. It uses a single 3D sensor to collect the input data for an object recognition component. Cloud-based architectures rely on external services to extend the storage or computing capabilities of the systems. This flexibility enables systems to satisfy availability and scalability requirements. For example, the Bighead framework [Brumbaugh et al., 2019] is based on Kubernetes to support the development of data-driven solutions across the whole AirBnB organisation. Such flexibility is usually achieved at the cost of the fees that cloud providers charge for the provided services.

Almost a quarter of papers report systems that are based on asynchronous entities [Schumann et al., 2012, Franklin et al., 2014, Zhang et al., 2016, Jiang et al., 2017, Dai et al., 2019, Lebofsky et al., 2019, Karageorgou et al., 2020, Schubert et al., 2021, Sultana et al., 2021, Nguyen et al., 2021, Herrero and Zorrilla, 2022]. Software components act as data producers and consumers when communicating between them. Components do not block each other and operate independently. They do not wait for responses from other components but subscribe to relevant data producers. Systems based on these loosely coupled interactions address low latency, resource constraint, and big data processing requirements because asynchronous entities enable parallelisation and decentralised computing. The papers that follow the asynchronous entities sub-principle also follow the message exchange protocol one. Autonomous entities in open environments need

\textsuperscript{11}Fiware: https://www.fiware.org/
message exchange protocols to have a common understanding with other systems’ components. These protocols describe when and how messages are produced and consumed. The sentiment analysis system proposed by Karageorgou et al. [2020] is based on asynchronous components. Data from different intensive sources is collected using the publish/subscribe pattern, which also enables different workers to process different data streams in parallel. This communication is based on RabbitMQ and Apache Kafka standards, while Spark enables a configurable number of workers. Similarly, BigDL [Dai et al., 2019] supports large-scale distributed training of deep learning models. A Spark cluster consists of different workers that compute local gradients for each model in parallel which are then aggregated. Spark provides a data-parallel functional computing model that rules how workers operate. Workers perform map, reduce, and filter operations to transform data represented as Resilient Distributed Datasets (RDD). Robotic systems are also based on asynchronous components as robots also perform tasks in parallel. For example, a rover must move while monitoring its sensors’ status [Schumann et al., 2012]. Similarly, the neuroprosthetic hand proposed by Nguyen et al. [2021] consists of three separate threads that perform data acquisition, preprocessing, and motor decoding tasks in parallel. A queues-based protocol drives the communication between these systems’ components. Data coupling (Section 3.4.1) enables asynchronous interactions by design as components write data to and read data from data mediums. We observe that most of the papers that follow the asynchronous entities sub-principle also follow the data coupling one.

Half of the papers report systems that partially adopt the asynchronous entities and the message exchange protocol sub-principles. Some of these systems use asynchronous communication to interact with end users because they are required to listen to requests continuously or notify users at any time [Calancea et al., 2019, Barachi et al., 2020, Gorkin et al., 2020, Shih et al., 2020, Shan et al., 2022]. For instance, Calancea et al. [2019] describe a system that uses the publish/subscribe communication pattern to interact with visually challenged people and support their daily activities. The system uses RabbitMQ together with the Advanced Message Queuing Protocol (AMQP) to handle data producers and consumers. Sharkeye [Gorkin et al., 2020] uses the AWS simple notification service (SNS) to send messages warning people about the presence of sharks via smartwatches. Users subscribe to the notification service following the publish/subscribe pattern too. Other systems use asynchronous entities to listen to data sources (e.g., sensor devices) that can produce data at any time [Agarwal et al., 2016, Bellochio et al., 2016, Niu et al., 2017, Shi et al., 2019, Quintero et al., 2019, Müller and Salathé, 2019, Gallagher et al., 2019, Habibi Gharakheili et al., 2019, Lu et al., 2020, Alves et al., 2020, Santana et al., 2020, Salhaoui et al., 2020, Sarabia-Jácome et al., 2020, Qiu et al., 2020, Alonso et al., 2020, Zhang et al., 2021, Conroy et al., 2022]. Bellochio et al. [2016] present the SEAL project to provide home automation solutions based on ML for building energy management and safety. SmartSEAL is the platform that manages IoT devices deployed in the buildings. It offers both asynchronous and synchronous communication using pub/sub and client-server protocols.
The pub/sub mechanism is supported by the Robot Operative System (ROS) where sensor nodes write and are subscribed to a topic. They extend ROS with a central entity (i.e., roscore) which provides naming and registration services to the devices. These services are consumed using synchronous API calls. Alves et al. [2020] propose an industrial monitoring system based on IoT nodes capable of acquiring and transmitting machine status parameters using the publish/subscribe architectural pattern (i.e., the Message Queue Telemetry Transport (MQTT) protocol). IoT nodes post their readings in a shared database that is consumed by the system’s components asynchronously. The interaction between the components that process the collected data follows traditional synchronous calls. Gallagher et al. [2019] uses both synchronous and asynchronous mechanisms in the IntelliMaV platform. IntelliMaV is a cloud-based system that applies machine learning to verify the performance of energy conservation measures in near real-time. The system is deployed in the cloud and follows a 3-layer architecture. The user tier is the access point for users through a web browser that presents options for model training, deployment and visualisation. The cloud tier is a virtual private cloud and hosts the cloud computing infrastructure of the application. The application requests are processed by this tier by running R code that is accessed through APIs. The site tier represents the pipelines that collect data from the industrial site. These pipelines push the collected data to the cloud storage in an asynchronous fashion for further processing. The smart diary tracer platform [Alonso et al., 2020] uses different protocols to collect data from different data sources. These protocols include Zigbee, LoRa, Wi-Fi, Bluetooth, and 3G which support communication with IoT sensors deployed on the crops and the barns used to feed the livestock, the cattle to monitor their health, the factories for the traceability of packaged products and energy monitoring, and the trucks that transport the dairy products. Sarabia-Jácome et al. [2020] propose a 3-layer fog-cloud architecture where new healthcare devices join using low-power wireless technologies such as Bluetooth, Zigbee, 6LoWPAN, and Wi-Fi. The Bluetooth protocol is also used by Quintero et al. [2019] to include new devices to collect photoplethysmography data and heart rate monitoring.

The rest of the reviewed papers (i.e., 26.7%) report systems whose components interact synchronously and do not need a particular protocol for such interaction. These are based on traditional communication patterns such as RPC or API calls. Some of these systems are small, self-contained, and attend users that require synchronous responses to their requests [Hawes et al., 2017, Cabanes et al., 2019, Bayerl et al., 2020, Kemsaram et al., 2020, Johny and Madhusoodanan, 2021, Hegemier and Kelley, 2021, Falcao et al., 2021]. Hegemier and Kelley [2021] propose a real-time danger avoidance systems for robots. The system deploys a neural network in an edge server, which receives prediction requests from robots in POST (HTTP) format. Requests include an image of the robot environment and the robot status information (e.g., damages report). The edge server responds synchronously with a classification of the environment image. Similarly, the system proposed by Johny and Madhusoodanan [2021] receives diagnosis requests from users that send histopathological images for metastasis detection. A
deep learning model deployed in a Raspberry Pi responds with the image classification. Users send their requests (i.e., images) to an embedded web server installed in the Pi. Components of larger systems also interact synchronously. These are usually deployed in the cloud and follow microservices architectures. They adopt layered and modular architectures that satisfy scalability and availability requirements [Ali et al., 2016, Gao et al., 2016, Brumbaugh et al., 2019, Amrollahi et al., 2020, Xu et al., 2018]. For example, the AIDEX [Amrollahi et al., 2020] system adopts a modular architecture to predict patients’ risk of developing sepsis. Each functionality of the system is encapsulated as a microservice. The system exposes microservices for preprocessing data, executing the prediction algorithm, storing the prediction outcomes, and visualizing outcomes. These components are orchestrated using API calls to offer end-to-end capabilities such as sepsis prediction.

### 3.4.4 Summary

This section summarises the survey findings and provides practical advice for practitioners towards deploying Data-Oriented ML-based systems in the real world. We found that several ML-based systems follow all or most of the DOA principles [Zhang et al., 2016, Jiang et al., 2017, Lebofsky et al., 2019, Dai et al., 2019, Karageorgou et al., 2020, Zhang et al., 2021, Shan et al., 2022, Herrero and Zorrilla, 2022]. DOA principles (Section 3.2) support these systems to satisfy requirements that are becoming increasingly common for data-driven systems deployed in the real world. These systems address users’ data requirements that demand managing big data from distributed and intensive sources, low latency processing tasks, and efficient management of storage and computing resources. We observed that systems based on architectural patterns such as dataflow and publish/subscribe are more data-oriented and follow more of the DOA principles. These systems are designed in terms of data exchange between components (i.e., data first), which does not assume any coupling on the control flow level. **Practical advice:** Developers need to focus first on the data while creating data-intensive systems and treat operations as secondary. Dataflow and publish/subscribe architectural patterns are well suited to support the design of DOA systems.

Engineers adopt the **data as a first-class citizen** principle because it enables efficient interactions between systems components. This design decision avoids data transmission between components as payloads of direct calls (e.g., REST API calls). Systems’ components act as producers and consumers of data stored in **shared data models**. Such **data coupling** offers asynchronous interactions by design, which makes components autonomous and non-blocking for each other. **Data coupling** also addresses low latency requirements and resource constraints because autonomous components can process large data sets in parallel. The nature of the data systems handle influences the selection of the **shared data models** that systems components nurture. **Data coupling** based on databases is appropriate for systems that work with data that needs to be persisted in time, while streams fit better systems that handle continuous data.
from different and dynamic data sources. An engineer can design a system with one or more shared data models. This decision depends on how the data must be transformed from inputs to outputs along the system. **Practical advice:** Data coupling enables systems to address big data processing requirements in data-driven systems. Databases, streams, and message queues are examples of data mediums to consider when designing data-first systems. These data mediums play the role of shared data models and can be implemented using technologies such as Apache Kafka, Spark Streaming, HDF5, RabbitMQ, or HBase.

Our results show that centralised architectures are the preferred design choice when deploying ML-based systems to the detriment of the prioritise decentralisation principle. It shows the current prevalence of cloud platforms that offer flexible services and facilitate the deployment of systems in production. Cloud platforms cope well with the most relevant requirements that ML-based systems have nowadays as they provide flexibility, high availability, and scalability, and most of the time satisfy current low latency requirements. However, it is natural to expect these requirements to be more critical and severe as data becomes more available, users require processing in shorter times, and applications become more complex (e.g., digital twins, augmented reality applications, etc.). Further research of decentralised architectures is necessary to enable the benefits of decentralisation while also making it a feasible option for systems deployment. Reviewed works that implement Edge computing already leverage federated architectures with increasingly powerful edge servers that offer local data storage and processing. However, the peer-to-peer interaction between edge nodes is missing in most of the cases according to our survey. Cloud servers as the backend still play the main storage and processing roles. A major collaboration between nodes at lower layers in edge architectures has the potential to enable more sustainable systems by exploiting the storage and computing power of everyday devices. **Practical advice:** The absence of a central orchestrator in favour of direct communication between any two nodes of the system is a straightforward way to move to decentralisation. Together with distributed storage technologies (e.g., Apache Kafka, HDFS, etc.), DHTs and BFT are two distributed protocols that were used by the reviewed papers to implement decentralised solutions.

Systems that collect data from data-generating components, such as sensors, are usually open. New sensors and data sources can be added to these systems based on the interoperability that current communication technologies offer (e.g., Wi-Fi, Bluetooth, Zigbee, etc.). Architectures are more closed and static at upper levels where software components are predefined in most cases. These components are designed as static entities which usually communicate synchronously via RPC or REST calls (i.e., tight coupling). Data coupling and open environments have a strong correlation according to our survey. Reading from and writing to data mediums is an asynchronous communication process where systems’ components are modelled as data consumers and producers. This process requires components to utilise message exchange protocols that describe how and when to read and write data. Such protocols also
enable seamless, sustainable, and flexible architectures where components can join or leave at any time. **Practical advice:** The use of message exchange protocols and data coupling results in systems that are open and flexible by design. It enables data availability and horizontal scalability as resources are added on demand. Communication protocols such as MQTT and RabbitMQ are well-known tools on top of which open systems are built.

### 3.4.5 Threats to validity

In this section, we discuss threats to validity and limitations of our work.

a. **Research design validity.** A lot of ML deployments are not described in scientific literature. Sometimes they are presented as blog posts, but more often their details are not published anywhere. Besides, we dismissed some of the published reports because they omitted information about their software architecture from the paper. We addressed this threat by covering a wide range of fields and areas of ML applications.

b. **Publication selection validity.** We used a multi-stage selection process to find papers for the survey. While we have followed an established methodology, this approach has its validity threats. First, we may have missed search terms while implementing lookups in digital libraries. We have iteratively improved our search procedure multiple times to mitigate that risk. Second, the search functionalities are different between databases in our automatic search. We tried to include as many databases as possible even if they overlap. Our tool is publicly available and easy to extend to include new sources in the future. Finally, the automatic filters applied to select papers could exclude relevant ones. The number of retrieved papers made it necessary to automate the filtering process. We used state-of-the-art algorithms for the automatic filters and tested them under different configurations to get the best possible result.

c. **Survey validity.** We selected the surveyed papers based on a well-established methodology for systematic literature. However, we also provide an in-depth analysis of the architectural design decisions of the reported systems and their relation to DOA principles. Therefore we classify our work as a survey on DOA adoption by real-life ML systems.

d. **Analysis validity.** The quality of analysis and conclusions of this paper hinges on the expertise of its authors and therefore can be prone to personal biases. To alleviate this risk we sought feedback on our work, presenting intermediate results of our study internally to our research group and at external scientific events.
3.5 Open questions

In the previous sections, we observed that while the DOA principles offer the desired properties that enable systems to achieve demanding requirements at deployment, their adoption rates vary greatly. More research efforts are needed to advance the community’s understanding of the DOA paradigm, its strengths and weaknesses, why and how to build DOA systems, and the advantages of the capabilities DOA offers. These research directions are discussed below.

As shown in the previous section, practitioners utilise a great variety of tools, frameworks and services while building data-driven systems. For example, databases, streams, message queues and file systems were all used as data communication mediums. At the same time, there is no single reference to which engineers could refer while deciding on which technology to use in their system. Other aspects of DOA, such as communication protocols, short and long-term storage solutions, and monitoring, are in a similar state. Therefore DOA can benefit from having a technical stack taxonomy so that developers could have a complete list of necessary abstractions in a DOA system, and a list of tools to choose for each abstraction (or their combination) together with the trade-offs involved with each tool. Such taxonomies already exist for other areas of engineering practice, such as MLOps or microservices [Garriga, 2018].

A question that is closely related to the choice of tools and frameworks is the operational maintenance of DOA systems. Over the years the software engineering community accumulated a vast amount of knowledge on how to run SOA systems in production: what metrics to monitor, how to scale horizontally and vertically, how to mitigate and troubleshoot performance issues [Lewis and Smith, 2008, Singh and Tyagi, 2015, Beyer et al., 2016, Delac et al., 2012]. A similar knowledge base about DOA systems is necessary and can be developed with accumulated experience of running DOA software in production. Nevertheless, careful study of various configurations of DOA setups, their performance and critical metrics can also be carried out in an academic environment.

The DOA paradigm advocates for open systems where entities are autonomous and can freely access shared data models [Miao et al., 2019, Wei et al., 2021, Pennekamp et al., 2019]. These data models store systems’ current and past states, which naturally raises questions regarding systems’ security and privacy [Tsai et al., 2021]. Malicious entities can access and modify systems’ data and behaviour at any time in such open environments. Depending on the application considered, data access and systems’ components may need to be restricted to a specific set of users. For example, a healthcare management system needs to implement restrictive data access policies to avoid data privacy issues. The decentralisation principle can mitigate the security and privacy threats by storing and processing data in devices closer to end users (e.g., smartphones) [Shi et al., 2016, Tabatabaee Malazi et al., 2022, Cabrera et al., 2022]. However, managing authentication, permissions, and encryption keys in such a setup is

---

12AI Infrastructure Alliance project: https://ai-infrastructure.org/
challenging.

Different research efforts from the security community can be applied to DOA open setups to address security and privacy challenges. Homomorphic encryption [Fontaine and Galand, 2007] is an interesting direction for performing decentralised computations directly on encrypted data, without needing to provide the decryption key to the participant nodes. For example, a payment system might inquire about the validity of a transaction without having access to the underlying data (e.g. bank account number). Early deployment of zero-knowledge proof [Fiege et al., 1987] and homomorphic encryption are taking place in the industry [Blum et al., 2019]. This technology offers a solution to privacy issues in decentralized networks, but the field is still in its infancy. Algorithms developed are often computationally expensive, so further research is needed to make them more practical in resource-constrained devices. In addition to these technical advances, security and privacy issues also require authorities to develop novel initiatives and policy frameworks to keep up with advances in technology [Montgomery and Lawrence, 2021].

### 3.6 Related Work

While DOA as a software paradigm for ML applications is an emerging pattern, the principles behind DOA are not new. Many software engineering industries have already discovered these principles and are reaping the benefits of applying them in practice. Namely, data-oriented design (DOD) applies many DOA principles on a lower level of abstraction, with claims of significant improvements over analogous OOP solutions. Video game development is one industry where DOD is particularly widespread, as it is used to improve memory and cache utilisation [Acton, 2014]. For example, Coherent Labs utilized DOD while creating their game engine Hummingbird [Nikolov, 2018] as a way to overcome the performance limitations of engines based on Chromium and WebKit. DOD helped the developers eliminate a considerable amount of cache misses and compiler branch mispredictions, leading to a 6.12x improvement in animation rendering speed. Outside of gaming, Mironov et al. utilized DOD to improve the performance of a trading strategy backtesting utility and observed a 66% speed increase as well as improved opportunities for parallelization [Mironov et al., 2021]. These examples illustrate that DOA-like principles, especially around data prioritisation, emerge at a different level of abstraction with different motivations, and nevertheless bring significant benefit to software developers.

A wide range of papers have surveyed the application of AI in the last few years. They are usually focused on the AI techniques and algorithms that have been applied to solve problems in specific domains. They report the challenges that these algorithms face and the open research gaps that require the development of novel AI methods. For example, Cai et al. [2019] provide a survey of multimodal data-driven techniques applied to the domain of smart healthcare.
They review systems for disease analysis, triage, diagnosis, and treatment. Bohg et al. [2013] survey data-driven methodologies for robot grasping. They focus on techniques based on object recognition and pose estimation for known, familiar, and unknown objects. The review includes a comparison between data-driven methodologies and analytics approaches. Qin [2012] review the state-of-the-art of data-driven methods for industrial fault detection and diagnosis. The focus of their survey is on fault detectability and identifiability methods for industrial processes with different complexity and at different scales. Wong et al. [2020] present the challenges of applying deep learning (DL) techniques in radio frequency applications. This work reviews DL applications in the radio frequency domain from the perspective of data trust, security, and hardware/software issues for DL deployment in real-world wireless communication applications. Joshi et al. [2022] surveys the deployment of deep learning approaches at the edge. Their review presents architectures of deep learning models, enabling technologies, and adaptation techniques. Their work also describes different metrics for deep learning models at the edge which can be used to design and evaluate DL techniques.

Recent survey papers have also focused on the AutoML domain, which focuses on the automatic selection, composition, and parametrisation of learning models [Waring et al., 2020]. Faes et al. [2019] reviewed and evaluated automated DL software and tools to develop medical image diagnostic classifiers by healthcare professionals. They found that these professionals can use AutoML software to develop DL algorithms whose performance is comparable to the ones applied in the existing literature. Waring et al. [2020] review the state-of-the-art of automatic machine learning from a computer science and biomedical perspective. This paper shows that automated techniques can support experts in different ML tasks by reducing processing times. Escalante [2020] describes the main paradigms of AutoML in the context of supervised learning. This paper surveys different research works in this area and outlines future research opportunities. Similarly, Zheng et al. [2023] review the application of AutoML techniques to the domain of recommender systems. They propose a reference architecture for a recommender system and analyse the state of the art according to the architecture’s components.

### 3.7 Summary and Conclusions

In this chapter, we discussed data-oriented architecture (DOA) - a high-level paradigm for building data processing software. We identified guiding principles of DOA – data as a first class citizen, openness, decentralisation – and explained how they can support software designers and developers to address the challenges that emerge from the deployment of ML algorithms as part of larger systems. We analyse to what extent existing real-world ML-based systems have adopted these principles. We have identified a few works that have fully adopted the data-oriented principles to address requirements such as big data handling, efficient real-
time processing, and resilience. Most of the reviewed systems partially adopt the principles, and we discuss patterns and tools used for adoption. We also distilled recommendations for practitioners wishing to build their systems following DOA.

We observe that while DOA is not a widespread paradigm yet, it offers a range of properties desirable for the deployment of data-driven solutions. This fact opens a wide research agenda towards developing a better understanding of DOA, its interplay with other paradigms, and its suitability for various ML deployment scenarios. DOA offers opportunities to develop new practices around monitoring systems with networks of statistical emulators [Damianou and Lawrence, 2013], thus moving beyond existing work that only uses emulation for auto-tuning of system parameters [Alabed and Yoneki, 2022, Dalibard et al., 2017]. Furthermore, an ability to emulate individual components can lead to efficient end-to-end optimisation of DOA systems [Zeng et al., 2016, Aglietti et al., 2020]. This is particularly pertinent in cases where the system has to satisfy multiple competing requirements [Avent et al., 2020], in which case multi-objective Bayesian optimisation techniques can be applicable [Paleyes et al., 2022b]. DOA can accelerate advances in the area of Edge Computing [Cabrera et al., 2022, Shi et al., 2016, Tabatabaee Malazi et al., 2022], Federated Learning [Bonawitz et al., 2019] and self-adaptive systems [Cabrera and Clarke, 2019, Gerasimou et al., 2019], all of which stand to benefit from shared data model as well as access to data history and context. The ability to discover and inspect explicit flows of data within DOA systems has been highlighted as a key feature required for data governance [Akoush et al., 2022, Carata et al., 2014, Schwarzkopf et al., 2019, Singh et al., 2018], meaning DOA can play a vital role in developing AI systems that comply with existing and upcoming legislatures, such as GDPR, EU AI Act or Equal Credit Opportunity Act (ECOA). Similarly, access to a complete dataflow graph can improve our ability to use causal inference techniques for operational tasks [Paleyes and Lawrence, 2023, Paleyes et al., 2023a].

The field of Natural Language Processing (NLP) has been experiencing rapid growth lately, with an increasing focus on Large Language Models (LLMs). Models such as GPT-4 [OpenAI, 2023], Llama [Touvron et al., 2023] or Codex [Chen et al., 2021], have shown remarkable capabilities in understanding and generating human-like text, images or computer code, revolutionizing various NLP applications. These models are massive in size and require significant computational and storage resources, making software infrastructure a key determinant of their success. Given LLMs’ dependency on the quantity and quality of data, DOA is poised to bring benefits to building efficient and robust infrastructure for them. Understanding to which extent DOA principles are already used for building LLMs, as well as how current LLM infrastructure can be improved with DOA, is an exciting direction of research.

We have noticed that systems that were built with dataflow architecture [Dai et al., 2019, Jiang et al., 2017, Sultana et al., 2021] followed most of the principles we discussed in Section 3.2. This observation led us to recommend practitioners to consider this paradigm for future projects in section 3.4.4. In the following chapters, we will take a closer look at the dataflow architecture,
how it corresponds to the DOA principles we discussed, its utility for building ML systems, as well as aiding the deployment of ML in general.
Chapter 4

Introduction to dataflow architecture

In the previous chapter, we observed that dataflow architecture supports the process of building software that complies with the principles of DOA. We saw that the dataflow approach to systems engineering provides data coupling between components, is naturally decentralised and encourages the creation of autonomous components. These observations warrant a closer look at this classic paradigm. Specifically, we would like to understand to what extent dataflow architecture follows the DOA principles, and how it can be used to implement DOA systems in practice. In this chapter, we will look at the history and concepts of dataflow, before investigating its potential benefits for ML deployment later in the thesis.

4.1 Dataflow and von Neumann architectures

We begin with a brief history of dataflow and an overview of its concepts. The foundations of dataflow architecture were laid in the 1970s [Arvind and Gostelow, 1978, Dennis and Misunas, 1974] as an attempt to come up with a paradigm that is more efficient for parallel processing than the von Neumann architecture [Von Neumann, 1993].

In the von Neumann architecture, the program follows a predetermined set of instructions which are executed sequentially by a processing unit following the program’s control flow. This approach yields relatively simple programs: data and instructions can be accessed from one memory unit and in the same manner, sequential programs are predictable and easy to understand. It was used to develop many early computers, such as EDSAC [Wilkes and Renwick, 1950] or UNIVAC [Eckert Jr et al., 1951]. As the demand for computing among scientists and the general public grew, von Neumann architecture gained popularity. The vast majority of modern computers are based on this architecture, using caching mechanisms so that most instruction and data fetches use separate buses. Nevertheless, achieving efficient parallel execution of code in von Neumann architecture is difficult [Buehrer and Ekanadham, 1987, Peláez, 1990]. Programs within this architecture are inherently sequential as instructions can only be executed one at a time, an effect known as “von Neumann bottleneck”. Any parallel processing requires
a lot of overhead of managing shared memory between separate processes, which requires additional effort and is prone to errors.

This inability to support efficient parallelism led to the invention of dataflow architecture. In dataflow, the processing is driven by the availability of data, rather than the order of instructions. The system consists of a set of processing nodes or components, which are connected by communication links, and a node is executed as soon as all its inputs are available. The system can thus be perceived as a directed graph, where vertices are components and edges are communication links. The graph is often assumed to be acyclic (commonly referred to as DAG – directed acyclic graph), although extensions for cyclic graphs exist [Murray et al., 2013]. The separation of data and instructions in the dataflow paradigm allows them to be naturally stored in separate memory units. Moreover, dataflow does not prescribe a specific order in which processing is done, again naturally fitting the concept of multiple computations running asynchronously\(^1\). Since its inception, the dataflow paradigm proliferated to various areas of computer science, including hardware architecture [Veen, 1986], concurrent process networks [Kahn, 1974], programming languages [Johnston et al., 2004]. The particular area we are focusing on in this thesis is dataflow as a software architecture pattern. This means our focus is on the organisation of software components and the way they communicate data with each other, without diving into the levels of language operators or underlying hardware.

### 4.2 Modern adoption of dataflow

Multiple well-known software systems were developed following dataflow principles. A well-known example of a dataflow software system is MapReduce, developed in 2008 for handling Big Data [Dean and Ghemawat, 2008]. MapReduce is a simple and scalable programming model for data processing, in which input data is passed through map operators (usually doing filtering, sorting, or transformations) followed by reduce operators that perform some sort of aggregation before producing the final output. This execution model can be represented as a dataflow graph, as shown in Figure 4.1.

Since then, multiple dataflow systems have been created by the engineering community. Apache Flink is a distributed processing engine for stateful computations over unbounded and bounded data streams that highlights the connection between dataflow and stream processing [Carbone et al., 2015]. Noria is a system that uses a partially stateful dataflow model to offer a scalable backend for high-performance web applications. Closer to the topic of this thesis, several popular machine learning tools are implemented with a dataflow approach. Notable examples include TensorFlow [Abadi et al., 2016], a general-purpose library with a focus on

\(^1\)We can observe that on one hand, the majority of computers are built with von Neumann approach, and the community developed a wealth of knowledge about this paradigm. On the other hand, there is a growing demand for parallel computing, and dataflow is better positioned to support it. This dissonance led to the development of hybrid dataflow/von Neumann architectures [Iannucci, 1988, Yazdanpanah et al., 2013, Bhagyanath et al., 2023].
neural networks, Neuflow [Farabet et al., 2011], a hardware architecture optimized for the computation of general-purpose vision algorithms, RLFlow [Liang et al., 2020], a reinforcement learning library, and more. More recently, Pathways was announced in 2021 as Google’s new asynchronous distributed dataflow platform for orchestrating ML training [Barham et al., 2022], and was used to produce large ML models such as PaLM [Chowdhery et al., 2022] and Minerva [Lewkowycz et al., 2022].

Furthermore, there is a growing body of literature exploring various aspects of dataflow systems, such as rollback recovery [Gog et al., 2021], performance profiling [Beischl et al., 2021], cyclic graphs [Murray et al., 2013], iterative algorithms [Gévay et al., 2021], strictness analysis [Schrijvers and Mycroft, 2010]. Furthermore, dataflow approaches are being actively applied to autonomous driving [Gog et al., 2022], astrodynamics simulation [Kenneally et al., 2020], legal compliance [Schwarzkoopf et al., 2019], Internet worms containment [Costa et al., 2005], and other practical areas. Community has also explored the duality of control and data flow in software systems [Treleaven, 1982, Lauer and Needham, 1979, Crowcroft and Deegan, 2005, Hasselbring et al., 2021].

In this thesis, we will demonstrate the benefits of dataflow architecture for building ML-based systems. Our work aims to explain the empirical success dataflow already found in
the ML community, and to showcase potential for even greater benefits this paradigm offers. Specifically, we will show how it follows all DOA principles discussed in Chapter 3, and how it can improve observability, lineage, troubleshooting and experimentation in software engineering for AI.

4.3 Flow-based programming

A particularly popular flavour of dataflow is flow-based programming (FBP). Compared to a general dataflow architecture, FBP offers additional formalism for system components: information packets with defined lifetimes, named ports, and data connections. In the later chapters we will be using FBP-inspired tools and frameworks, such as flowpipe\(^2\), SciPipe\(^3\) and Node-RED\(^4\), therefore it is worth introducing this paradigm to the reader.

Flow-based programming was introduced by J. P. Morrison in the 1970s for distributed processing [Morrison, 2010]. FBP defines a software system as a set of isolated processes that pass data between each other via connections that are external to those processes. Each process exposes data interfaces, known as named ports, that define inputs and outputs for that process. FBP exhibits “data coupling”, which is considered in computing to be the loosest form of coupling between components [Offutt et al., 1993, Olsson, 2014], thus promoting a flexible software architecture. One of the key features of FBP-driven system design is that by defining data processing components and connections between them developers naturally build a dataflow graph of the entire software system, such as the one shown in Figure 4.2.

From this short introduction, we can already see how well FBP abstractions correspond to DOA principles. The fact that data communication channels are external to the processes they connect increases the visibility and priority of the data. Dataflow graph models computations of the system as a decentralised network. Finally, the fact that processing nodes are asynchronous

\(^2\)https://github.com/PaulSchweizer/flowpipe
\(^3\)https://scipipe.org/
\(^4\)https://nodered.org/
and communicate via information packets follows the definition of the openness principle.

Flow-based programming has been found to optimise speed, bandwidth and processing power in multi-tasking, parallel processing applications. For example Szydlo et al. [2017] consider FBP’s application to IoT, Lampa et al. [2016] explore FBP’s potential in the context of drug discovery, Zaman et al. [2015] present an FBP-based framework for mobile development, Mahapatra and Banoo [2022] use FBP to develop ML training pipelines. Recent years saw the birth of several general-purpose projects built with FBP, such as NoFlo [Bergius, 2015] and Node-RED\(^5\). Node-RED in particular became popular in the IoT community [Clerissi et al., 2018, Chaczko and Braun, 2017], as the FBP model was found to be a good fit for building data processing pipelines in IoT. Nevertheless, the FBP paradigm remains relatively niche [Sibirov, 2022], and in the later chapters, we show why we believe it deserves more attention from the software engineering community.

### 4.4 Data streaming

In addition to the introduction to dataflow architecture, we shall give the reader a brief overview of data streaming, a technology that is often used as a data communication channel in dataflow systems. We will often refer to and make use of streaming in the following chapters.

Data streaming technology refers to a method of transmitting and processing data in a continuous and real-time manner. A data stream is an abstraction that represents an indefinite sequence of data records that are made available over time. It enables the efficient and near-instantaneous transfer of data from one point to another, allowing for the processing and analysis of data as it is generated or ingested. Streaming is often contrasted to traditional batch processing methods, where data is collected over a period of time and then processed all at once.

Machine learning on data streams is a well-established concept. Data processing platforms such as Apache Spark [Meng et al., 2016], Apache Flink [Carbone et al., 2015] or Google Cloud Dataflow [Krishnan and Gonzalez, 2015] are widely used for manipulating large datasets and executing machine learning tasks. AWS Kinesis\(^6\) and Apache Kafka [Kreps et al., 2011] are two of the most commonly used data streaming services. In the following chapters, we will be referring to streaming abstractions, as well as using Apache Kafka to implement a production-ready dataflow ML system.

\(^5\)https://nodered.org/

\(^6\)https://aws.amazon.com/kinesis/
4.5 Summary and Conclusions

In this chapter, we have introduced the dataflow architecture and its popular flavour flow-based programming (FBP). We discussed several important concepts and characteristics of these paradigms, such as named ports, dataflow graph and data coupling. As we shall see, these ideas play an important role in building ML-oriented systems. We also provided a brief overview of data streaming.

Dataflow architecture and FBP correspond well to the principles of DOA which, as we discussed before, is an emerging paradigm that has great potential to simplify the process of deploying ML in production. As later chapters show, this means dataflow can address or mitigate many of the challenges of ML deployment, particularly those related to data discovery and management. In the next chapter, we will start exploring the suitability of dataflow and FBP in the ML deployment context.
Chapter 5

An Empirical Evaluation of dataflow architecture

As we have already learned, when deploying machine learning (ML) algorithms in real-world systems, software developers face a new set of challenges [Paleyes et al., 2022c, Figalist et al., 2022]. In particular, real-world systems produce large quantities of heterogeneous, time-varying, high-dimensional data that feeds decision-making in these systems. This puts under question the effectiveness and efficiency of current software development and deployment practices for ML deployment. The challenges are present across the entire ML application workflow, including the stages of data engineering, model learning, model verification, and model deployment. For example, data analysts spend most of their time in looking for, acquiring, understanding, cleaning and preparing the data before using an ML algorithm [Nazabal et al., 2020]. These challenges arise because the ML is deployed on top of existing software solutions which were built to fulfil goals that are important but not directly related to ML, such as high availability, robustness, and low latency. Machine learning brings a new set of requirements that the majority of existing software architectures were not designed for [Lewis et al., 2021, Ozkaya, 2020].

Seeking a way to mitigate some of these challenges, in the previous chapter we discussed Data Oriented Architectures (DOA), an emerging paradigm that aims to facilitate the integration of data processing components within modern software systems [Joshi, 2007, Vorhemus and Schikuta, 2017, Lawrence, 2019]. DOA treats data in the system as a first class citizen in a shared information model, where stateless system components perform distributed processing. These components communicate between each other using an asynchronous message exchange protocol. Such features enable DOA to achieve high data discoverability, availability, and reuse. The stateless and loosely coupled system components also allow DOA to deal with large-scale dynamic environments [Vorhemus and Schikuta, 2017].

Despite a high level consensus about the potential benefits that DOA brings to the implementation and deployment of ML algorithms, there is no clarity on which tools, frameworks and
programming paradigms should be used as the building blocks of a DOA system in practice. We explored some of the practical approaches used in real-life solutions in chapter 3 and identified dataflow architecture as a pattern that complies with DOA principles. This chapter presents a quantitative evaluation of flow-based programming (FBP, Morrison [1994]), a particular flavour in the dataflow family of approaches, as a paradigm for building DOA-based applications. FBP defines applications as networks of "black box" processes, which communicate via data streaming connections, where the connections are specified externally to the processes. Through external connections and named ports FBP promotes data coupling between system components. We evaluate to what extent FBP is suitable for the development of ML-based applications at different stages of the ML deployment workflow. As a baseline for comparison in our evaluation, we use the currently prevalent Service Oriented Architecture (SOA) [Perrey and Lycett, 2003, O'Reilly, 2020], the shortcomings of which we discussed in the chapter 3.

As a form of dataflow, FBP was contrasted with other paradigms and design principles, although never in the context of ML. In his book “Flow-Based Programming: A New Approach to Application Development” the creator of FBP, J.P. Morrison, compares FBP and Object-Oriented Programming (OOP) [Morrison, 2010]. Similarly, Roosta [2012] contrasted dataflow and functional programming paradigms in his book “Parallel Processing and Parallel Algorithms: theory and computation”. On a more practical side, Mironov et al. [2021] compared OOP and data-oriented approaches in the context of back-testing for trading strategies. They found the data-oriented approach is more efficient because of parallelism mechanisms. Lobunets and Krylovskiy [2014] implemented the same IoT processing system with SOA and FBP and reported their experiences regarding code coupling, debugging and testing. While our methodology is similar to this work, our evaluation focuses on data and ML.

Our evaluation follows the Goal-Question-Metric (GQM) methodology for experimentation in software engineering proposed by Wohlin et al. [2012]. We first use the GQM framework to define the goals of the evaluation, their respective questions, and the evaluation metrics. We next describe and develop four data processing applications set in different domains and formulate a business problem for each application that can be solved with ML. We then carry out the complete workflow of integrating an ML solution on top of each application, collecting previously defined metrics to the code base at each stage of the deployment. Finally, we compare the fitness of FBP and SOA paradigms for ML deployment based on the measurements taken. Results show that FBP can address some of the key challenges around ML deployment by exposing the data in the system. At the same time, there are still some gaps that remain before it can be considered a go-to paradigm for DOA.
Table 5.1: Goals, questions and metrics of our experiment for comparing FBP and SOA. We are following the methodology suggested by the GQM framework [Wohlin et al., 2012].

<table>
<thead>
<tr>
<th>Goals</th>
<th>Questions</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal 1: Analysing software paradigms for the purpose of evaluating their impact in the software development process with respect to <strong>data collection tasks</strong> from the point of view of software developers in the context of the development of data processing applications.</td>
<td>Question 1: How much additional code is required to implement a data collection task in the evaluated applications for each paradigm?</td>
<td>Logical lines of code</td>
</tr>
<tr>
<td></td>
<td>Question 2: How does the system’s maintainability change when data collection tasks are implemented in the evaluated application for each paradigm?</td>
<td>Maintainability Index</td>
</tr>
<tr>
<td></td>
<td>Question 3: How complex does the evaluated application become after implementation of data collection task for each paradigm?</td>
<td>Cognitive Complexity</td>
</tr>
<tr>
<td></td>
<td>Question 4: How intrusive is the dataset collection task in the evaluated applications for each paradigm?</td>
<td>Number of affected components</td>
</tr>
<tr>
<td>Goal 2: Analysing software paradigms for the purpose of evaluating their impact in the software development process with respect to <strong>ML model integration</strong> from the point of view of software developers in the context of the development of data processing applications.</td>
<td>Question 1: How much additional code is required to implement an integration task in the evaluated applications for each paradigm?</td>
<td>Logical lines of code</td>
</tr>
<tr>
<td></td>
<td>Question 2: How does the system’s maintainability change when model integration task is implemented in the evaluated applications for each paradigm?</td>
<td>Maintainability Index</td>
</tr>
<tr>
<td></td>
<td>Question 3: How complex does the evaluated application become after integration of ML model for each paradigm?</td>
<td>Cognitive Complexity</td>
</tr>
<tr>
<td></td>
<td>Question 4: How intrusive is model integration task in the evaluated applications for each paradigm?</td>
<td>Number of affected components</td>
</tr>
</tbody>
</table>
5.1 Experiment design and implementation

Our evaluation of the DOA-based applications follows the methodology proposed by Wohlin et al. [2012]. We first define a set of metrics following the GQM framework, and then develop four applications that cover a variety of business domains and areas of ML. These applications allow us to study the properties of FBP and compare it against the classical SOA approach in the experimental setting. In this section we describe the metrics we used, the applications we implemented, and give details of the experiment we defined to examine the fitness of both paradigms in the ML deployment context.

5.1.1 Metrics Definition

We follow the GQM framework to define the metrics for evaluation [Wohlin et al., 2012]. This framework proposes to define the evaluation goals as a first step. These goals are then mapped to metrics through questions required to achieve the goals. Table 5.1 introduces the goals of this evaluation, the questions we want to answer to achieve these goals, and the metrics that will allow us to provide such answers. The resulting metrics are defined as follows:

- **Logical Lines of Code**, which counts executable statements and ignores comments and blank lines. We use it as a measure of an application’s codebase size. Specifically, we assess how much additional code is needed to implement additional functionalities in the applications.

- **Maintainability Index**, as defined in the Radon package\(^2\), is a composite metric that is calculated using several other metrics as operands. This is a unitless metric that assigns a codebase a score between 0 and 100. We use it to assess how maintainable is the application’s codebase.

- **Cognitive complexity** [Campbell, 2018], which measures the complexity of the control flow of code. We use it to assess how easy it is to understand the application code. This metric is similar to McCabe’s cyclomatic complexity [McCabe, 1976], and was proposed as a replacement that focuses on human’s understanding of the source code, which is critical for software development and maintenance. Since cognitive complexity is measured separately for each code block, we consider average cognitive complexity across the codebase.

- **Number of Affected Components**, which counts the number of components that were added or changed during a certain stage of development. It allows us to evaluate the

\(^1\)Full source code of the project can be found at [https://github.com/mlatcl/fbp-vs-soa](https://github.com/mlatcl/fbp-vs-soa)

intrusiveness of a particular feature, that is how many parts of the codebase had to be changed or added for implementation of that feature. For the purposes of this experiment, we identify processing nodes and data streams as components of an FBP program, and APIs and data access routines as components of an SOA program.

5.1.2 Applications

We have implemented four applications separately with FBP and SOA paradigms. For each application, we formulated a business problem that is a typical task for practising data scientists. We then carried out the deployment of a data-powered solution to the task while observing the evolution of the codebase throughout the deployment cycle. In this section, we give a brief description of each application as well as the deployment stages.

**Ride Allocation** application maps incoming ride requests to available drivers and tracks history of existing rides [Paleyes et al., 2021]. We have formulated a task of estimating pickup wait time for each ride allocation based on historical data. This task is approached as a supervised learning regression problem with offline model training of the collected dataset.

**MBlogger** is a micro-blogging platform that keeps track of users, their following/follower relationships between each other, and builds a timeline of posts for each user based on the activity of those this user follows. As a task, we decided to build a post-generating bot that, given a particular user, can generate posts this user is likely to be interested in. The solution can be considered a simple generative NLP approach. In MBlogger we do not collect offline dataset file, instead storing all data that is needed to generate posts on the fly using runtime infrastructure.

**Insurance Claims** application models a workflow that processes car insurance claims. Claims undergo a series of classification routines which affect the choice of the final payout process. In this application, we use ML to replace all internal logic with one classification model. As is the case with Ride Allocation, this model is trained offline and then used for online inference. Additionally, this application is different from the ones described above because albeit the ML model is being deployed, it does not affect the user interface and only changes the internal data processing mechanism of the application.

**Playlist Builder** creates a movie playlist for a specified genre. Initial functionality builds playlists at random, and later stages only add the highest-grossing movies to a playlist. Unlike all other applications, here we do not collect raw data or train any models. Instead, quantiles of movies’ gross earnings are calculated and then used for filtering online. Playlist Builder illustrates a simple yet realistic use case where a data scientist needs to build a solution that collects certain statistics about the data and then makes automated decisions based on it.

---

3This application closely follows Metaflow tutorial: https://docs.metaflow.org/getting-started/tutorials/season-1-the-local-experience/episode01
Table 5.2: List of all created versions created for each application. The first column gives the key by which a particular version is referred to in the codebase and the chapter.

<table>
<thead>
<tr>
<th>Key</th>
<th>Paradigm</th>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb_app_min</td>
<td>FBP</td>
<td>1</td>
<td>Basic functionality</td>
</tr>
<tr>
<td>fb_app_data</td>
<td>FBP</td>
<td>2</td>
<td>Same as fb_app_min plus data collection</td>
</tr>
<tr>
<td>fb_app_ml</td>
<td>FBP</td>
<td>3</td>
<td>Same as fb_app_data plus model integration</td>
</tr>
<tr>
<td>soa_app_min</td>
<td>SOA</td>
<td>1</td>
<td>Basic functionality</td>
</tr>
<tr>
<td>soa_app_data</td>
<td>SOA</td>
<td>2</td>
<td>Same as soa_app_min plus dataset collection</td>
</tr>
<tr>
<td>soa_app_ml</td>
<td>SOA</td>
<td>3</td>
<td>Same as soa_app_data plus model integration</td>
</tr>
</tbody>
</table>

We defined three stages of the implementation of each application to evaluate codebase changes:

- **Stage 1**: minimal code to provide basic functionality without any ML-powered capabilities. The stage is denoted in the code and this chapter by suffix *min*.

- **Stage 2**: code for Stage 1 plus implementation of data collection. Denoted by suffix *data*.

- **Stage 3**: code for Stage 2 plus code necessary for integrating the data-driven solution. If the ML model is created, it is trained on the dataset collected at the previous stage. Denoted by suffix *ml*.

In total, our experiment contains six implementations of each of the applications described above. Table 5.2 summarises these versions, and Figure 5.1 illustrates all six implementations for one of the applications (Insurance Claims) with sequence (for SOA) and data flow (for FBP) diagrams.

### 5.1.3 Implementation notes

Each application in the experiment is structured the same way: the application itself and the code that simulates events happening in the outside world. The application part processes the incoming data according to some business logic, and outputs data that is then used by the simulation. The simulation part is implemented as a discrete-event simulation and is responsible for generating events that would happen if the system was deployed in real life.

The SOA version of each application is done in a form of a RESTful service. We use the lightweight Flask\(^4\) framework to develop the SOA applications because of its flexibility and popularity among the Python community. Microservices persist and manipulate data using SQLite\(^5\). This popular database engine offers a small, fast, self-contained, high-reliability, high-performance database.

\(^5\)SQLite - https://www.sqlite.org/index.html
full-featured platform for data management. We implement a set of services that offer the required capabilities for each application and implementation stage, and a data access layer that manages database queries. All services are hosted locally, and the communication is happening via HTTP requests.

The FBP version of each application is built with two major building blocks: data streams and stateless processing nodes. We use the lightweight Python “FBP-inspired” framework flowpipe. Each data stream within the application belongs to one of three categories:

- Input streams, that receive data from the outside world;
- Output streams, that hold data produced by the application;
- Internal streams, that hold intermediate data within the application. These streams are necessary because processing nodes by definition are not allowed to have a state.

A processing node takes one or more streams as input, performs some operations on them, and then puts the result into output or internal data streams. Such nodes do not carry any internal state and do not make external calls to outside services or databases. All data influencing a processing node should be registered in the system, so if such additional input is necessary, it should be represented as a data stream.

Figure 5.1 shows diagrams that describe all 6 versions of one app. As can be seen in these diagrams, the entry point for all versions is an App object, that either orchestrates calls to necessary services (SOA) or triggers an evaluation of the data flow graph (FBP). Within each paradigm, subsequent stages introduce new APIs or graph nodes, all of which are highlighted for clarity.

Since applications are implemented using Python frameworks, we used Python tools Radon and Flake8 for metrics collection.

5.2 Evaluation

In this section we present the results of the experiment. We follow the goals and questions formulated in Table 5.1 and answer each question by analysing the corresponding metric.

5.2.1 Data collection task

First we address the questions around the data collection task, by measuring the changes between min and data stages.

---

6flowpipe - https://github.com/PaulSchweizer/flowpipe
Figure 5.1: Diagrams of Insurance Claims app evolution through three stages of ML deployment for both paradigms. We use sequence diagrams for SOA (left) and data flow diagrams for FBP (right). Yellow boxes in SOA diagrams represent individual services. Circles in FBP diagrams represent processing nodes, and boxes represent data streams: red for input, yellow for internal, and green for output streams. Deployment stages top to bottom are: min, data, ml. New or updated components and APIs at each subsequent stage are highlighted with bold red dashed lines.
Figure 5.2: Measurements of the impact data collection task had on FBP and SOA implementations of four applications. While changes to the SOA-based applications are smaller in size (as there is no need to implement graph traversal logic), their FBP counterparts are more localised and easier to comprehend.

5.2.1.1 How much additional code is required to implement a data collection task?

We used the Logical Lines of Code (LLOC) metric to answer this question, thus measuring how much code, in percentage of the initial size of the codebase, was added for the data collection stage. We use percentage change to make this comparison independent of the amount of boilerplate code that might be different for different paradigms. As can be seen from Figure 5.2a, two out of four FBP and SOA applications required a comparable amount of additional code to implement data collection. In both cases the difference is small: the growth difference was less than 0.2% for the Insurance Claims codebase and 4% for the Ride Allocation codebase. MBlogger and Playlist Builder show a different behaviour, suggesting that setting up additional runtime infrastructure for data may bear a higher initial cost for FBP⁹. Overall SOA exhibits the same or slower growth of the codebase for implementation of data collection.

⁹As a reminder, MBlogger and Playlist Builder do not create an offline dataset file to accomplish the data collection task.
5.2.1.2 How does the system’s maintainability change when data collection tasks are implemented?

We used the Maintainability Index (MI) metric to answer this question. MI is a score from 0 to 100 and does not scale with the size of the codebase, which allows us to compare its absolute values. For three out of four applications, maintainability was not dramatically impacted in either paradigm, dropping a similar amount of score points (between 2 and 5). But importantly, the MI score has consistently decreased more for the FBP codebase than it did for the SOA codebase across all four applications, as can be seen in Figure 5.2b. The reason for that is the fact that to collect data in FBP we needed to implement a relatively complex dataflow graph traversal logic, which had a great impact on the MI metric. A tool designed to collect data from a dataflow graph that encapsulates such traversal operations would improve developer experience.

5.2.1.3 How complex does the application become after implementation of the data collection task?

This question is answered with the help of the cognitive complexity metric. To make this comparison independent of the size of the codebase, we measure the complexity of each block of code independently, and average these measurements across the whole codebase. Results can be found in Figure 5.2c. Because of its focus on modelling how the data flows through the system, FBP code shows lower average complexity across all four applications, and on three occasions the cognitive complexity is at least twice as low. This suggests that the overall system is simpler to comprehend when it models dataflow, and thus might be easier to work with in the long term.

5.2.1.4 How intrusive is the data collection task?

We counted the number of components that were affected to implement the data collection task. FBP required fewer changes for all applications (Figure 5.2d). Only one component had to change for two out of three FBP codebases that collected the offline dataset. The lower impact of FBP in this metric can be explained by the ability of FBP programs to traverse the dataflow graph, which allows for more localized changes in the system when it comes to data collection.

5.2.2 ML model integration task

We now address the same questions in the context of ML model integration, by comparing measurements of data and ml stages.
5.2.2.1 How much additional code is required to implement the model integration task?

FBP codebase grew relatively slower at the model integration stage for three applications: the growth difference is 15% for Insurance Claims, 5.3% for MBlogger and 10% for Playlist Builder, as can be seen in Figure 5.3a. The difference between the paradigms is more pronounced than at the previous stage, which is a benefit of loose data coupling. We conclude that FBP requires the same or less effort to host a trained model than SOA.

5.2.2.2 How does the system’s maintainability change when the model integration task is implemented?

In terms of maintainability, FBP exhibited similar behaviour at this stage compared to data collection, dropping between 1 and 4 points, see Figure 5.3b. Interestingly, SOA exhibited a different behaviour, with a much more severe impact for Insurance Claims and MBlogger: MI of these applications decreased by 4 and 5 points, compared to 2 points for data collection.
The maintainability of these applications is more impacted because they need more code (Figure 5.3a) at this stage. FBP shows more maintainable way of adding a component that hosts an ML model.

5.2.2.3 How complex does the application become after implementation of the ML model integration task?

Both paradigms allow hosting trained models or other data-driven components in a way that is loosely coupled with the rest of the system. Therefore the picture around cognitive complexity does not change significantly when compared to the data stage - FBP solutions are still easier to comprehend, and deployment of data-driven components does not affect the complexity of the overall solution in either paradigm.

5.2.2.4 How intrusive is the model integration task?

As was the case with the data collection task, FBP again required changes in fewer components in all four cases. But the underlying reason for this is different. Recall that in the data collection stage, FBP required a dataflow graph traversal, which on two occasions could be implemented from a single component of the application. For model hosting no traversal is required. However, the model needs to be given the data to make predictions. In the case of FBP that only means additional wiring within the graph, to make sure the model node is connected to necessary data streams. In SOA that means changes that have to be made both on the service and data access layer, to ensure the right services are invoked and the right data is fetched for the model. All in all, FBP allows for less intrusive changes for both stages of the deployment.

5.2.3 Discussion of the evaluation results

In this chapter, we evaluated flow-based programming (FBP) as a way to build software that follows DOA principles. In this section, we discuss the results of our evaluation and compare the experiment outcome against these expectations.

SOA-based applications showed smaller size growth in the data collection phase. In contrast, the FBP codebases often has to deal with traversing the dataflow graph to collect an offline dataset from multiple sources, which can be a complex operation to implement. Additionally, FBP required more boilerplate code to add new components for those applications where data was collected online. All of these resulted in a bigger impact on FBP codebases in LLOC and Maintainability Index metrics. However, the size of the codebase is not the only factor when it comes to the maintainability of a system, and FBP excelled in other aspects of maintainability. Changes in FBP code are always more localised and have to be made in fewer parts of the system (according to the Number of Affected Components metric). A smaller number of components’ changes brings down the costs of maintenance and increases the robustness of the system over
time [Sommerville, 2011]. FBP code is also easier to comprehend, according to the Cognitive Complexity metric, which is again beneficial for maintenance. Quantification of these benefits of DOA systems implemented with FBP can be subject to a long-term observational follow-up study.

Interestingly, the FBP code showed the same or less impact by the model integration stage across all four metrics. This result might be surprising considering that SOA is usually chosen by professionals because of its loose coupling of components, and the extensibility and scalability benefits it brings. This kind of robustness to the deployment of a data-driven component is the direct consequence of data coupling that DOA promotes and FBP exhibits via the external data connections principle.

The evaluation highlighted the gaps that need to be covered to make FBP a default choice for building DOA software. We have kept the number of programming tools and frameworks to a minimum, aiming at comparison of the paradigms in their vanilla state. This also means our evaluation can highlight the areas where appropriate tools can make the biggest improvement for the overall development experience. For FBP this is the processing of the dataflow graph. Operations on that graph add a lot of complexity (e.g., node wiring and traversals). Targeted tools that provide such operations would simplify the development process with FBP and thus increase the adoption of this paradigm for data-driven applications.

5.3 Threats to Validity

5.3.1 Internal validity

We used a particular toolset for the development of all the applications described in this study. Specifically, we used Python for all our development, Flask for SOA implementations, and flowpipe for FBP implementations. The particular choice of tools might have affected the code metrics collected. We have compensated for this in several ways. First, we have kept the number of tools to a minimum, to ensure we evaluate the paradigms on their own, and implemented our own boilerplate whenever necessary (e.g. data stream abstraction for FBP, entity mapping for SOA). Second, we developed different types of applications from different domains following the same architectures and using the same tools. This variety allows us to counter the possible tools’ impact.

The code metrics we used might be affected by the particularities of the development process within a single paradigm. For instance, size-dependent metrics can produce larger values for paradigms that require more boilerplate code. We accounted for that by considering relative growth (LLOC) or average values (Cognitive Complexity), thus converting the metrics to size-independent ones.
5.3.2 External validity

The main threat to the external validity of our work is its environment. Ideally, the experiment described in this chapter shall be run in a real production setting and observed over multiple projects. However, that would require running two production-ready systems with equal functionality simultaneously. That would double the requirements on workforce and computational resources, which is prohibitively expensive to expect from a business. Additionally, FBP at the moment is a niche paradigm, rarely used outside of IoT and embedded systems, which would introduce a skew into such study, either positive or negative depending on the business domain. The main purpose of our work is a wider adoption of FBP and other dataflow-type approaches outside of these areas. For that reason we designed the experiment to model the real ML deployment process as closely as possible, separating jobs done by software engineers and data scientists, and considering a range of various data science tasks. In that sense, the scale of the experiment we chose is a trade-off between a realistic setup and clarity of our message. Nevertheless, the long-term benefits of adopting FBP as a primary design paradigm for data-driven projects are not fully explored yet and can be a subject of an extension industrial study.

The design of software systems has to account for other aspects, besides availability and collection of data. Modern distributed systems have to be fast, resilient, scalable, and secure. We left the evaluation of FBP in all these areas out of the scope of this chapter to keep our work focused on data science related tasks. However, all of these aspects are important and shall influence design choices made by software engineers while designing new systems. The extent of their influence depends on the experience of the engineering team, business requirements, available hosting infrastructure, and other project-specific factors. With this in mind, the next chapter integrates the ideas of dataflow architecture into a production system.

5.4 Summary and Conclusions

This chapter presents the results of an evaluation of FBP for ML deployment by comparing it against the widely used SOA paradigm. We implemented four data-driven applications in both paradigms, completed a data science task for each of them, and measured the evolution of the codebase throughout the process.

FBP is considered primarily a paradigm for distributed computing. Our experiment shows that FBP is a viable paradigm for developing general-purpose software according to DOA principles. We showed how FBP features (such as dataflow graphs and data coupling) make data discovery and collection easier than in a traditional SOA setting. In addition, we highlighted how better tooling, that allows developers to define and traverse dataflow graphs at a higher level of abstraction, could help improve FBP adoption.

We hope that this chapter has highlighted the potential for FBP, and the broader family of
dataflow architectures, for efficient implementation of DOA systems. Exploration of approaches that promote better treatment of data is crucial to address challenges in ML deployment, considering the growing demand to leverage data for scientific and business purposes. DOA promotes explicit handling of all data in the system, which makes data discovery available by design. FBP realises this principle with the dataflow graph. In the control flow approaches (e.g. SOA) data and data relationships are difficult to discover. This problem grows with scale, as data becomes scattered between multiple data storage facilities. Therefore the dataflow graph feature is a significant improvement in FBP applications compared to SOA. Nevertheless, SOA also offers important benefits to developers, for example, client-server communication and horizontal scalability. But the ability to build, access and traverse the dataflow graph can be achieved with hybrid approaches as well, for instance using FBP approach for the definition of the whole graph and microservices or serverless computing for implementation of individual components. Motivated by this idea and the need to provide better tooling for low-level operations in FBP systems, the next chapter will focus on building a dataflow-based platform for ML deployment. We will continue our exploration of the dataflow paradigm and describe a production-ready distributed system built on Apache Kafka that allows us to further evaluate the dataflow architecture as the means of building ML infrastructure.
Chapter 6
Dataflow for machine learning operations

In the previous chapter, we have made some observations about the potential utility of dataflow architecture, and specifically flow-based programming, for building data processing software. We have noticed that the dataflow approach improves discoverability and collection of data, which is critical for building machine learning (ML) applications. These observations were made on the basis of four projects with relatively small codebases that, importantly, were not deployed in production. In this chapter we will continue exploring the potential of dataflow architecture, this time using it to implement a production-ready ML inference platform.

Model inference has become an important part of modern ML infrastructure. Various sources estimate up to 90% of compute resources dedicated to ML are used for inference tasks [Aminabadi et al., 2022, Barr, 2019, Leopold, 2019]. To answer a growing demand for inference infrastructure, many model serving platforms appeared on the market: BentoML ¹, KServe ², Ray Serve ³. In addition, cloud providers are also offering services that simplify model serving for their users (Amazon SageMaker, Google Vertex AI). These solutions fulfil the most basic requirements for model inference, such as model hosting, unified inference API, horizontal scalability, containerisation, support for multiple modelling frameworks.

While the existing ML model serving frameworks answer some of the initial challenges of ML deployment, there are a number of important properties such frameworks need to have to ensure a seamless model deployment experience. These properties include end-to-end data traceability, monitoring of anomalies and drifts, explainability of individual models as well as an entire inference pipeline, flexible experimentation, and seamless support for synchronous and asynchronous use cases. None of the currently available ML inference platforms, open source or commercial, possesses all of these properties. However these demands are satisfied

¹https://www.bentoml.com
²https://kserve.github.io
³https://docs.ray.io/en/latest/serve/
with DOA principles and, as we have discussed in chapters 4 and 5, dataflow architecture.

This chapter describes the second version of Seldon Core (SCv2), a platform for model inference. SCv2 implements all DOA principles and follows the dataflow architecture. The inference pipeline is represented as an asynchronous dataflow graph, which provides data tracing and experimentation capabilities. Data communication is accomplished with the use of streaming via Apache Kafka [Kreps et al., 2011]. The use of data streams allows SCv2 to provide native support for both synchronous and asynchronous interaction patterns. All components communicate via Open Inference Protocol, ensuring the openness of the system. SCv2 was created as a collaboration between Seldon, a London-based MLOps company, and ML@CL research group from the University of Cambridge.

In this chapter, we discuss the motivation behind the design decisions we made in SCv2, as well as our choices of the technological stack. We describe the architecture of the platform, focusing on the particularities of implementing dataflow architecture on data streams. We present an evaluation of SCv2 performance and close with a discussion of how features of SCv2 correspond to the requirements for the next generation of ML inference platforms we presented earlier.

### 6.1 Design motivation

We shall begin by explaining the motivation behind some of the key design decisions made in SCv2 in this section.

**Dataflow architecture.** Model inference is a complex data processing pipeline. It can include input and output data transformations, multiple ML models, monitoring components, custom business logic, and so on. An example of a complex inference graph is shown in Figure 6.1. It is imperative to have a clear view of the flow of data through the entire pipeline, both for its developers and users. Following the *data as a first class citizen* principle⁴, runtime access to any intermediate data in the pipeline allows for better experimentation, troubleshooting and monitoring experience.

There are two possible ways for a model serving platform to facilitate access to a dataflow graph at runtime. It can be discovered post hoc, for example with distributed tracing systems (Jaeger⁵, Snicket [Berg et al., 2021], Redux [Nethercote and Mycroft, 2003]). This approach is applicable to platforms implemented with service-oriented approaches, such as microservices. While a popular paradigm for software system design, service orientation might be ill-suited for ML inference, as its control flow nature poorly reflects the data-centricity of the inference process [Stopford, 2016]. Alternatively, inference pipelines can be built with a dataflow-first approach.

---

⁴Throughout this chapter we will be referring to DOA principles described in chapter 3. To avoid excessive repetition of referring to that chapter every time a principle is mentioned, we will instead highlight principles with *italic font* to make the reference explicit.

⁵Jaeger: open source, end-to-end distributed tracing, [https://www.jaegertracing.io/](https://www.jaegertracing.io/)
approach, such as flow-based programming (FBP), as is already the case for ML training pipelines [Cai et al., 2016, Mahapatra and Banoo, 2022]. As we have observed in the previous chapter, FBP provides access to the dataflow graph naturally [Paleyes et al., 2022a], and thus might be a more appropriate choice for the implementation of inference graphs. FBP also fulfils the decentralisation principle, which is beneficial for pipelines where steps are expected to take unequal time for processing. For these reasons, we decided to follow the dataflow architecture for SCv2.

**Pipeline component abstractions.** To allow for the construction of complex data pipelines as shown in Figure 6.1, simple but flexible architectural building blocks are required. ML models usually run on dedicated specialised servers. The models will have generally been created by data scientists while the final serving infrastructure is usually handled by separate dedicated operations teams. The definition of models and servers for inference should be kept separate to allow for their distinct creation and the possibility of model sharing across servers.

Given a set of models, many data pipelines may be built which share them at inference time. Consequently, a pipeline abstraction should be a higher-level concept that defines the flow of data between the functional steps and how that data is joined and split as needed. Teams should be able to tap into any data source, consume any output stream or extend the inference pipeline for extra processing.

SCv2 defines a range of building blocks from which the users can build inference pipelines with the YAML-based configuration mechanism which we describe later. Following the **openness** principle, the components interact via Open Inference protocol, providing users with additional
flexibility in how the components are combined.

Support for synchronous and asynchronous scenarios. Modern ML inference services should natively support two modes of operation: synchronous and asynchronous. Synchronous inference is the traditional mode of doing inference, also known as request-driven batch processing. In this mode, a user makes a request with a batch of input data points to the platform and receives a response with corresponding predictions. For that interaction to happen the inference platform has to provide an API endpoint suitable for accepting prediction requests [Team, 2016]. Asynchronous inference allows for a different interaction pattern, where the input data is arriving and the predictions are produced continuously. Such behaviour can be achieved with data streams and is often referred to as stream processing [Shahrivari, 2014].

With real-time ML gaining attention because of its data availability, flexibility and ability to scale [Huyen, 2020], asynchronous use cases will be encountered more often. Stream processing possesses a range of qualities which make it better suited for real-time analytics [Wingerath et al., 2016], such as the ability to adapt to variable workloads [Das et al., 2014, Lohrmann et al., 2015]. At the same time, reports estimate around 77% of software development teams are working with service-based architectures [O’Reilly, 2020], which means it is easier for many users to include batch processing APIs in their software setup. To satisfy the need for both interfaces, hybrid approaches are now being proposed, in an attempt to provide convenient interfaces for both synchronous and asynchronous interaction modes [Carbone et al., 2015, Dissanayake and Jayasena, 2017, Fino et al., 2021].

Following these observations, SCv2 is built as the stream-first platform composed of asynchronous components (subprinciple of openness) with gateways for additional service-oriented interactions with pipelines and individual models.

6.2 Seldon Core v2 platform architecture

The first version of Seldon Core (SCv1) was designed in a classic service-oriented manner following the central orchestrator pattern: there was a central orchestrator service controlling the overall inference execution, and that service made calls to all the other parts of the pipeline, also exposing their functionalities as service APIs. As we have already learned, service orientation is not optimal in terms of accessing intermediate data state, which is a critical requirement for hosting ML inference workflows and troubleshooting their performance. Additionally, the central orchestrator controlled the entire topology of the pipeline, making experimentation hard and intrusive. Moreover, this design was limited in terms of asynchronicity support and ability to recover from pipeline execution failures.

In this chapter, we explore whether DOA principles through a dataflow architecture can address these pain points. To this end, we designed Seldon Core v2 (SCv2) as a fully decentralised system closely following the dataflow paradigm, with models and other inference pipeline steps
Figure 6.2: Seldon Core v2 control plane. The user interacts with the platform via operators. The scheduler is responsible for the allocation of models and pipelines. Models are hosted on an inference server (MLServer or Nvidia Triton), fronted by an agent. The data plane is managed by a dataflow engine, and gateways take care of conversions between HTTP requests and data streams.

running in separate Docker containers fronted by a service proxy and communicating data via Apache Kafka streams. In this section, we discuss the platform’s control and data planes.

6.2.1 Control plane

The control plane in SCv2 consists of multiple components depicted in Figure 6.2. Each component has a separate function in platform orchestration.

For an end user interaction with the platform happens via operators. SCv2 can run on or off Kubernetes and therefore provides the Kubernetes operator as well as the CLI operator. Operators turn user commands into gRPC or REST requests and send them to the platform to handle. Management requests are handled by the scheduler, which is responsible for the allocation of models and pipelines. Pipelines are defined as YAML files, an example of which is shown in Figure 6.3. Routing and load balancing of communication between services in the data plane is done by a service proxy (SCv2 is using Envoy, [Klein, 2017]).

SCv2 supports two inference servers: MLServer\(^6\) and Nvidia Triton\(^7\). Each of these technologies offers unique capabilities in terms of hardware acceleration and the modelling frameworks compatibility, therefore it is beneficial for an inference platform like Seldon Core to support both. At the same time, they have different ways of managing uploaded models, which may

---

\(^6\)https://github.com/SeldonIO/MLServer

\(^7\)https://developer.nvidia.com/nvidia-triton-inference-server
apiVersion: mlops.seldon.io/v1alpha1
class: Pipeline
metadata:
  name: example-pipeline
  namespace: seldon-mesh
spec:
  steps:
  - name: resize-image
  - name: vectorize
    inputs:
      - resize-image
  - name: detect-objects
    inputs:
      - vectorize
  - name: count-objects
    inputs:
      - detect-objects
output:
  steps:
  - count-objects

Figure 6.3: Pipeline definition YAML example. This is a simple linear pipeline, with two preprocessing steps (resizing and vectorization of images), one inference step that detects objects on an image, and a postprocessing step that counts detected objects.
create inconsistencies in the control plane. To abstract away details of interfacing with a particular inference server, the communication between an inference server and the SCv2 platform is managed via agents. SCv2 runs two types of agent services, one for MLServer and one for Triton, that provide a common interface to manage models on the inference server of choice.

All data communication in SCv2 is done over data streams. However, the models and other pipeline steps are run within inference servers that serve models via service-oriented APIs. This creates a need for a mechanism that can convert service calls to and from data stream records, which in SCv2 is accomplished by a model gateway. Similarly, a pipeline gateway handles an incoming inference request, converts it to a data record to put on a pipeline input stream, waits for a corresponding record to appear on a pipeline output stream, and converts it to an inference response. The gateways allow for synchronous interaction with an inherently asynchronous platform.

Finally, the dataflow engine is responsible for orchestrating the entire data plane of the platform. In the remainder of this section, we will discuss the details of how the dataflow engine is implemented, how dataflow architecture is realised, and how DOA principles are followed in SCv2.

### 6.2.2 Data plane

From the high level, the data plane in SCv2 is organised as a series of processing nodes that receive data from the input streams and output computation results into output streams, thus forming a directed dataflow graph. The evaluation of the graph is triggered by an incoming inference request that is added to the pipeline input stream and finishes when the corresponding record arrives in the pipeline output stream. Figure 6.4 displays a simple linear application in SCv2.

We use Apache Kafka for data streaming in SCv2. There were several considerations behind this choice. SCv2 is an open-source project, and therefore all its dependencies have to be open-source. SCv2 also needs to be platform-agnostic, as users might want to run SCv2 on their laptops, on commodity hardware, or in the cloud. This also means we need a streaming framework that is relatively lightweight in terms of its hardware demands as well as setup, ruling out function-rich but complex frameworks such as Apache Flink. Additionally, Kafka Streams library was a big differentiating factor for us because of all the streaming primitives it provides, which we would have to implement ourselves otherwise. Finally, we decided against message queues because of pub/sub semantics that data streaming provides.\(^8\)

\(^8\)https://flink.apache.org/
\(^9\)https://kafka.apache.org/documentation/streams/
\(^10\)In this section we assume familiarity with core Kafka concepts, such as topics and consumer groups. We recommend referring to official Kafka documentation for a detailed explanation of these concepts: https://kafka.apache.org/documentation/streams/
Figure 6.4: High-level overview of the Seldon Core v2 data plane. Incoming inference request is received by a pipeline gateway, converted and passed on to a pipeline input stream, which triggers the evaluation process. Each pipeline node has an input data stream it listens to. When the input data arrives, the node processes it and produces output (or outputs) that are sent to the node’s output stream. Eventually, this flow arrives at the stream that holds the pipeline output, at which point it is converted to an inference response and returned.

There are multiple reasons why the pub/sub model offered by data streams is a good fit for an ML pipeline. Firstly, it fully complies with DOA principles and allows for naturally open and decentralised applications. Secondly, it allows us to reuse a single model across multiple pipelines. It is achieved by multiplexing pipelines on a single topic and differentiating between them with Kafka headers. This feature relies on the ability for a single record to be read, non-destructively, by an arbitrary number of pipelines. There is a separate consumer group per pipeline, and consumers who are not interested in a pipeline can simply discard messages for it and move on. Thirdly, pipelines can be trivially extended without affecting the rest of the graph. This is a benefit of data coupling that the pub/sub model exhibits, as well as the decentralisation principle of DOA. A new processing node can be setup to listen to one of the existing data streams in the pipeline, without impacting any of the existing nodes. Fourthly, Kafka streams are repeatable, meaning one can replay the exact sequence of events in a stream. This feature can be used for incident analysis, benchmarking, and explanations.

6.2.2.1 Data format

To support data coupling and openness of components, models and pipelines in SCv2 have two Kafka topics each associated with them, one for input and one for output data records. This means the Kafka cluster in a SCv2 deployment can have a lot of topics, and requires an unambiguous way to address them. The naming scheme we adopted has the following format:

\[
< \text{prefix} > . < \text{namespace} > . < \text{pipeline|model} > . < \text{name} > . < \text{inputs|outputs} >
\]
Each topic name starts with a prefix, which could be an organisation name or other literal. The namespace is required for compatibility with Kubernetes. Outside Kubernetes, the namespace is set to “default”, although it can be configured to support namespacing in other orchestration systems. The next part specifies if a topic belongs to a pipeline or a model, followed by the name of that entity. The final part of the topic’s name specifies if it is an input or an output topic. This means that if a model B directly consumes the output of model A, there will be two topics, \(...model.A.outputs\) and \(...model.B.inputs\), and the dataflow engine will perform mapping of the data between them. Having an explicit, per-model topic for outputs, allows users to extend the pipeline in the future. We will describe the mapping process in more detail later in this section.

Developers operating SCv2 need a straightforward way to receive signals about issues and failures happening within the pipeline. For that reason, there is also a dead letter queue in SC2, a special topic to surface errors in the SCv2 deployment. If a model fails to return a response, this information is propagated via this special topic and can be used to respond to synchronous requests, as well as for audit trails. It has the following name format:

\(<prefix> . <namespace> . errors.errors\)

The structure of the message in the SCv2 data plane follows the standard metadata-payload format. The metadata block contains headers (such as pipeline name) and a request ID as the Kafka topic key. The payload in SCv2 is always an Open Inference Protocol\(^\text{11}\) message in Protobuf format. The fact that all components in SCv2 communicate via single message exchange protocol makes the system open for modification and extension, in agreement with openness principle in DOA.

### 6.2.2.2 Data manipulation

The fact that components in SCv2 are autonomous and coupled with data means the data engine needs to perform many operations related to moving data records across streams. In this section, we will discuss operations that are currently supported.

Each pipeline step in SCv2 ingests and outputs a stream of data, which together combine to the shared data model of the pipeline. The most basic operation the dataflow engine carries out is moving an entire inference message from one topic to another, also known as topic chaining. Typically the source topic is the output of a model/step, and the destination topic is the input to another model/step. The data and the request ID are copied over, as well as the headers. Similarly, topic chaining occurs when data is going from pipeline input to the first model input, and likewise from the last model output to the pipeline output.

---

\(^{11}\)Open Inference Protocol is a standard protocol for performing ML model inference across serving runtimes for different ML frameworks, which SCv2 follows. More information about it can be found at https://github.com/kserve/open-inference-protocol.
In the simplest case, the payload is copied as is. However, since the inference pipeline steps can be developed independently, their developers may have made different naming decisions, and the names between output and input tensors may not match. In addition, it is possible the downstream model does not use all the tensors produced by the upstream. To pass and discard unnecessary tensors, a more frugal approach to data coupling is necessary. The process of matching non-identical tensors in SCv2 is known as tensor projection. The user has the option to specify such cases in the configuration, and the dataflow engine performs the data mapping accordingly:

```
steps:
  - name: model1
  - name: model2
    inputs:
      - model1
    tensorMap:
      model1.outputs.OUTPUT0: INPUT0
      model1.outputs.OUTPUT1: INPUT1
```

As an optional optimisation technique, SCv2 supports batching, that is concatenating the data from several individual requests along each tensor. While not required under any DOA principles, this can be used to maximise the throughput and efficiency of the model inference. It can also be used in cases where models operate on a series of data points, e.g. models for detecting drift in the data. Batching is a stateful operation, and the dataflow engine uses KTables\(^{12}\) for implementation of this feature. To enable batching, the user needs to specify the batch size for a pipeline step:

```
- name: drift-detection-step
  batch:
    size: 20
```

By representing batches within the data flow, SCv2 ensures consistent treatment of all types of messages (batched or not) across the deployment. Such consistency is vital for data-driven systems in which entities are autonomous.

Components of DOA systems operate autonomously on local chunks of data. These data chunks can be different across different components and therefore the system needs to support the branching and joining of data streams so that necessary data records can be formed for each component. In SCv2 branching is realised via chaining and projection operations described above. SCv2 also supports the joining of multiple streams, which can be seen as a reverse

\(^{12}\)More information on the KTable abstraction from Kafka Streams library can be found on the Confluent documentation website: https://developer.confluent.io/learn-kafka/kafka-streams/ktable/.
transformation. All joins are bound to a request ID, meaning that only the data related to a specific request can be joined. SCv2 defines three types of join semantics. **Inner join** requires all input streams to be present for a transaction to join the tensors passed through the pipeline. If not every stream has a record for a certain request ID, a message for this request is not put in the joining stream. **Outer join** requires a subset of all streams to be available. The dataflow engine waits for the messages to arrive during a given time window and then collects messages that are available while ignoring streams where the data on the streams was not produced. **Any join** is a special type of outer join that needs only one message to arrive at any of the joined streams. This operation can be useful in multi-armed bandit scenarios, e.g., experimenting with multiple versions of the same model running in parallel.

For added flexibility of **dataflow graph** definition, joins are also used in SCv2 in a mechanism called **triggers**. Trigger is an instrument that implements conditional logic in the pipeline. If a step in the pipeline has a trigger, the flow of data into this step needs to wait until records on one or more trigger streams appear. Trigger streams can also be joined, realising **OR** and **AND** conditions. The data in these triggers is not passed onward from the join.

### 6.3 Data observability features

As SCv2 follows all principles of DOA systems, its users can leverage access to all intermediate data states and the dataflow graph. While **openness** of the platform means it is possible to interact with individual Kafka topics and standalone containers that serve inference steps, SCv2 also provides several high-level features that build on its data observability qualities. We now describe some of these features in SCv2 that the dataflow architecture and more generally the DOA approach bring.

#### 6.3.1 Data tracing

As we mentioned in the design motivation section, model inference is a complex process that can have multiple steps, forming a possibly non-trivial pipeline. Developers working with such pipelines require a quick and convenient way of tracing requests and retrieving intermediate data states, a key property of observable **data-driven** DOA systems. Straightforward access to data, its states and history gives developers the ability to perform tasks ranging from simple debugging to complex provenance [Carata et al., 2014].

To that end, the SCv2 operators provide a ‘pipeline inspect’ command. This command allows the user to access data from any input or output stream within a pipeline or a specific tensor in that stream. It can also retrieve data states from all pipeline steps for end-to-end tracing. The dataflow design of the data plane allows for a unified interface and implementation of this command at all levels of granularity.
6.3.2 Experimentation

Machine learning is an iterative process where new components are updated over time to better harness the data that flows through them. Therefore, an inference system needs flexible and simple ways to test updates both to individual atomic models and entire data pipelines. Furthermore, as the number of data pipelines increases the task of validating experiments manually becomes more onerous. In recent years the use of progressive rollouts for new models where clear service level agreements (SLAs) and acceptance criteria are defined and the data traffic is managed automatically between challenger and champion models is becoming more popular with tools on the market (e.g. Argo 13, Flagger 14, Iter8 15). The dataflow approach allows the extension of these techniques to handle entire data pipeline rollouts efficiently. In the dataflow view model and pipeline are both graphs (on single and multiple nodes respectively) with edges going in and out of them. As a result, the experiments on models and pipelines in SCv2 can be set up and conducted in the same uniform way.

6.3.3 Monitoring and explainability

The key responsibility of monitoring is to identify abnormal situations in the inference system and allow fast and accurate root-cause analysis (RCA). A good monitoring solution for ML systems needs to be able to flexibly allow RCA for any connected set of components in the inference graph. Access to the dataflow graph at runtime provides the necessary intermediate data streams and runtime graph traversal operations to achieve such flexibility. A particular example of such a flexible monitoring feature is context-aware drift detection [Cobb and Van Looveren, 2022], where a data drift observed downstream can be analysed jointly with some of the upstream data used as a context. Another example is recursive attribution [Singal et al., 2021], where an observed behaviour is linked to the most likely cause, and this procedure is repeated recursively until the user input is reached. In addition, operations and auditing teams need to be able to explain on demand any part of a pipeline for technical, regulatory or business reasons [Klaise et al., 2020]. Data snapshots themselves might not always suffice to derive such explanations, and the dataflow graph allows dynamic addition of ML interpretation models, which in SCv2 are referred to as explainers.

6.4 Evaluation

In the previous chapter, our main goal was to illustrate the potential of the dataflow approach to AI software engineering, which is why we used artificially constructed applications with relatively small codebases. The small size of these systems ensured that the business and data
science functions were not obscured by the technical complexity. Compared to them, SCv2 is a much larger and more complex software whose main purpose is to serve ML models in a large software infrastructure. That means SCv2 has to be flexible and robust to support various patterns of workloads. Therefore in the evaluation section, we will focus on metrics that are critical for any production system, such as latency and throughput. Our main goal is to understand if the data observability offered by SCv2 leads to a decrease in performance, and if so to measure this decrease. This will also allow us to gain insight into the operational complexity of running a DOA system, one of the main open questions highlighted in chapter 3. Additionally, we apply SCv2 to one of the applications used in chapter 5 for continuity.

6.4.1 ML deployment stages

We begin our evaluation of SCv2 by replicating the experiment conducted in the previous chapter. As a brief reminder, we observe the evolution of a data processing application through stages of ML model deployment and measure the impact of these stages on the codebase with code quality metrics. For a detailed description of the experiment please refer to section 5.1. We have chosen to replicate the experiment for the “Insurance claims” application, also described in Chapter 5.

Figures 6.5 and 6.6 show the results of the experiment. Interestingly, FBP and SCv2 codebases show almost identical metrics for the ml stage. However, there are a few differences between these codebases that appear in the data collection stage.

Compared to the vanilla FBP implementation we have used in the previous chapter, the SCv2 version of the same application has some significant differences. On one hand, there is no need to implement traversal or data collection operations, as these functionalities are provided by the “inspect” utility of SCv2. This excludes the necessity to implement complex graph traversing code. On the other hand, messages in SCv2 are formatted using Open Inference protocol, and at the moment simple boilerplate code is necessary to extract the payloads from these messages. In other words, while FBP required a smaller amount of complex code at the data stage, SCv2 required a lot of (relative to the size of the codebase) simpler changes. This explains the much higher growth of the LLOC metric (Figure 6.5a) and the smaller drop of the Maintainability Index (Figure 6.5b).

Overall, we broadly observe the same results for SCv2 that we saw previously for the FBP-based solutions: required changes are simpler to understand and more localised than their SOA counterparts. This result is expected, as SCv2 is designed as a dataflow platform, similar to the way FBP applications in the previous chapter were built.
Figure 6.5: Measurements of the impact data collection task on FBP, SOA and SCv2 implementations of the “insurance claims” application. SCv2 changes are more localised and simpler. The higher growth of the LLOC metric is explained by a much smaller size of the initial codebase (140 lines for SCv2, compared to 210 lines for FBP and 247 for SOA).

6.4.2 Dataflow overhead

While DOA offers significant advantages for data discovery and management in software systems, it is unclear how efficiently it can be run compared to more traditional service-oriented setups. Intuitively, data coupling means all data shall go through some data storage medium, instead of being sent directly from one service to another, and that might incur a processing overhead. The second experiment we present is designed to illustrate the added latency overhead of dataflow-related components and operations (data streams, gateways, conversions between HTTP requests and stream records) in SCv2.

We consider a simple add10 model: given a list of integer numbers, it adds 10 to each number. Since this model ingests and produces a tensor of exactly the same shape and type, it can be fed its own output, allowing us to reapply this model as many times as necessary. Using this fact, we design two pipelines with it, both of which apply add10 model to an inference request.
Figure 6.6: Measurements of the impact ML model integration task had on FBP, SOA and SCv2 implementations of the “insurance claims” application. FBP and SCv2 exhibit almost identical results and demonstrate improvement upon SOA-based solution.

For the second pipeline, we develop a helper script, also deployed to the same MLServer container host, that is able to call \( \text{add10} \) model repeatedly \( N \) times, bypassing the entire dataflow machinery and thus not being subject to latency overheads introduced by data streaming or proxying service calls. We shall refer to this pipeline as “in-server pipeline”. The two pipelines are shown schematically in Figures 6.7 and 6.8. As the number of steps in the pipelines grows, we expect to see an increasing difference between them in the end-to-end latency.

Since the calculation in \( \text{add10} \) model is simple and deterministic, its effect on latency is negligible, allowing us to attribute all processing time to SCv2 control and data planes. This experiment is performed in a single machine with 4 CPUs and 32 GB of RAM. In the
Figure 6.7: **Dataflow pipeline** for dataflow latency overhead experiment. An inference request goes through $N$ steps of reading data from an input stream, sending it to the model, and putting the model’s response to an output stream.

Figure 6.8: **In-server pipeline** for dataflow latency overhead experiment. Once an inference request is read from an input stream, it does not leave the MLServer and goes through $N$ iterations between the $\text{add10}$ model and the helper testcaller model. After all $N$ iterations are completed the result is returned as normal.
Results of SCv2 dataflow overhead experiment

Figure 6.9: Results of dataflow overhead experiment. The left plot shows the absolute end-to-end latency of the two pipelines handling a single inference request. Mean value as well as a region between 5th and 95th percentiles are shown. We observe a linear increase in the metric of interest as the number of steps in a pipeline increases. The right plot shows a mean difference in latency per step between two pipelines. The difference stays almost constant at 1 ms. The negative difference for 1 and 2-step long pipelines is due to the in-server pipeline having to operate two models, which outweighs streaming overheads in short pipelines.

next experiment, we will see how better hardware capacity leads to an improvement in the performance of SCv2.

The results of the experiment are shown in Figure 6.9. The evaluation is done for pipelines $N = [1, 2, 5, 10, 20, 40, 60, 80]$ steps long, each measurement repeated 20 times. The plot on the left shows the mean end-to-end latency of handling a single request, as well as 5th and 95th percentiles. The results confirm our intuition that the data observability features offered by the dataflow approach come at a cost of increased end-to-end latency. As can be seen on the right plot, the added cost amounts to an average of 1ms per pipeline step, which includes one cycle of detecting a new record on a Kafka stream, converting it to an HTTP request, sending a request to an inference server, waiting for a response, converting the response to a stream record and sending it to a Kafka stream. Notice that the dataflow pipeline exhibits slightly higher variability in the response time compared to the in-server pipeline, induced by the variability in time gateways required to detect a new message in the stream and pull it.

6.4.3 Performance under load

The next experiment aims to measure the performance of the system under progressively increasing load, for which we use the k6 load testing tool. The tests were conducted on the GCP fleet of 4 e2-standard-32 instances that have 32 virtual CPUs and 128 GB of memory each, which ensured we did not encounter hardware limitations. We fix the number of Kafka brokers

16Grafana k6: Load testing for engineering teams, https://k6.io/
Results of SCv2 load testing

Figure 6.10: Results of load testing layers of the control plane of SCv2. We start with sending requests to the full pipeline, then to the Envoy proxy bypassing the dataflow engine, then to the agent fronting the inference server, and finally to the server directly. The throughput of the system (left) grows at a variable rate for different layers but reaches the common peak of approximately 2650 requests per second. Full pipeline mode shows a higher initial latency of added 5-10ms compared to other layers (right). Latency then grows uniformly for all layers as the load increases. At the peak load full pipeline test again shows a latency overhead of approximately 5ms.

The experiment is designed to stress the components of SCv2 jointly as well as individually. We deploy a single-step pipeline with a Tensorflow model that was trained to classify MNIST digits. The inference requests are sent to the model, and the number of layers of the control plane the request has to go through is gradually reduced. We start with the full SCv2 pipeline, then the dataflow engine is removed and the request is sent to the Envoy proxy, then the request is sent to an agent bypassing the proxy, and finally, we query the inference server directly. The load of the system is controlled by the number of concurrent clients sending inference requests, starting from 10 clients and peaking at 100 clients. Respectively, we deployed 100 replicas of the same ML model to the inference server to ensure concurrency inside the inference server does not become a bottleneck.

Results of the load testing are shown in Figure 6.10, which reports throughput (left) and latency (95th percentile, right). We observe that full pipeline mode adds noticeable overheads under low load both in terms of throughput and latency. However as the load increases and the system approaches its peak performance (under fixed hardware limits), the difference between layers decreases. All modes reach the same top throughput of approximately 2650 requests per second. This upper limit exists as the inference server cannot process more requests and the incoming queue of requests forms. Similarly, when the load reaches 50 concurrent clients (hence at most 50 concurrent requests), all modes exhibit close to identical latency, again due
to the queue of requests gradually building up. We can therefore conclude that under high load the overhead added by the communication and conversion of the dataflow engine layer becomes negligible and secondary to the ability of the ML model and other inference steps (as well as the hardware they are being run on) to handle the load.

6.4.4 Discussion

The evaluation results reveal that the additional levels of processing that the inference request undergoes in SCv2 introduce a small but noticeable latency overhead. Concretely, under the low load the added latency grows by approximately 1 ms per each consecutive step in the inference graph. Nevertheless, under high load the throughput of the system flattens out, indicating that the hosted ML model and other inference steps play a higher role in the overall performance of the system, compared to the dataflow engine and other parts of the control plane. Additionally, we showed that the observations we made in chapter 5 for synthetic FBP systems still hold for a production-level dataflow platform, even though SCv2 would benefit from utilities to simplify parsing of Open Inference protocol messages.

High-load production systems are extremely sensitive to fluctuations in performance, therefore it is important developers understand the trade-offs involved in developing DOA-style software. For SCv2 the trade-off is necessary for achieving better observability, reproducibility and extensibility in the system. We therefore advise practitioners to carefully consider the performance and functionality offered by SCv2 against their requirements.

6.5 Summary and Conclusions

In this chapter, we presented Seldon Core v2, an ML inference platform architected according to dataflow principles. We discuss the reasons why dataflow is a good fit for an MLOps system and provide motivation for some of the main design decisions in SCv2, including the usage of Kafka streams for data communication. We also discussed the way control and data planes are organized in the platform, with a special focus on the intricacies of implementing dataflow processing on streams, and explained how each design decision is connected to DOA principles.

We also presented evaluation results for SCv2 and discussed its ability to scale under load and the latency introduced by the control plane components. We measured the added overheads and discussed the price paid in performance gains in terms of important data observability features of the system, such as flexible experimentation, monitoring and explainability. Nevertheless, we do not think the potential of the system’s design is explored fully yet. In particular, the dataflow approach is beneficial for auditing, compliance and privacy of the data processing system [Tang and Østvold, 2022, Schwarzkopf et al., 2019]. Turning this potential into concrete features of SCv2 is an exciting direction for the platform’s growth.
Seldon Core v2 has become a popular dataflow system\textsuperscript{17} that has both open-source and commercial users. Since its launch SCv2 users created over ten thousand Docker installations\textsuperscript{18} and opened over 300 GitHub issues and pull requests\textsuperscript{19}, indicating popularity of the platform in the community.

Machine learning inference is a vital part of the wider MLOps agenda, which is only going to grow in importance as the demand for safe and scalable deployment of ML increases. As we have shown in this chapter, the dataflow design of an ML inference system provides a way to satisfy the requirements inference platforms are facing while maintaining a reasonable level of performance. However, the dataflow architecture can also benefit our ability to understand and maintain the software. In the next chapter, we will use SCv2, as well as other dataflow frameworks, to illustrate an exciting connection between dataflow architecture and causal inference for software systems.

\textsuperscript{17}https://github.com/SeldonIO/seldon-core/tree/v2
\textsuperscript{18}This can be seen on Docker Hub. Seldon Core v2 consists of multiple components, all of which are required for installation. Therefore we can refer to any of them to retrieve the installations data, for example to the dataflow engine: https://hub.docker.com/r/seldonio/seldon-dataflow-engine
\textsuperscript{19}https://github.com/SeldonIO/seldon-core/issues?q=label%3Av2
Chapter 7

Dataflow graphs as causal graphs

In the previous chapters, we have reviewed the challenges posed by the process of deploying machine learning (ML) to production and focused on dataflow architecture as a way to design systems less susceptible to these challenges. We discussed the advantages of dataflow architecture for ML deployment and illustrated them with prototypes (Chapter 5) and production-ready (Chapter 6) software systems. We also discussed how dataflow realises the principles of the DOA paradigm. In this chapter, we consider one more way in which the dataflow approach to system design can be beneficial: applications of causal inference to systems. We point out the connection between dataflow graphs and structural causal models and use multiple expository examples to show how this connection can be leveraged to improve day-to-day tasks in software projects, with a particular focus on causal fault localisation. The ability to create software with clear causal relationships between components can significantly decrease the intellectual debt we incur while building complex systems.

The modern software engineering industry has become increasingly data-centric. There are two primary motivations behind this trend. First, as we have discussed in Chapter 2, businesses and research groups seek to deploy more data-driven software solutions, including those powered by ML, which means software engineers are more often faced with data availability and analysis requirements [Lewis et al., 2021, Paleyes et al., 2022c]. Second, the rise of DevOps as a discipline of software maintenance practices emphasises attention to metrics that help assess and investigate hardware and software health. These metrics build on a growing amount of technical data which, while not directly related to a business domain, still needs to be collected, stored and analysed [Forsgren and Kersten, 2018].

However, raw data is often not enough to explain certain behaviours or answer questions about a software system. Modern software applications are complex and consist of a large number of components. It is often necessary to understand causal relationships between these components to effectively use the data available to answer business and technical questions, as well as to ensure quality [Clark et al., 2023, Clark, 2023]. The absence of such understanding can harm developers’ ability to maintain software in the long term, an effect known as “intellectual..."
debt” [Zittrain, 2019]. We saw examples of such problems that occur during the process of ML deployment, especially in relation to data management (Section 2.2) and model deployment (Section 2.5). Dataflow graphs of all data states and transformations can assist in understanding the system’s inner mechanics. Unfortunately, as we have discussed many times in this thesis, most common software paradigms, object and service orientation, focus on protecting data and making recovery of dataflow graphs a complex task [Paleyes et al., 2022a, Lin and Ryaboy, 2013, Stopford, 2016].

In this chapter, we suggest that, in addition to the benefits of the dataflow approach shown before, there is a significant potential to leverage the connection between dataflow graphs and structural causal models. In particular, we argue that dataflow graphs of software systems can be treated as complete causal graphs, and unlike other software design methodologies dataflow design provides such graphs natively. We highlight the importance of causally-aware decision-making in systems, as it can evaluate the effects of actions and guide future policies. A key bottleneck of causal reasoning methods is their reliance on a causal graph for inference tasks. This limits the applicability of causal inference [Dawid, 2010], and hence much of the theoretical work focuses on discovering causal structure from observational data [Peters et al., 2017, Guo et al., 2022]. We argue that dataflow architecture and its flavours, such as flow-based programming (FBP), have the advantage in offering a known causal graph, which allows for a compact and efficient representation to infer effects and attribute explanations to address practical engineering questions.

Our work is not the first to consider applications of causality to software systems. Causal testing tool Holmes [Johnson et al., 2020] generates test cases using input perturbations and then applies counterfactual causality to help developers identify sources of buggy behaviour in programs. Feature effects and interactions can be estimated with the notion of feature causality [Dubslaff et al., 2022]. BoGraph uses causal structure learning to auto-tune complex parametrised systems [Alabed and Yoneki, 2022]. Closer to the topic of our work, causality was used for fault localisation in software on method [Shu et al., 2013, Küçük et al., 2021] and program levels [Baah et al., 2011], as well as in cloud computing [Aggarwal et al., 2021] and microservices [Ilkram et al., 2022]. The common challenge for these works is the necessity to discover the underlying causal graph of a software system post hoc [Siebert, 2023]. Dataflow architecture resolves this problem, as it produces a dataflow graph as a part of the design process. This emphasises the need for a dedicated study of the connection between causality and dataflow systems, and the present chapter makes a step in that direction.

7.1 Motivating scenarios

This section introduces several scenarios of typical problems we aim to address. The setting for this section is a fictional online coffee retailer CoffeeFlow. CoffeeFlow operates a supply chain
of coffee beans: they order coffee from the vendors, store it in the warehouses, and deliver it to customers worldwide. To automate routine tasks CoffeeFlow invested in the development of a software platform that models the entire supply chain process. Following modern practices of distributed system design, the platform is implemented with microservices. Each service is maintained by a separate team of software engineers.

**Scenario 1.** Nadya is a software developer on pager duty for a service that handles customer orders. She receives a notification that her service’s latency is spiking. Nadya investigates and finds out that the latency increase is associated with a corresponding increase in the upstream service. She engages her colleague, who again identifies the problem up the call stack. Eventually, the issue is located three levels up from Nadya’s service and is due to a memory shortage of the warehouse storage management service. The pager alarm should have gone to a different person, saving precious time. Can this troubleshooting process be expedited?

**Scenario 2.** Arthur is a business analyst at CoffeeFlow. He has recently noticed that the company’s sales in a certain region have dropped by 5%, and investigates the issue. Arthur spends multiple days reviewing and analysing every business metric in that region and identifies several potential causes: (i) some vendors delayed delivery, (ii) demand has diverged from the forecast over the past month, and (iii) a temporary shortage of staff at one of the warehouses. Arthur proceeds to produce a report which gives the best estimates of how likely, in his opinion, each issue affected the sales figure. Is there a way to make this analysis process more precise?

**Scenario 3.** Alisa is an applied ML scientist. She has recently finished training a new forecasting model to improve customer demand prediction. Model quality metrics show considerable improvement. However, Alisa has concerns that some of the downstream services in the platform might have adapted to the behaviour of the previous forecasting algorithm, and thus model accuracy improvement might not translate into business value after the deployment. Can Alisa validate this without running expensive experiments?

The problems described above share a common characteristic: their resolution would be simpler and easier if the design of the CoffeeFlow platform provided its users with a better understanding of its components and their dependencies. In other words, if the platform development committed less intellectual debt. In subsequent sections, we show how the connection between dataflow and causality can yield a clearer design and thus address the questions raised above.

### 7.2 Causality

The science of causality has become invaluable both for scientific inquiry and everyday decision-making. It has a profound impact in multiple areas, including but not limited to healthcare [Glocker et al., 2021], economics [Hoover, 2017], technology [Lan et al., 2022]. In the field of AI, there is a growing realisation that AI systems need to shift from correlation-based predictions
to understanding causal relationships. Recently some of the most prominent machine learning researchers, including Bernhard Schölkopf and Yoshua Bengio, argued that “combining the strengths of both fields, i.e., current deep learning methods as well as tools and ideas from causality, may be a necessary step on the path towards versatile AI systems” [Schölkopf et al., 2021]. Inspired by these developments, we propose to use the theory of causality to answer the questions asked in Section 7.1. Here we give a quick introduction into its most relevant concepts, for the benefit of the readers.

Causality aims to discover, characterise and measure causal relationships from observational data. Current statistical methods, in contrast, exploit associative relationships to improve prediction accuracy, which is sound as long as distributions do not shift (e.g. if the data is independent and identically distributed). However interventions in a system generally lead to a distribution shift, and correlation does not imply causation. For example, the website Spurious Correlation \(^1\) illustrates that the total revenue generated by arcade games is correlated with the number of computer science doctorates awarded in the US. Although this information can help us predict arcade revenue given the number of computer science doctorates awarded, it does not mean that if we want to improve the revenue of arcade we should award more doctorate degrees. The key difference is that rather than focusing on prediction, causality allows for actions. Below we introduce the language of structural causal model and show its natural connection with dataflow graphs.

Definition (Structural Causal Model (SCM, Pearl [2009])) A SCM \(\mathcal{M}\) consists a set of variables \(X_1, \ldots, X_d\) and corresponding structural assignments of the form

\[ X_i := f_i(P A^\mathcal{G}_i, U_i) \quad (7.1) \]

for all \(i \in \{1, \ldots, d\}\) where \(P A^\mathcal{G}_i\) are parents of \(X_i\) in graph \(\mathcal{G}\), referred to as direct causes of \(X_i\). \(U_1, \ldots, U_d\) are noise variables, which we assume to be jointly independent. Here joint independence means for a sequence of random variables \(U_1, \ldots, U_d\), the joint distribution can be factorised as \(p(u_1, \ldots, u_d) = \prod_{i=1}^d p_i(u_i)\). Further given a SCM \(\mathcal{M}\), there is a corresponding directed acyclic graph (DAG) \(\mathcal{G}\) where each variable \(X_i\) has incoming edges from all the parent in \(P A^\mathcal{G}_i\) to \(X_i\). Any distribution generated by an SCM \(\mathcal{M}\) can be factorised into causal conditionals via the Markov factorisation

\[ P(x_1, \ldots, x_d) = \prod_i p(x_i | pa^\mathcal{G}_i) \quad \text{causal conditional} \quad (7.2) \]

Definition (Intervention [Didelez et al., 2006]) An intervention is the act of affecting the system to control the outcome. In SCM it is denoted using the do-operator. Let \(\sigma_X\) represent the intervention on the random variable \(X\).

\(^1\)http://www.tylervigen.com/spurious-correlations
• **Atomic intervention** $\sigma_X = \text{do}(X = x_0)$: It sets the discrete random variable $X$ to a fixed value $x_0$ such that $p(x | pa^G_X; \sigma_X) = \delta(x, x_0)$, where $\delta(x, y)$ takes value one if $x = y$ else it is zero.

• **Soft intervention**: Instead of fixing value, we may want it to take values according to some user-defined distribution $q$ possibly depending on $P A_i^G : p(x | pa^G_X, \sigma_X = d_{pa_X}) = q(x | pa^G_X)$.

We can consider the intervention as a functionality change in the computational node and we would like to find out the effects of the intervention without running the whole system. One natural solution is via **truncated factorisation** [Pearl, 2009]. It transforms the post-intervention distribution to:

- $P(x_1, \ldots, x_d | \sigma_X) = \prod_{j \neq i} p(x_j | pa^G_j)$ if $x_i = x_0$ else 0 under atomic intervention $\sigma_{X_i} = \text{do}(X_i = x_0)$.

- $P(x_1, \ldots, x_d | \sigma_X) = q(x_i | pa^G_i) \prod_{j \neq i} p(x_j | pa^G_j)$ under soft intervention $\sigma_{X_i} = d_{pa_X}.$

For readers who wish to understand the causal terms and techniques deeper we recommend books “The Book of Why: the new science of cause and effect” 2018 and “Causality: Models, Reasoning and Inference” 2009 written by Judea Pearl.

### 7.3 Dataflow graph as causal graph

Object-oriented and service-oriented architectures, while being de-facto standard approaches in modern software engineering, fail to provide developers with a complete data dependency graph of a system [Lin and Ryaboy, 2013, Paleyes et al., 2022a, Nikolov, 2018, Sharvit, 2022]. Developers either have to use sophisticated and often commercial tools (e.g. Dynatrace\(^2\), AWS X-Ray\(^3\), Jaeger\(^4\), Redux [Nethercote and Mycroft, 2003]), or invent custom ways to build data dependency graphs [Baah et al., 2011]. These approaches rely on strong assumptions about the system being analysed, and cannot guarantee completeness [Wang et al., 2021b, Chowdhury et al., 2023].

The key observation we put forth is that the dataflow graph produced in the course of building a software system with FBP or other dataflow approaches is a complete data dependency graph of the entire system. For any given node all its upstream and downstream nodes can be found by traversing the graph. The design process guarantees that the graph is complete, with no hidden inputs or connections. Out-of-the-box availability and completeness mean this graph can be used immediately for causal inference on system components. FBP programs


\(^3\)AWS X-Ray: distributed tracing system, https://aws.amazon.com/xray/

\(^4\)Jaeger: open source, end-to-end distributed tracing, https://www.jaegertracing.io/
require no additional tools to enable causal reasoning and no assumptions about hidden con-
founders. In a recent review of applications of causality to software engineering, it is noted
that discovering the full data dependency structure of a system remains the main challenge
for such applications [Siebert, 2023]. Dataflow architecture addresses this challenge since the
dataflow graph becomes available to developers as a direct output of the design process.

We now introduce a causal attribution for flow-based programming. To our best knowledge,
this is the first attempt to directly model software modules as causal conditionals. Often theo-
retical work focuses on attribution in graphical models without connection to software systems
[Singal et al., 2021] and application-oriented work requires extra effort to build data dependency
graphs [Baah et al., 2011]. Algorithm 1 demonstrates how a complete data dependency graph
help automate analysis and troubleshooting. Suppose we have \( n \) data streams \( \{X_i\}_{i=1}^n \)
and \( m \) computational nodes \( \{C_j\}_{j=1}^m \). For example, the circle nodes in Figure 7.1 represent
\( C_j \) and rectangles represent \( X_i \). Let \( (C_{ij}, C_{ij}^o) \) denote the set of (input, output) streams for a
node \( C_j \). Each record arrives in a data stream at time \( t \), which is denoted as \( X_{i,t} \). We assume
\( X_{i,t} \sim d \sim p(X_i) \) for all \( t \in T \) within a user-defined time interval \( T \). Note there are two kinds
of reasons a change in the output stream can manifest: due to the change in an input data
stream(s) and due to the change in a computational process(es).

The Algorithm 1 contains several subroutines, the implementation of which can vary. De-
viation for a computational node can be computed with KL divergence [Kullback and Leibler,
1951] or testing independence [Budhathoki et al., 2021]. Attribution that computes scores
reflecting how much a data stream node contributes to the observed shift in the output \( Y \)
can be calculated with Shapley values [Hart, 1989] or proportional change of the KL divergence
between the input data streams and output data stream, i.e.
\[
\frac{\text{KL}(p(\text{input})||q(\text{input}))}{\text{KL}(p(\text{output})||q(\text{output}))}.
\]
When proportional change is used, Aggregation backtracks multiple attribution values along the path from
a given node to the output, in other cases Aggregation is a no-op.

We can now discuss how these observations and techniques can allow practitioners to
answer questions posed in section 7.1 if the CoffeeFlow platform was designed with a dataflow
paradigm instead of microservices.

### 7.3.1 Fault localisation

With the proliferation of ML, the engineering community started to develop ML-based methods
for root cause analysis (RCA) and fault localisation in software [Ascarì et al., 2009, Wong et al.,
2016, Zheng et al., 2016]. However, ML algorithms struggle to generalise well to rare events
[Lal and Pahwa, 2017]. Bugs that take significant developer time are not typical, and therefore
time saving enabled by classical supervised and semi-supervised ML techniques is limited
especially if automation is desired. However, the ability to automatically explain the origin
of a rare event in software may arise from knowledge of causal relationships between system
components. For example, Baah et al. [2011] analyse a computer program and produce two
Algorithm 1 Change attribution in software dataflow graphs

**Input:** Dataflow graph with data streams $X_i$ and computational nodes $C_j$; $p \in D^{\text{new}}, q \in D^{\text{old}}$ distributions of all data streams in time frames $T^{\text{new}}, T^{\text{old}}$ respectively; target data stream $Y := X_d$

1: Model each data stream $X_i$ as a random variable.
2: Model each computational node $C_j$ with (input, output) data streams $(C^i, C^o)$ as conditional distributions.
3: Initialise a dictionary of deviations $F_{\text{dev}}$ with computational nodes as keys.
4: Initialise a dictionary of attribution scores $F_{\text{attr}}$ with data stream nodes as keys.
5: Initialise an empty queue of data stream nodes $Q$.
6: Suppose we observe a change in $Y$ measured as $\Delta Y = D(p(y)||q(y))$, where $D$ is KL divergence.
7: Push $Y$ to $Q$.
8: **while** $Q$ is not empty **do**
9: Pop top element of $Q$ into a variable $S$
10: Find computational nodes $C_j$ that contribute to $S$.
11: Compute deviation of $C_j$’s functionality:
   $\Delta C_j = \text{Deviation}(p(s|pa_s), q(s|pa_s))$
12: $F_{\text{dev}}[C_j] = \Delta C_j$
13: **for** each input data stream $X_i \in C_j$ **do**
14: Compute its contribution to the change in outcome:
   $F_{\text{attr}}[X_i] = \text{Attribution}(X_i, \Delta C_j, Y)$
15: Push $X_i$ to $Q$
16: **end for**
17: **end while**
18: **return** $F_{\text{dev}}, F_{\text{attr}}$

graphs: a data dependencies graph and a control dependencies graph. These graphs are then used as causal graphs to facilitate RCA. While this approach works well for a single program, their graph-building method does not scale to complex systems consisting of multiple programs and services exchanging API calls.

Causal fault localisation in FBP programs can offer a better experience. A readily available dataflow graph can be used as an SCM with every graph node assigned a causal conditional distribution $p(x_i|pa^G_i)$, where $x_i$ is its output data stream and $pa^G_i$ is its input data stream(s). Such conditional distribution contains the information on the graph node’s functionality and can be estimated directly via observing its (input and output) pairs. Change contribution can then be calculated according to Algorithm 1, and the largest deviation from independence will signify most likely faulty nodes. Moreover, the relative magnitude of the contributions can be used to quantify the uncertainty of the algorithm. In our scenario 1, Nadya was paged into a service latency issue that was caused by an upstream service. The troubleshooting process required engaging additional engineers and manual metric inspection, which was both slow and difficult. Instead, we envision an automated monitoring system that collects
operational metrics (e.g., each software component’s information encoded as $p(x_i | p_G^0)$) and detects distributional shift in an event of operational issue. This is followed by an automated change attribution calculation procedure that would allow to automatically discover the faulty service and only engage the engineer responsible for operating that service, saving time and effort for Nadya and her colleagues.

### 7.3.2 Business analysis

Business analysts also often require tools to identify root causes of unexpected shifts in business metrics and Key Performance Indicators (KPI) [Ambler, 2000, Carpi et al., 2017]. The process of answering such questions bears some resemblance to the technical troubleshooting discussed above, as business analysts often require an understanding of connections between input and output data. The main difference is their concern with business metrics and KPIs [Cadle et al., 2010]. Therefore analysts need to backtrack the effect of complex software computations for a given business metric and attribute its change to highlight which original input data factors contributed the most to the value of interest.

We envision an automated tool that, given the observed output, applies Algorithm 1 and returns the contribution of each input data stream. In our scenario 2 Arthur could use such a tool to calculate the effects of each of the suspected reasons on the observed drop in sales. The relative magnitude of the calculated effects could be used to understand how certain the estimates are and whether any additional investigation is required.

### 7.3.3 Experimentation

It is difficult to do efficient experimentation in large software systems. Isolated test environments with generated traffic can be an efficient way to verify the correctness of new changes before they are pushed to production. However such test environments have to be complete replicas of the production setting, which means engineers have to maintain two separate identical stacks. Unfortunately, in practice, test stacks tend to be de-prioritised and fall behind in deployment schedule [Struble, 2001], which eventually leads to situations where successful deployment on the test stack does not indicate issue-free deployment to production [Ramlar and Gmeiner, 2014, Page et al., 2008]. A/B testing on a live system is another popular approach to validate the effect of new changes. However, A/B tests are hard to design correctly [Olsson et al., 2017] and may have unforeseen side effects on the entire live system [Xu et al., 2015]. It is also an expensive practice, as it requires a large amount of traffic from a real user base to generate statistically valid results [Kohavi et al., 2022].

Causal inference on dataflow graphs offers an alternative solution to experimentation in software. Once developers have identified node(s) in the graph that are planned to be updated and are able to quantify expected changes in the output distribution of these nodes,
Figure 7.1: Dataflow graph of the Seldon Core v2 (SCv2) demo for processing car insurance claims. Underscores in component names are removed for readability. Green circles are processing nodes, red and white rectangles are data streams. Input and output streams are highlighted with red background and bold text.

post-intervention distributions across the entire system can be calculated with do-calculus. In scenario 3 Alisa could retrieve historical data for demand, measure the output distribution of the new demand prediction model, and estimate how the new model may affect other CoffeeFlow components.

To support our claims with practical evaluations, in the following sections, we provide multiple demonstrations of causal fault localisation in dataflow systems.

### 7.4 First example

We begin demonstrating the potential utility of causal inference for fault localisation in dataflow programs by showing how it can be used to detect an unintentional change of behaviour in one of the components, often referred to as a software bug.

The setting for this section is an application for processing insurance claims, already described in chapter 5 and re-implemented here with Seldon Core v2. As a brief reminder, the application processes car insurance claims by passing them through a series of data processing and decision-making steps, finishing with calculating the payout for each claimant. The dataflow graph of our implementation of this application can be seen in Figure 7.1. Code for this experiment can be found at https://github.com/apaleyeyes/dataflow-causal-graph.

As an intervention for this first experiment we create a bug in the system. Specifically, we change the behaviour of the component `classify_claim_complexity` that classifies low value insurance claims as simple or complex. The bug causes this component to classify all claims as simple, thus impacting distributions of all downstream data nodes including the output stream. The shift of the output data distribution is shown in Figure 7.2. We assume that the system
has some kind of monitoring in place that detects the shift and issues an alert. Our aim is to describe an automated causal fault detection procedure triggered by the alert.

To identify the location of the bug, we build the dataset of data passing through the application by collecting raw data from each stream before and after the intervention and run the causal attribution procedure described in Algorithm 1. We expect the classify_claim_complexity component to receive the highest attribution score, identifying this node as the reason for the observed output distribution shift. Table 7.1 shows the result of applying the causal attribution procedure to identify the fault location. We display attribution scores for each data stream node of the application, and we also convert them to probabilities for convenience of comparison.

We note that the causal attribution procedure correctly assigned the highest score to the simple_claims stream, which is an output stream of the classify_claim_complexity node. A developer seeing this information can immediately deduce that since no other nodes contribute to the simple_claims data stream, it is its producing node that must be the problem. Moreover, we notice that since the attribution scores are assigned to every node, this method provides a quantification of uncertainty: the developer can see how confident the procedure is, and if any additional root cause investigation is required.

---

Figure 7.2: Payout amount distribution in the insurance claims application before and after an intervention. In this case intervention is a bug introduced into the system. Payout is the final output of the claims processing pipeline, and detection of drift in its distribution shall trigger the fault localisation procedure.

---

5Detection of data distribution shifts is a separate topic that we consider to be outside the scope of this chapter. Interested readers can refer to Klaise et al. [2020] and Quiñonero-Candela et al. [2009].
Table 7.1: Example of attribution scores and corresponding probabilities for a single run of fault localisation experiment. The highest score and probability are highlighted in bold font.

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Score</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>0.0014</td>
<td>0.043</td>
</tr>
<tr>
<td>calculate_claim_value</td>
<td>0.0011</td>
<td>0.034</td>
</tr>
<tr>
<td>low_value_claims</td>
<td>0.0011</td>
<td>0.034</td>
</tr>
<tr>
<td>high_value_claims</td>
<td>0.0003</td>
<td>0.009</td>
</tr>
<tr>
<td>simple_claims</td>
<td><strong>0.0163</strong></td>
<td><strong>0.503</strong></td>
</tr>
<tr>
<td>complex_claims</td>
<td>-0.0081</td>
<td>0.249</td>
</tr>
<tr>
<td>calculate_simple_claim_payout</td>
<td>0.0031</td>
<td>0.095</td>
</tr>
<tr>
<td>calculate_complex_claim_payout</td>
<td>-0.0004</td>
<td>0.013</td>
</tr>
<tr>
<td>output</td>
<td>-0.0006</td>
<td>0.020</td>
</tr>
</tbody>
</table>

7.5 Demonstrations

In this section, we continue to demonstrate the causal fault localisation technique by showcasing experiments built with a variety of dataflow frameworks. We first discuss the frameworks themselves and then move on to the description of experiments. Code for all experiments can be found at https://github.com/apaleyes/causality-fbp, our implementation of the causal attribution procedure is done using DoWhy-GCM package [Blöbaum et al., 2022].

7.5.1 Dataflow frameworks

Here we give a brief overview of the dataflow frameworks used in our experiments, describing their general programming model and the features most relevant to our purposes.

Seldon Core v2 (SCv2) was presented in detail in the Chapter 6, thus here we only give a brief description of it. SCv2 is an open source framework for hosting machine learning models and running online and batch inference with them⁶. Despite its strong focus on MLOps infrastructure, SCv2 can also be considered a general dataflow engine. SCv2 describes pipelines (also called “inference graphs”) as a series of computations in Docker containers that communicate via Apache Kafka streams. Pipeline definitions are provided by the users in a form of YAML files with specifications of steps and connections. This computation model makes SCv2 decentralised and asynchronous by design [Akoush et al., 2022], while also making it possible to access all intermediate data states of each inference request via Kafka streams. To facilitate debugging, traceability and data provenance SCv2 provides an “inspect” CLI utility that allows users to retrieve data for an entire pipeline as well as for individual steps. We make extensive use of the ‘inspect” CLI to collect data for the experiments.

Node-RED is a flow-based visual programming tool particularly popular in the IoT space\textsuperscript{7}. Node-RED provides a lot of pre-made nodes that implement a large variety of functions, and users define the processing pipeline by placing nodes in the visual editor, specifying nodes’ properties and connections between them. If a custom function is desired, it can be implemented as a custom node in JavaScript programming language. The “msg Profiler” node in the Node-RED ecosystem leverages the flow-based nature of the Node-RED programming model to provide the end-to-end tracing information for each message flowing through the system\textsuperscript{8}. We make use of this node to retrieve necessary operational information for each step of the dataflow graph.

SciPipe [Lampa et al., 2019] is a Go package for scientific workflows\textsuperscript{9}. Workflows in SciPipe operate on files, with each workflow step being a command line expression that receives input file(s) and produces output file(s). The workflows themselves are described as Go programs. SciPipe’s design closely follows principles of flow-based programming [Morrison, 1994]. SciPipe puts a strong emphasis on provenance and creates separate audit files for each successfully executed workflow step. Audit files contain metadata, such as timestamps, inputs and outputs, and parent tasks. We make use of these audit files to discover connections between workflow steps, validate dataflow graph structure, and automate the analysis.

7.5.2 Experiments

We begin our evaluation of the causal fault localisation method with two experiments based on the insurance claims application described in the previous section, implemented with SCv2 and displayed in Figure 7.1. The first experiment involves the intervention in a form of a software bug and is already described in Section 7.4. In the second experiment, we intervene on the input data distribution, increasing the originally claimed amount by 50%. We expect the attribution procedure to ignore distribution changes in all intermediate nodes and assign the highest score to the input stream.

Our third experiment demonstrates that the causal fault localisation procedure can be used not only for business data but also for technical operational metrics. For that experiment, we use the seismic activity monitoring dashboard built with Node-RED on OpenEEW data\textsuperscript{10}. The dashboard allows a user to select time and location and then loads sensor data to examine for possible earthquake signals. Dashboard’s dataflow graph as seen in the Node-RED interface is shown in Figure 7.3. The dashboard has an “auto-load” feature that continuously loads new data, and allows us to measure the throughput of the pipeline. We define the throughput of the system (node) as the number of messages that pass through the system (node) in a fixed time period of 5 seconds. We then insert a random sleep delay between 0.5s and 1s in the

\textsuperscript{7}https://nodered.org/
\textsuperscript{8}https://github.com/bartbutenaers/node-red-contrib-msg-profiler
\textsuperscript{9}https://scipiipe.org/
\textsuperscript{10}https://github.com/openeew/openeew-nodered
Figure 7.3: Dataflow graph of the Node-RED demo, as seen in the Node-RED editor. The demo is an interactive dashboard for monitoring earthquake data, the majority of the nodes in the graph depicted here are responsible for UI elements and interaction with the user. The subgraph that is responsible for loading and analysing the data is highlighted with a thick black border and is the one we use in the experiment.
Build Quake Charts node, observe the drop in the end-to-end throughput and expect the causal attribution procedure to correctly identify the node responsible for the drop in performance.

Our final pair of experiments is designed to show that the causal fault localisation procedure can correctly identify drifting data stream among multiple input sources. For that, we are using the workflow implemented with SciPipe that computes the GC ratio across two segments of a DNA sequence. Two inputs are given to the workflow: the number of lines to collect from the first and second segments referred to as \texttt{count1} and \texttt{count2} respectively. The workflow then proceeds to extract the specified number of lines from each of the two segments, calculate AT and GC content in each, aggregate them and produce the final GC ratio. The dataflow graph of the workflow can be seen in Figure 7.4.

![Dataflow graph of the SciPipe demo for calculating GC ratio across two segments of a DNA sequence. This graph’s image is generated with the SciPipe API.](image)

To induce output drift in the workflow, in this example we intervene on the distribution of one of the input nodes. We expect our attribution procedure to identify the affected input source, ignoring the other input source and any interim workflow steps. Since there are two different input sources in this workflow, named \texttt{count1} and \texttt{count2}, we run two separate experiments for this demo.

Figure 7.5 displays the results of running the described experiments. Each experiment is repeated 20 times on randomised input to verify its robustness, and the figures report mean values and confidence intervals of attribution scores for each node in the dataflow graph of
an application. In all five experiments, the causal attribution approach was able to correctly identify the offending component of the corresponding dataflow program. We have also used Welch’s t-test [Welch, 1947] to verify that our results are statistically significant and observed $p < 0.01$ in all experiments.  

### 7.6 Limitations and next steps

Although in all our experiments causal inference was able to correctly localise the faulty node, we think this technique is not ready yet for adoption in production-ready applications. This chapter is merely the very first step towards the general availability of causal inference techniques in dataflow systems. For instance, more efforts are likely required to validate the scalability of this method. Graphs in our experiments do not exceed 20 nodes, while modern software can yield complex and large graphs with hundreds or even thousands of nodes [Musco et al., 2017]. Similarly, the method we use assumes that the dataflow graph is acyclic, while it is normal for complex software workflows to include loops. Thus an extension of Algorithm 1 for graphs with cycles is desirable. Further, it is necessary to study cases where data flowing through the graph represents not a single record but a dataset, a use case commonly seen in machine learning pipelines.

Multiple immediate next steps can be made from this starting point. One of the demonstrations described in Section 7.5.2 can be turned into an automated utility for fault localisation in the respective dataflow framework. Since all frameworks we used are fully open-sourced, that utility could become a part of the framework’s monitoring and alerting capabilities. Additionally, we postulated that causal inference can be used for business analysis and experimentation in dataflow systems (in addition to fault localisation we considered in the demos), and similar demonstrations are necessary to validate the viability of these claims.

The focus of this chapter is on the use of causal inference techniques in dataflow systems. Therefore our results are most relevant to systems built with dataflow frameworks, similar to Seldon Core v2, SciPipe or Node-RED. Our method relies on the fact that the dataflow frameworks we used provide developers with complete and explicit dataflow graphs of an entire system. However, it might be possible to apply the same techniques in cases where only approximate graphs can be recovered, for example where existing software not built with a dataflow approach is being analysed. A similar demonstration study on such software systems might be of interest to the community.

---

11The statistical significance is tested in a pairwise manner, testing null hypothesis for top scoring node against each other node in the system. Welch’s t-test is used because it supports distributions with unequal variances.
Figure 7.5: Attribution score distributions for all experiments across 20 repeats. For each node, we report the mean attribution score and confidence intervals. In all cases, the intervention node is correctly assigned the highest mean score. For all experiments we report $p < 0.01$ on Welch’s t-test for picking the correct offending node over any other node, supporting the statistical significance of our results.
7.7 Summary and Conclusions

This chapter draws the attention of the community to the connection between dataflow paradigm and causal graphs, and the potential implications this connection has for technical and business use cases. As a concrete use case, this chapter shows how causal inference on dataflow graphs can be used for fault localisation. We have implemented multiple demonstrations exploring various possible failure modes in dataflow systems and showed how the same causality-based technique can be used to locate these failures in different dataflow engines. The source code of all experiments is openly available, allowing researchers to replicate and extend our work.

In addition to the realisation of DOA principles, the connection to causal inference is another benefit of dataflow architecture that makes it a particularly potent approach for building AI systems. We hope that this chapter highlights an interesting opportunity to advance understanding of dataflow graphs in software and their connection to causality, leading to improvements in software development, maintenance and usage, ultimately resulting in systems that bear less intellectual debt.
Chapter 8

Conclusions

In this chapter, we conclude the dissertation with an overall summary, ideas for future research, and final remarks.

8.1 Summary of contributions

This thesis seeks ways to improve the process of deploying machine learning (ML) models in production systems through the lens of software architecture approaches.

Chapter 2 starts by surveying the existing reports on ML deployment, focusing on the issues described in these reports. Matching these issues against a typical ML deployment workflow, it is shown that every step in the deployment process is prone to give rise to challenging questions, many of which are related to current practices around managing data and models. It concludes that making modern software systems more data-focused shall yield great benefits for ML deployment.

Chapter 3 follows this suggestion by drawing the attention of the reader to Data-Oriented Architecture (DOA), a set of high-level principles that aim to build decentralised, open and flexible systems around data as their first priority. After summarising the existing works that formulate DOA principles, the chapter focuses on assessing the current state of DOA adoption in real-life deployments of ML applications. It is discovered that while DOA itself is relatively little known, its principles are already actively employed in existing ML systems. Furthermore, dataflow architecture is identified as an existing approach that embodies DOA principles, thus narrowing the focus of the remaining chapters to exploring the interplay between dataflow software and ML. To give the reader the required background, Chapter 4 briefly introduces dataflow architecture and related concepts.

Chapter 5 starts an exploration of dataflow in the context of ML deployment by comparing the process of deploying an ML model to an existing software application built with flow-based programming (FBP, a flavour of dataflow architecture with an additional layer of useful abstractions) and with service-oriented architecture (SOA, currently most prevalent approach
to software engineering). It is shown that despite lacking tooling for basic routine operations, FBP follows many of the DOA principles and has a lot of potential to improve the observability and availability of data in the system, as well as to reduce coupling between data-driven components. It is further noted that a real-life examination of these claims is required.

Chapter 6 followed that suggestion by presenting Seldon Core v2 (SCv2), an open-source dataflow platform for ML model inference. The design as well as implementation details are discussed, with a particular focus on how data communication via Apache Kafka streams is done. We show that SCv2 follows all DOA principles, and further validates the claims about the benefits of dataflow architecture for observability of data. Concretely, SCv2 provides users with flexible experimentation, monitoring and explainability features, all of which are vital properties for a MLOps platform. Through SCv2 the ideas described in this thesis are available in an industrial ML infrastructure framework.

In addition to data observability, the dataflow approach also can benefit our ability to understand causal relationships between its components and to troubleshoot software. Chapter 7 proves that by exposing a strong connection between dataflow graphs and causal inference and proposing an algorithm that computes change attributions in software dataflow graphs. To illustrate the utility of this idea, the proposed algorithm is used for causal fault localisation, and its efficiency is demonstrated using three modern dataflow frameworks (including SCv2). An ability to use causal inference on software without the need to make assumptions about confounding factors or the necessity to discover the causal graph post hoc is highlighted as an important way for software engineers to avoid accumulating intellectual debt in their systems.

8.2 Research agenda

Each chapter in this thesis ends with suggesting potential research ideas to extend the work described in that chapter. To avoid unnecessary repetition, in this section we will focus on a more holistic far-fetched vision of the research avenues for dataflow architecture and DOA in ML systems.

8.2.1 Systems Monitoring and Shadow Systems

Exposure of intermediate data steps and access to the dataflow graph open an opportunity to develop new practices around monitoring dataflow systems. A hypothetical approach that fits well with these key features is a network of shadow emulators or “shadow system”. A statistical emulator is a probabilistic surrogate model of a given process that can be trained in a data-efficient manner and allows to quantify uncertainty to inform decision-making [Paleyes et al., 2019, 2023b]. By exposing all intermediary data streams within the system, dataflow

\footnote{With the exception of chapters 1 and 4 that contain introductory material and do not describe novel contributions.}
architecture makes it possible to create and maintain a separate emulator (or a set of emulators) that corresponds to each component of the system. Crucially, because of the availability of intermediate processing data at arbitrary points, it is possible to pick any subsystem of choice (a subgraph of the entire dataflow graph) and emulate it. A network of such emulators acts as a shadow system, which is capable of measuring the gap in behaviour between the real world and the software system, monitoring and identifying fluctuations in data streams, and estimating the effects of potential changes.

To move the idea of a shadow system beyond a mere concept, two main research efforts are required. First, more efficient approaches and practices around the automatic fitting of surrogate models to software components are required. Existing work mostly focuses on auto-tuning of system parameters [Alabed and Yoneki, 2022, Dalibard et al., 2017] and has limited scalability potential. Thus, more case studies are needed that illustrate the use of shadow emulators as monitoring and explainability tools for software, as well as suggesting scalable ways of automatically building surrogate models of systems’ components. Second, innovation in the mathematical composition of individual emulators is required to build networks that can efficiently propagate uncertainty between components [Damianou and Lawrence, 2013]. While there is prior work on uncertainty propagation in software [Mishra and Trivedi, 2011], it still is not common to see interfaces and APIs that provide access or otherwise handle input or output uncertainty. A network of shadow emulators can provide visibility into data-related uncertainty within the system, thus enabling new research directions.

**8.2.2 End-to-end Systems Optimisation**

Optimisation is a ubiquitous problem in production environments, where developers and users often seek to find the best configuration of a certain hardware or software tool. With the ever-growing intricacy of modern software systems, their end-to-end optimisation becomes progressively more complex. There are many sources of such complexity. First, individual components within a system may have their own parameters affecting their behaviour, and the end-to-end optimisation process has to account for them, thus growing the input space. Second, isolated optimisation of a single component can have unforeseen downstream effects, thus emphasising the need for joint optimisation of all components taking into account their interactions [Zeng et al., 2016]. Third, developers have to deal with many, often conflicting, priorities, thus leading to the added complexity of multi-objective optimisation and Pareto front discovery [Avent et al., 2020, Ficiu et al., 2023]. Finally, large-scale systems can be expensive to execute, thus limiting the amount of time their performance with different parameter values can be observed.

While not eliminating all of these challenges completely, the dataflow approach, and the DOA paradigm in general, provide tools that can make them easier to tackle. The availability of intermediate data allows automated data-driven analysis of connections and interactions
between components. Networks of emulators described above allow the construction of a surrogate model of the entire system, thus making multi-objective Bayesian optimisation techniques applicable [Paleyes et al., 2022b]. Clarity of data dependencies between components can allow for causality-aware optimisation procedures [Aglietti et al., 2020]. High-quality surrogate models can also significantly reduce the need for running the real system and thus improve experimentation.

Another open research question around end-to-end optimisation of software systems lies in the area of “deep emulation” — propagation of uncertainty in hierarchical structures arising from dataflow graphs of systems, and the ability to combine multiple emulators that form a single network. New methods to represent uncertainties in hierarchical and multi-component systems are required, as well as the ability to evaluate explicit and implicit variational approximation techniques for deep structural learning.

8.2.3 Self-adaptive and Continual Learning Systems

Systems must deal with evolving requirements and unexpected failures when they are deployed in dynamic real-world environments. Self-adaptive systems address these challenges by sensing possible sources of changes in the environments and triggering adaptations of the system’s behaviour and configurations [Lalanda et al., 2013, Giese et al., 2013]. These systems use ML techniques (e.g. reinforcement learning) to predict possible changes in the environment and act accordingly [Gerasimou et al., 2019, Cabrera and Clarke, 2019]. One desired property of self-adaptive systems is the ability to learn new tasks without forgetting about the past ones. Continual learning (CL) systems have this ability as they focus on learning a large number of tasks without forgetting previous knowledge [Liu, 2017]. Besides the ability to adapt to new tasks, multiple additional applications of CL for systems are possible. ML-based systems suffer from hidden feedback loops [Sculley et al., 2015], and CL techniques can help mitigate these problems [Khritankov, 2021]. Real-life data contains outliers and exhibits sudden distribution shifts, which can be addressed with strategies proposed in the CL literature [Cai et al., 2021].

DOAs propose to create shared data models where systems’ historic data is fully available and traceable Cabrera et al. [2024]. This high data availability enables the building of CL and self-adaptive systems that can easily access systems’ past states. For example, Diethe et al. [2018] describes a reference architecture for self-adaptive systems, noting that these systems are capable of self-maintenance and therefore handling one of the biggest challenges modern software faces: evolving data. The suggested architecture is decentralised and stream-based, which are some of the core principles behind DOA. Self-learning systems are a step towards life-long learning [Silver et al., 2013, Liu, 2017], an ultimate goal of intelligent and autonomous systems research.
8.2.4 Systems FITness

There is a growing interest in understanding the impact of automated decision-making on individuals and communities. In parallel, there is a legislative effort to control and mitigate the potentially negative effects of such decision-making systems. In the area of ML, a lot of attention is given to the field of algorithmic fairness, which aims to understand the impact of ML models on different groups of population, and privacy, which aims to protect the data of individuals used for model training. Overall, the community is increasingly more concerned with FIT models - models that are Fair, Interpretable and Transparent. While these research efforts are commendable, they might be shifting the focus of the community towards standalone models. Crucially, ML systems include many components in addition to the models themselves. This brings forth important questions of the FITness of ML systems as a whole. While understanding the behaviour of a single model is important, it is equally important to understand the behaviour of the entire system and its effect on a particular individual or a group. This transition opens a range of research opportunities in understanding the FITness of software, such as analysis of complex interactions between individual components, propagating effects of data shifts, tracing a system output throughout the decision pipeline, system-wide counter-factual explanations [Wachter et al., 2017b].

Dataflow systems are ideally suited to address this shift of attention from FIT models to FIT systems, because of decomposability, data flow modelling and traceability they exhibit. Since components in dataflow software are loosely coupled and decentralised, any subset of connected components can be extracted and examined independently, with entire historical data of its inputs and outputs available for fairness and interpretation analysis. Such subsets can range from individual components, such as a model, to an entire system, providing engineers and analysts with flexibility. System FITness is closely related to legislative initiatives on data privacy and protection, such as GDPR or Equal Credit Opportunity Act (ECOA). Many existing works highlight dataflow as a key feature that allows systems to successfully fulfil these law requirements. Dataflow architecture exposes the flow of the system’s data by design, thus naturally providing support for such concepts as compliance by construction [Schwarzkopf et al., 2019], data provenance [Carata et al., 2014] and decision provenance [Singh et al., 2018].

8.2.5 Security and Privacy

The DOA paradigm advocates for open systems where entities are autonomous and can freely access shared data models [Miao et al., 2019, Wei et al., 2021, Pennekamp et al., 2019]. These data models store systems’ current and past states, which naturally raises questions regarding systems’ security and privacy [Tsai et al., 2021]. Malicious entities can access and modify systems’ data and behaviour at any time in such open environments. Depending on the application considered, data access and systems’ components may need to be restricted to
a specific set of users. For example, a healthcare management system needs to implement restrictive data access policies to avoid data privacy issues. The decentralisation principle can mitigate the security and privacy threats by storing and processing data in devices closer to end users (e.g., smartphones) [Shi et al., 2016, Tabatabaee Malazi et al., 2022, Cabrera et al., 2022]. However, managing authentication, permissions, and encryption keys in such a setup is challenging.

Different research efforts from the security community can be applied to DOA open setups to address security and privacy challenges. Homomorphic encryption [Fontaine and Galand, 2007] is an interesting direction for performing decentralised computations directly on encrypted data, without needing to provide the decryption key to the participant nodes. For example, a payment system might inquire about the validity of a transaction without having access to the underlying data (e.g. bank account number). Early deployments of zero-knowledge proof [Fiege et al., 1987] and homomorphic encryption are taking place in the industry [Blum et al., 2019]. This technology offers a solution to privacy issues in decentralised networks, but the field is still in its infancy. Algorithms developed are often computationally expensive, so further research is needed to make them more practical in resource-constrained devices. In addition to these technical advances, security and privacy issues also require authorities to develop novel initiatives and policy frameworks to keep up with advances in technology [Montgomery and Lawrence, 2021, Hardalupas, 2023].

8.3 Final remarks

The majority of software engineers are aware of the notion of technical debt. Those of us who had to support old production systems are painfully aware of it. It is well known that committing and accumulating technical debt is a long-term liability of any software project. But as our software becomes more complex, and includes more sophisticated components such as ML models, a notion of intellectual debt, mentioned multiple times throughout this thesis, grows in importance. This notion, proposed by Jonathan Zittrain, suggests that the more complex our systems become, the less we are able to explain and understand what they are doing. Our ability to develop systems whose behaviour cannot be fully understood or explained is disturbing. In Zittrain’s own words: “A world of knowledge without understanding becomes, to those of us living in it, a world without discernible cause and effect, and thus a world where we might become dependent on our own digital concierges to tell us what to do and when” [Zittrain, 2019]. In high-stakes decision-making domains, such as healthcare or finance, this is arguably more critical than technical debt — an incomprehensible working system is probably worse than a system that is explainable but hard to maintain. Therefore as a community we shall be seeking ways to build AI systems that are more, not less, transparent.

We believe that building systems that prioritise data, provide explicit access to their dataflow
graphs and apply causal inference to these graphs gives us an excellent opportunity to create software that is transparent and explainable by design. This is the main message behind the present thesis, and we hope to see more interest both among academics and industry professionals in the discussed techniques and paradigms.
Bibliography


Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the
dangers of stochastic parrots: Can language models be too big? In Proceedings of the 2021

Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taegyun Kim, Anirudh Sivaraman,
Cited on page 104.


Lucas Bernardi, Themistoklis Mavridis, and Pablo Estevez. 150 successful machine learning
tional Conference on Knowledge Discovery & Data Mining, pages 1743–1751, 2019.
Cited on page 28.

Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. Site reliability engineering:

Anoop Bhagyanath, Nadine Kercher, and Klaus Schneider. Allocation and scheduling of dataflow
graphs on hybrid dataflow/von Neumann architectures. Formal Methods and Models for
Codesign (MEMOCODE), Hamburg, Germany, 2023. Cited on page 82.

Umang Bhatt, Alice Xiang, Shubham Sharma, Adrian Weller, Ankur Taly, Yunhan Jia, Joydeep
Ghosh, Ruchir Puri, José MF Moura, and Peter Eckersley. Explainable machine learning in de-
ployment. In Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency,

Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine

Bernd Bischl, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek
Thomas, Theresa Ullmann, Marc Becker, Anne-Laure Boulesteix, et al. Hyperparameter

Patrick Blöbaum, Peter Götz, Kailash Budhathoki, Atalanti A Mastakouri, and Dominik Janzing.


171


176


Ben Stopford. The data dichotomy: Rethinking the way we treat data and services, 2016. Available at https://www.confluent.io/blog/data-dichotomy-


Mustafa Suleyman and Dominic King. The Information Commissioner, the Royal Free, and what we’ve learned, 2017. Cited on page 36.


tems and Service Management (ICSSSM), pages 1–6, 2016. doi: 10.1109/ICSSSM.2016.7538620. Cited on pages 56, 58, 60, 61, 62, 63, 64, 67, 68, 69, and 72.


