Department: Head
Editor: Name, xxxx@email

Architectural Design Decisions for the Machine Learning Workflow

S. J. Warnett and U. Zdun
University of Vienna, Faculty of Computer Science

***Abstract*—Bringing machine learning models to production is challenging as it is often fraught with uncertainty and confusion, partially due to the disparity between software engineering and machine learning practices, but also due to knowledge gaps on the level of the individual practitioner. We conducted a qualitative investigation into the architectural decisions faced by practitioners as documented in gray literature based on Straussian Grounded Theory and modeled current practices in machine learning. Our novel Architectural Design Decision model is based on current practitioner understanding of the topic and helps bridge the gap between science and practice, foster scientific understanding of the subject, and support practitioners via the integration and consolidation of the myriad decisions they face. We describe a subset of the Architectural Design Decisions that were modeled, discuss uses for the model, and outline areas in which further research may be pursued.**

Introduction

Software engineering (SE), software architecture (SA), and machine learning (ML) are established disciplines, but a noticeable mismatch between engineering and architectural practices on one side and more data-focused activities on the other is evident, despite the latter often constituting complex software solutions. As a consequence, a rift has formed between the SE and ML communities [1]. This apparent discrepancy could partially be explained by acknowledging the fact that, generally speaking, ML developers are not typically from a software engineering background and thus do not routinely apply common engineering and architectural techniques. Conversely, software engineers and architects do not typically work as ML engineers, and so do not regularly concern themselves with developing appropriate solutions for ML systems [2, 3].

The issues resulting from the divergence of these disciplines are further compounded by the technical challenges customarily arising in ML projects that are generally non-existent in non-ML projects, such as the development and maintenance of ML solutions [4, 5, 6].

To address these issues, we conducted an empirical gray literature study [7] based on Straussian Grounded Theory (GT) [8, 9]. We aimed

to identify relevant architectural concepts applied by practitioners in ML data processing, model building, and Automated Machine Learning (AutoML). The result is a model of Architectural Design Decisions (ADDs), decision options, considerations, practices, and their relations.

In this article, we briefly describe a simplified ML pipeline workflow to illustrate our research context. We then describe our research methods and study design. Next, related studies in the field are discussed. After that, we present the ADDs resulting from our study. Finally, we conclude by briefly outlining future directions and further research possibilities based on our work.

The Machine Learning Workflow

To illustrate the parts of the ML architecture we consider in this article, Figure 1 depicts a typical ML workflow. An ML pipeline generally starts via a triggering event – perhaps manually or following a commit to the code base. This initiates the data ingestion step, where the data sets are updated, e.g. by fetching the data afresh from a repository or using data received via a stream. The data is then processed in various ways and is typically transformed (e.g. normalized then cleaned) before features are extracted. The features and processed data are then passed to the model building components, in which models are trained. The trained models are then deployed, and can subsequently be used for predictions.

Research Methods and Study Design

We studied methods and techniques currently applied by practitioners in the context of ML solution development and gained valuable insights into the software engineering and architectural state of the art as applied to ML. The study was based on Straussian GT [8, 9] and involved the qualitative analysis of gray literature [7]. We describe our research method in detail in our prior work [10]. We aimed to identify relevant ADDs, options, considerations, practices, and their relations and strove to reduce the gap between science and practice and between SA/SE and ML.

Initially, we used search engines (e.g. Google, StartPage, DuckDuckGo) and topic portals (e.g. InfoQ, DZone) to search for practitioner sources consisting of blog posts, presentations, and videos. Sources were considered if they were relevant and weren't marketing a business or product. The authors checked each other's selection for suitability.

We open-coded each source in depth and performed axial and selective coding steps, in which we formally modeled each ADD, ADD relation, and decision driver in Python-based models and realized automated generators for UML models in PlantUML. Further sources were actively sought out based on preceding findings during the coding process. This involved continuous consideration of the topics needing to be coded and their potential contribution to the model. We continued coding sources until theoretical saturation (i.e. "the point in category development at which no new properties, dimensions, or relationships emerge during analysis" [9]) was reached. The result was a formal UML-based ADD model covering decision options, the relations between them, and relevant decision drivers.

Twenty-nine sources were considered in total, and these are listed in Table 1. Due to space constraints, we only describe a subset of the ADDs that were modeled in this article. Aspects such as deployment, continuous integration and delivery, MLOps, and development environments are omitted. To support the reproducibility of our study and provide the full models, we offer the detailed coding data and the resulting models, as well as our Python implementation to generate UML diagrams and tables from the models, in a long-term online archive [11].

Sidebar: Related Studies on the Relations of SE/SA and ML

Some headway has already been made in the field of SE/SA for ML. The systematic literature review conducted by Washizaki et al. [12] provides a comprehensive, categorized collection of software (anti-)patterns for ML, and several other publications [4, 5, 13] document the challenges and open research topics in the field.

Lwakatare et al. [5] conducted an empirical investigation into software engineering challenges for ML systems and propose a taxonomy of issues that includes data dependency management, deployment difficulties as well as model and result reproduction challenges.

Sculley et al. [4] describe the significant technical debt incurred when attempting to maintain

Table 1. List of Knowledge Sources Included in the Paper

ID	Title	Archive URL	Source Type	Example	Source Code
s1	How to power up your product by machine learning with python microservice, pt. 1	https://tinyurl.com/ml-adds-u1	Practitioner Audience Article	True	False
s2	Architecting a Machine Learning Pipeline: How to build scalable Machine Learning systems - Part 2/2	https://tinyurl.com/ml-adds-u2	Practitioner Audience Article	True	False
s3	Some Thoughts on Modularization in Machine Learning	https://tinyurl.com/ml-adds-u3	Practitioner Audience Article	False	False
s4	Productionizing Machine Learning with a Microservices Architecture	https://tinyurl.com/ml-adds-u4	Presentation Video	False	False
s5	Microservices Suck for Machine Learning (and what we did about it)	https://tinyurl.com/ml-adds-u5	Practitioner Audience Article	True	True
s6	MLOps: Continuous delivery and automation pipelines in machine learning	https://tinyurl.com/ml-adds-u6	Practitioner Audience Article	True	False
s7	Composing Deep-Learning Microservices for the Hybrid Internet of Things	https://tinyurl.com/ml-adds-u7	Practitioner Audience Article	True	False
s8	Continuous Intelligence: Moving Machine Learning Application into Production Reliably	https://tinyurl.com/ml-adds-u8	Slides	True	False
s9	Architecture of a real-world Machine Learning system	https://tinyurl.com/ml-adds-u9	Practitioner Audience Article	True	False
s10	Architecting a Machine Learning System for Risk	https://tinyurl.com/ml-adds-u10	Practitioner Audience Article	True	False
s11	Architecting a Scalable Real Time Learning System	https://tinyurl.com/ml-adds-u11	Practitioner Audience Article	True	False
s12	System Architectures for Personalization and Recommendation	https://tinyurl.com/ml-adds-u12	Practitioner Audience Article	True	False
s13	Architectural thinking in the Wild West of data science	https://tinyurl.com/ml-adds-u13	Practitioner Audience Article	True	False
s14	Machine Learning Architecture: The Core Components	https://tinyurl.com/ml-adds-u14	Practitioner Audience Article	False	False
s15	Scalable Software and Big Data Architecture - Big Data and Analytics Architectural Patterns	https://tinyurl.com/ml-adds-u15	Practitioner Audience Article	False	False
s16	Machine Learning in Production: Software Architecture	https://tinyurl.com/ml-adds-u16	Practitioner Audience Article	True	False
s17	AutoML	https://tinyurl.com/ml-adds-u17	Practitioner Audience Article	False	False
s18	AutoML is Overhyped	https://tinyurl.com/ml-adds-u18	Practitioner Audience Article	True	False
s19	Three Levels of ML Software	https://tinyurl.com/ml-adds-u19	Practitioner Audience Article	True	False
s20	MLOps: Methods and Tools of DevOps for Machine Learning	https://tinyurl.com/ml-adds-u20	Practitioner Audience Article	False	False
s21	MLOps: What It Is, Why it Matters, and How To Implement It (from a Data Scientist Perspective)	https://tinyurl.com/ml-adds-u21	Practitioner Audience Article	False	False
s22	MLOps Principles	https://tinyurl.com/ml-adds-u22	Practitioner Audience Article	False	False
s23	Machine Learning Monitoring: What It Is, and What We Are Missing	https://tinyurl.com/ml-adds-u23	Practitioner Audience Article	False	False
s24	Automated monitoring of your machine learning models with Amazon SageMaker Model Monitor and sending predictions to human review workflows using Amazon A2I	https://tinyurl.com/ml-adds-u24	Blog Post	False	False
s25	MLOps: Model management, deployment, and monitoring with Azure Machine Learning	https://tinyurl.com/ml-adds-u25	Practitioner Audience Article	False	False
s26	The Pros and Cons of Using Jupyter Notebooks as Your Editor for Data Science Work TL;DR: PyCharm's probably better	https://tinyurl.com/ml-adds-u26	Practitioner Audience Article	False	False
s27	10 reasons why data scientists love Jupyter notebooks	https://tinyurl.com/ml-adds-u27	Practitioner Audience Article	False	False
s28	5 reasons why jupyter notebooks suck	https://tinyurl.com/ml-adds-u28	Practitioner Audience Article	False	False
s29	Jupyter Notebook is the Cancer of ML Engineering	https://tinyurl.com/ml-adds-u29	Practitioner Audience Article	False	False

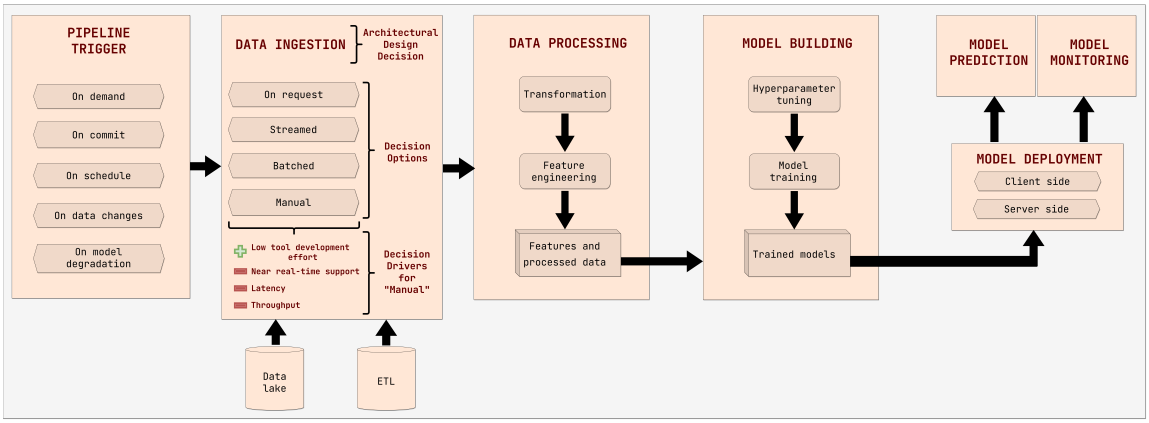

Figure 1. A simplified ML pipeline.

real-world ML systems. Such risk factors include entanglement, data dependencies, configuration issues, reproducibility debt, and system-level anti-patterns.

Bosch et al. [13] provide an overview of the software engineering challenges associated with ML solutions, including model management, reuse and deployment, data pipeline challenges, monitoring and logging, design methods, and data quality management. They also identify open research areas, such as distributed model creation, distributed data storage, data generation, and automated experimentation.

Nascimento et al. [6] conducted a comprehensive systematic literature review on the subject of software engineering for artificial intelligence (AI) and ML. They summarize the state of the art and identify various unsolved challenges in testing, AI software quality, and data management. They also identify several other topics such as architecture design, AI engineering, model development, model deployment, integration, infrastructure, operation support, requirements engineering, project management and education.

Serban et al. [14] mined academic and gray literature, identifying twenty-nine engineering best practices for ML applications. They also surveyed practitioners to determine the degree of adoption of these practices and validate their perceived effects. Their findings helped improve understanding of the relationship between team size and practice adoption, and the relative adoption rate of SE practices compared to ML-specific practices.

Martínez-Fernández et al. [15] conducted a systematic mapping study, considering 248 studies, and identified and classified various SE approaches for AI-based systems, identifying prevalent and neglected areas of study.

Alves et al. [16] identified existing product engineering methods and practices for industrial ML applications and platforms. They conducted a Gray Literature Review to investigate methods and practices applied to ML product lifecycles.

In contrast to the state-of-the-art, our approach was to formally model real-world practices to assist practitioners in finding suitable options to common design decisions and mitigate or avoid challenges common to the field. Our ADDs could potentially be used to assess uncertainty or complexity of the ML system design space, assess architectural conformance or identify anti-patterns.

Architectural Design Decisions

Table 2 presents an overview of the various ADDs, the decision options, gray literature sources, and decision drivers discussed in this article. We describe each of the ADDs in this section. Our replication package contains detailed UML models of all ADDs and relations described here.

Data processing

Data processing involves ingesting raw data into an ML workflow and transforming it into

Table 2. Study Results: Overview of Design Decisions, Decision Options, Evidences and Related Forces

Design Decision	#	Decision Option (Solution)	Evidences (from Practitioner Sources)	Decision Drivers (Forces)
How to automatically process the data used for model building?	16	1. No data processing automation	s1, s6, s13	f1(--), f2(-), f3(-), f4(--), f5(+)
		2. Data pipeline	s1, s2, s4, s5, s6, s8, s9, s13, s14, s15, s16, s17, s18, s19	f1(++), f2(++), f3(+), f4(+), f5(-)
		3. ETL pipeline	s4, s5, s9, s13, s15	f1(+), f2(+), f3(o), f4(+), f5(-)
		4. Data processing component	s9, s13, s14	f1(+), f2(+), f3(-), f4(o), f5(-)
Which data processing tasks can be performed by a data processing pipeline or component?	20	1. Data extraction	s1, s2, s6, s8, s13, s14, s15	∅
		2. Data transformation	s2, s5, s6, s10, s13, s14, s15, s19	∅
		3. Data preparation	s1, s2, s4, s5, s6, s9, s10, s11, s13, s14, s15, s17, s18, s19, s20, s23, s24, s25	∅
		4. Data validation	s6, s9, s13, s19, s20, s24	∅
		5. Data selection	s2, s6, s14	∅
		6. Feature engineering	s1, s2, s4, s5, s6, s8, s9, s10, s13, s14, s15, s17, s18, s19, s23, s25	∅
		7. Data processing hyperparameter tuning	s9, s11, s17, s18	∅
How to persist and provide access to features?	10	1. Store features in data stores without specific support	s2, s4, s5, s6, s13	f6(-), f7(-), f8(-), f9(-), f10(o), f11(o), f12(o), f13(o), f14(o)
		2. Feature store	s2, s4, s5, s6, s9, s10, s11, s13, s15	f6(+), f7(+), f8(+), f9(+), f10(+), f11(+), f12(+), f13(+), f14(+)
Should data be processed in batches or in realtime?	10	1. Batch-based data processing	s1, s2, s5, s6, s8, s10, s13, s15	f15(++), f12(++), f13(-), f14(--), f10(-), f11(+)
		2. Real-time, stream-based data processing	s1, s2, s4, s5, s6, s8, s9, s10, s13, s15	f15(+), f12(+), f13(++), f14(++), f10(++), f11(o)
How to ingest data into ML projects or applications?	16	1. Streamed data ingestion	s2, s4, s5, s7, s9, s11, s13, s14, s15, s19	f10(+), f11(-), f12(+), f13(++), f14(++), f5(-)
		2. Data ingestion by request	s2, s4, s5, s9, s13, s14, s15, s19	f10(o), f11(+), f12(o), f13(o), f14(+), f5(-)
		3. Data ingestion in batches	s4, s5, s13, s14, s15, s19	f10(-), f11(o), f12(++), f13(-), f14(--), f5(-)
		4. Manual data ingestion	s4, s5, s6	f10(--), f11(-), f12(--), f13(--), f14(--), f5(+)
How to trigger a machine learning pipeline or orchestrator?	11	1. On-demand trigger	s2, s4, s6, s9, s12	∅
		2. On-commit trigger	s4, s8, s25	∅
		3. On-schedule trigger	s1, s2, s6, s9, s12, s14	∅
		4. On availability of new training data trigger	s6	∅
		5. On model performance degradation trigger	s6	∅
		6. On changes in the data distribution trigger	s6	∅
How to perform model building in an ML project?	13	1. Model building in development tool	s2, s4, s6, s10, s13, s15	f16(-), f17(+), f13(--), f18(-), f19(--), f3(+), f20(-), f21(-), f4(-), f22(-), f23(-), f5(+)
		2. Model building pipeline	s2, s4, s6, s8, s9, s10, s13, s14, s17, s19, s25	f16(+), f17(-), f13(+), f18(++), f19(+), f3(o), f20(+), f21(+), f4(+), f22(+), f23(+), f5(-)
		3. Model builder component	s9, s11, s13, s14	f16(+), f17(-), f13(+), f18(o), f19(o), f3(o), f20(o), f21(+), f4(o), f22(o), f23(-), f5(-)
Which tasks can be performed by a model building pipeline or component?	22	1. Model training	s1, s2, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s17, s18, s19, s20, s23, s24, s25	∅
		2. Data splitting	s2, s4, s8, s9, s14, s19	∅
		3. Data checkpoints	s2	∅
		4. Model validation	s4, s6, s8, s9, s10, s13, s14, s17, s18, s19, s20, s25	∅
		5. Model selection	s2, s4, s9, s11, s14, s17, s18, s19	∅
		6. Train multiple model versions with different parameters and/or algorithms	s2, s4, s6, s8, s9, s11, s19	∅
		7. Model packaging	s9, s19	∅
		8. Model hyperparameter tuning	s2, s6, s9, s11, s17, s18, s19, s24, s25	∅
		9. Development tool facade	s2, s4, s6	∅
		10. Development tool export	s10, s15	∅
When and how to train the model?	7	1. Batch-based learning	s2, s6, s10, s11, s12, s19	f24(--), f25(--), f26(++), f10(-), f11(+), f27(--), f14(-), f19(+), f28(++)
		2. Incremental learning	s6, s10, s11, s12, s19	f24(++), f25(++), f26(--), f10(+), f11(-), f27(+), f14(+), f19(-), f28(-)
		3. Hybrid batch-based and incremental learning	s12, s19	f24(+), f25(+), f26(+), f10(+), f11(+), f27(o), f14(o), f19(+), f28(--)
Should AutoML be used and if so where?	9	1. No AutoML	s4, s6, s9, s10, s17, s18, s19, s22, s23	f4(o), f29(o), f30(o), f22(o), f5(o), f2(o), f20(+)
		2. AutoML	s4, s6, s9, s10, s17, s18, s19, s22, s23	f4(+), f29(+), f30(+), f22(+), f5(o), f2(+), f20(-)

Forces Codes/Sources: **f1**: Automated data collection [s1], **f2**: Process and work automation [s1, s8, s15, s18, s20], **f3**: Iterative development [s1, s4], **f4**: Reproducibility [s6, s8, s13, s17, s20], **f5**: Tool development effort [s1, s2, s10, s17], **f6**: Feature reuse [s5, s6], **f7**: Feature discovery [s5, s6], **f8**: Maintainability [s1, s5, s6, s13, s16], **f9**: Avoid training/serving skew [s6, s8], **f10**: Online support [s2, s6, s9, s10, s11, s12, s13, s19], **f11**: Offline support [s2, s6, s9, s10, s11, s12, s13, s19], **f12**: Data throughput [s2, s4, s6, s9, s12, s15], **f13**: Low latency [s2, s4, s6, s15, s16], **f14**: Near real-time support [s1, s2, s4, s6, s10, s11, s12, s13, s15, s19], **f15**: Reliability [s2, s5, s8, s10], **f16**: Interchangeability of machine learning algorithms [s2], **f17**: Interactive prototyping [s2, s13], **f18**: Loose coupling [s2, s5, s6, s7, s16, s19], **f19**: Scalability [s1, s2, s3, s4, s5, s7, s11, s12, s13, s15], **f20**: Production-ready development [s4, s13, s16, s18, s19], **f21**: Experimental operational symmetry [s6, s13], **f22**: Development agility [s10, s13, s17], **f23**: Flexibility [s10], **f24**: React to unforeseen changes in the data [s11, s19], **f25**: Memory requirements [s11], **f26**: Handling massive amounts of data [s11, s12, s15], **f27**: Speed performance [s2, s3, s4, s5, s6, s9, s10, s11, s15], **f28**: Variety of machine learning algorithm choices [s11, s12, s19], **f29**: Explainability [s17], **f30**: Development velocity [s5, s17, s18, s19]

Note on Empty Forces(∅): Please note that some ADDs do not report decision drivers as they are sub-decisions of the respective prior ADD, and share the same forces (see explanations in the remainder of the article).

usable data that can be further processed by ML algorithms. Data is central to ML, and the main benefits of an ML approach lie in the processing of large volumes of it at scale [17]. Typical phases of the data processing task involve data selection, transformation, and output, as well as the generation, storage, and versioning of data artifacts such as features and data subset samples [5].

As with many other ML tasks, data processing can be performed manually or in an automated fashion, with the latter yielding many potential benefits. The data processing stage also presents various data collection options. Once again, this is a task that may be automated. Data may be ingested manually, in real-time (e.g. streamed), batched, online, or offline with various implications for latency, throughput, reliability, and so on – the various options are discussed below. Data labeling (e.g. for classification tasks) is another activity that may be performed manually or automated and we consider the implications. Generated artifacts, such as features, may be stored in various ways and each option has different implications when it comes to their respective benefits. Finally, the data processing stage needs to be triggered in one way or another. Again, there is a multitude of options to choose from, including triggering on-demand, on-commit, on-schedule, on new data availability, on model performance degradation, or on changes to the data distribution.

Data processing automation decision

A core ADD is *how to automatically process the data used for model building*. A simple option is to provide *no data processing automation*.

The most common automated architecture proposed by practitioners in our sources is a *data pipeline*, i.e. to perform data processing in a dedicated, flexible pipeline. A similar option is using an *ETL pipeline* which is a kind of data pipeline in which a specific selection and order of steps are used: extract, transform, and load. While pipeline architectures are often proposed, proposals for *data processing components* that do not follow a pipeline architecture also exist.

The primary benefits of these options are supporting *automated data collection* and a potentially high degree of *process and work automation*. These, in turn, may lead to faster data ingestion and more rapid model training turnaround times, more frequent deployments, a reduction in human error, an improvement in *iterative development* and *reproducibility*, not to mention the resulting cost savings of all of the above. Our sources indicated that the *data pipeline* is the most *flexible* of the automated options. These options require some *tool development effort* which potentially represents an overhead – especially in the initial phases of a project.

Data processing tasks decision

If a *data processing pipeline* or *data processing component* is chosen, the tasks to be performed during automated data processing can be decided in a subsequent decision. Common tasks performed in automated data processing flows are *extraction*, *transformation*, *preparation*, *validation*, *selection*, and *feature engineering*.

Data processing hyperparameter tuning – which refers to the tuning of hyperparameters during automated data processing – is a less frequently applied method. For example, if an automated featurizer is utilized during feature engineering, this component has hyperparameters that can be tuned. Automation is again possible, e.g. in AutoML (see the discussion on AutoML below).

Feature persistence and access decision

The *data processing tasks* decision and its *feature engineering* option generate the features in ML. It is often important to provide persistence and access for these features – frequently beyond the current run of data processing and model building. One option to achieve this is to *store features in a data store without specific support* for features. The alternative option is a dedicated *feature store*, in which features are accessible for training following standards for definition, storage, and access. To facilitate access to the features, the feature store can include either a *batch feature API* or a *real-time feature API*, or both. Some practitioners also suggest using event-driven abstractions here, e.g. using *event sourcing* as a technique to realize the feature store.

Compared to the alternative, a dedicated *feature store* facilitates easier *feature reuse*, *feature discovery*, and *maintainability*. It also helps avoid *training-serving skew*, i.e. making sure that the features used for training are the same as the

ones used during serving. If dedicated APIs are supported, it can additionally offer better support for *offline* and/or *online* training and serving. Finally, those APIs can be optimized for *data throughput* (e.g. when large volumes of data or a large number of clients are expected), *low latency* (e.g. in safety-critical systems), or *near real-time support* (e.g. when user interaction is intended).

Data processing in batches or in real-time decision

Automated data processing can occur either in batches or in real-time. Thus, *batch-based data processing* and *real-time, stream-based data processing* are two options of this follow-on decision to be taken for all automated options of the data processing automation decision.

Real-time, stream-based data processing is especially beneficial for *online* data processing, and, unlike *offline* data processing, is beneficial for *low latency* and *near real-time* processing, e.g. when the practitioner wishes to gain fast insights, or when a system needs to be able to react instantly to events at runtime. While modern online architectures offer high *data throughput* and *reliability*, the extra processing steps at runtime make them inferior to offline architectures for those forces. *Offline* data processing is well-supported by *batch-based data processing*, for instance, when processing high volumes of data, performing particularly deep data analyses, or when processing speed does not take precedence.

Data ingestion decision

Another central ADD is how to ingest data into ML projects or applications to facilitate data processing and consequently model building.

An initial option is to *ingest data manually*. Alternatively, automated options exist, such as *data ingestion in batches* and *streamed data ingestion*. One can also actively acquire raw data from sources via *data ingestion by request*. It is customary to store the raw data that is ingested and this is typically done in a *raw data store*. The automated options can be abstracted and orchestrated in a dedicated *data ingestor* component, with the two variants *data ingestor queue* (a data ingestor that queues up ingested data, e.g. for online, incremental learning) and *ground-truth collector* (a data ingestor collecting data on predictions in production, e.g. with input from users when predictions are wrong). The data ingestor can be used in the context of data labeling. For example, a ground-truth collector can support an *automatic data labeler* which is used instead of *manual data labeling*.

Data ingestion needed for experimental or *offline* learning is often performed using *data ingestion in batches* or by *manual data ingestion*. *Streamed data ingestion* and *data ingestion by request* are suitable for *online* learning or ingesting data online into the ML-based application. *Streamed data ingestion* is ideal for continuously incoming data and offers relatively high *data throughput*, *low latency*, and *near-real-time support*. *Data ingestion by request* can receive specific data as needed, but this increases latency and is usually not best suited for reaching high *data throughput* as each new item of data requires an additional request. *Data ingestion in batches* offers a very high data throughput, but is *offline*, and therefore not well suited to data ingestion in model training or applications that require *low latency* or *near real-time support*. *Manual data ingestion* has even more negative impacts on those forces. Its main advantage is that it does not require *tool development effort* for engineering a custom automated ingestion solution.

Trigger pipelines or ML orchestrator decision

ML components, such as data processing pipelines, ML orchestrators, and model building components, can be triggered at various points. For each of these, we are required to decide on how they should be triggered. The first option is an *on-demand trigger* either by an ad-hoc, manual execution of a pipeline or component (e.g. by a human operator) or via a call coming from another pipeline or component.

Another option is an *on-commit trigger* which can start a pipeline when changes are made in a version control system. Some pipelines or components need to run at regular intervals, e.g. daily, weekly, or monthly, in which case an *on-schedule trigger* is used.

Triggering can also occur *on the availability of new training data*. Further, pipelines can be triggered by *model performance degradation* or *changes in the data distribution*. Each of these options requires monitoring of the model or data respectively.

Model building

Building an ML model involves applying a learning algorithm to features of a pre-prepared data set to yield predictions of a specific type, such as qualitative (classification) or quantitative (regression) values.

Hyperparameter tuning is the practice of adjusting values used by ML algorithms to optimize their performance metrics. This task may be automated or manual.

The actual work involved in model building may be performed using tools, such as an integrated development environment (IDE) or computational notebook, or automated using AutoML or a model building pipeline.

These approaches each have their respective advantages and disadvantages and are discussed in the following sections.

Model building decision

A core ADD is *how to perform model building in an ML project*. Many practitioner sources claim that providing some level of automation is key to successful production delivery and that many ML projects never progress beyond the experimental phase. Very often the *model building is performed using tools*, such as computational notebooks or IDEs. The main benefits of *model building using a tool* are that this option better supports *interactive prototyping* and *iterative development* than the other options. This option also offers low *tool development effort*, since the tools don't have to be engineered from scratch.

While this is useful for development and experimentation, a *model building pipeline* for projects that are intended for delivery to production is recommended due to the higher degree of automation. While pipeline abstractions are common for this automation task, some practitioners also suggest other kinds of *model builder components*, such as one architectured in a pipeline-based style or a dedicated *ML orchestrator* which coordinates disparate components at the next higher level of abstraction.

Automated model building enables *interchangeability of ML algorithms*, *low latency* in the model building process, *symmetry between experimental and operational tasks* (in the sense that the same software components are used for both pre-production and production), *reproducibility* of each step in the model building phase, and enhanced support for *production-ready development*. The main drawback is increased *tool development effort*.

Our sources indicate that, when comparing the two automation options above, the pipeline abstraction is usually preferred, since it is a highly *flexible* architecture that supports a greater level of *development agility*. Pipelines are also known to be *scalable* and *loosely coupled*.

Model building tasks decision

If a *model building pipeline* or *model builder component* is chosen, the tasks to be performed during automated model building can be decided in a subsequent decision. Common tasks typically performed in such components are *data splitting* (e.g. into training and test data), *model training*, *model validation*, and automated or semi-automated *model selection* if multiple models were built. *Model selection* is usually needed if the practice of *training multiple model versions with different parameters and/or algorithms* is applied. *Model hyperparameter tuning* and optimization describes the challenge of selecting optimal hyperparameters for the learning algorithm(s). Some approaches automate *hyperparameter tuning* for the model (e.g. in AutoML) as a model building subtask. Our sources describe the tasks of *model packaging* (e.g. in storage or exchange formats). To enable model training with error tolerance, *data checkpoints* can be defined to enable retraining if a previous attempt failed due to a transient issue (e.g. a timeout). Some automated pipelines or components do not only use development tools in early, experimental stages but also integrate them in the automated flows. This can either be done using a *development tool export* or more elegantly using a *development tool facade* component integrated into the automated flow. This way, the "best of both worlds" can be achieved.

When and how to train the model decision

The *model training* task entails a follow-on decision on *when and how to train the model*. Model training can occur via *batch-based learning* or by an *incremental learning* approach. Alternatively, *hybrid batch-based and incremental learning* is possible, where some factors of the model are trained in batches and others incrementally.

Significant benefits of *batch-based learning* lie in *offline* training, especially when *massive amounts of data* are to be handled. Thus, it usually has high *memory requirements* and rather poor *speed performance*, leading to precomputation, which is beneficial for *scalability* (provided incremental updates are not needed).

Incremental learning has the opposite effect on those forces. Thus, it is rather applicable when a *reaction to unforeseen changes* or the *near real-time support* for model training is required. A downside is that the *variety of ML algorithm choices* supporting *incremental learning* is limited.

AutoML

AutoML is a relatively recent and trending practice in which many of the tasks usually performed manually by ML engineers, such as data preparation, feature engineering, model training, hyperparameter tuning, model validation, and selection, are automated. To achieve this, AutoML provides search algorithms for finding the optimal solutions for those tasks in the ML pipeline.

AutoML decision

The AutoML decision consists of two parts:

1) *whether AutoML should be applied*, and,
2) if so, to further decide *which practices should be automated*.

In particular, AutoML can automatically preprocess the data during *data preparation*. It can be used while *feature engineering* to autogenerate new features and select meaningful ones. Throughout *model training*, it can be applied to automatically train models, maybe using different parameters and/or ML algorithms by applying automated *hyperparameter tuning* of data processing and model building components and then make an automated *model selection*, e.g. based on ML or domain-based metrics. In this context, it can use *model validation* to score the models.

The goals of applying AutoML are e.g. requiring less manual *tool development effort*, and thus supporting higher *development velocity* and *agility*. This is achieved through increased *process and work automation*, which also offers two further key benefits: via automation, AutoML offers a greater level of *explainability* and *reproducibility* in the data processing and model building tasks, as every step follows precisely-specified algorithms. Unfortunately, existing AutoML solutions do not always deliver results to the desired quality standards, which often results in a negative impact on *production-ready development*, potentially leading to additional *development effort*. Extra *tool development effort* for engineering the AutoML solution would also be required in this case.

Discussion: Lessons Learned, Limitations and Threats to Validity

Modeling ADDs as performed in this study yields numerous benefits. In particular, the ADD model can be used to assist scientists in their understanding of practitioners' needs and the challenges they face, in addition to guiding architectural decision-making on the part of practitioners based on existing practices. The study and its resulting model also open up new avenues for exploration and further research in this domain.

One topic that is mentioned in several sources (s4, s6, s13, s21, and s26) is security, which appears in the general context of operations and infrastructure (e.g. serverless, enterprise infrastructure, and fully-managed platforms) rather than the ML workflow, and thus falls outside the scope of this article.

The validity of the study is premised on the correct application of GT as well as consistent and correct coding practices. We have endeavored throughout to apply the method correctly (as documented in our replication package), but mistakes cannot be fully excluded. Validity may also potentially be threatened by unreliable sources or by neglecting to include relevant sources. Since we continued coding sources until theoretical saturation was reached, this increased the likelihood that a fair representation of the literature was achieved, and the risk that relevant sources were not considered was reduced.

Bias poses a potentially significant threat to any study and we have attempted to reduce the risks associated with it. One of the main advantages of applying GT to gray literature rather than interviews conducted by the authors is the prevention of authors influencing the results of their findings. Furthermore, the authors of this article were not connected in any way with the practitioner sources used in the study, thus lower-

ing the potential for bias even further. Finally, the modeling conducted by each author was independently checked by the other author, thus reducing the risk of individual bias during the modeling process.

Not all possible sources were considered while conducting this study, so it cannot be ruled out that some relevant sources were not found for inclusion, resulting in important factors not having been discovered and thus not being included. This cannot be avoided, since it is infeasible to include all available sources. However, since we coded to theoretical saturation, we believe this limitation does not invalidate our results. Also, please note that GT does not claim to achieve completeness [8, 9] but rather describes phenomena that exist. In the same sense, our study does not claim to document "best practices" – these are extensively covered in Serban et al. [14]. The modeled practices, decisions, etc. are merely phenomena that have been observed to exist – no further claims are made in this regard and the options, decisions, etc. may not apply universally in every practical case.

CONCLUSIONS

Our study modeled practitioner practices, ADDs, and decision drivers in the field of SE/SA for ML. The resulting ADD model can help researchers better understand practitioners' needs and the challenges they face, and guide their decisions based on existing practices. The study also opens new avenues for further research in the field, and the design guidance provided by our ADD model can also help reduce design effort and risk. In future work, we plan on using our findings to provide automated design advice to ML engineers.

Acknowledgments

This work was supported by: FFG (Austrian Research Promotion Agency) project AMMONIS, no. 879705.

References

1. Foutse Khomh, Bram Adams, Jinghui Cheng, Marios Fokaefs, and Giuliano Antoniol. Software engineering for machine-learning applications: The road ahead. *IEEE Software*, 35(5):81–84, 2018.
2. Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering*, 44(11):1024–1038, 2018.
3. Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The Emerging Role of Data Scientists on Software Development Teams. In *Proceedings of the 38th International Conference on Software Engineering*, pages 96–107, New York, NY, USA, 2016. ACM.
4. D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2503–2511, Cambridge, MA, USA, 2015. MIT Press.
5. Lucy Ellen Lwakatare, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. A taxonomy of software engineering challenges for machine learning systems: An empirical investigation. In Philippe Kruchten, Steven Fraser, and François Coallier, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 227–243, Cham, 2019. Springer International Publishing.
6. Elizamary Nascimento, Anh Nguyen-Duc, Ingrid Sundbø, and Tayana Conte. Software engineering for artificial intelligence and machine learning software: A systematic literature review, 2020.
7. Vahid Garousi, Michael Felderer, Mika V. Mäntylä, and Austen Rainer. Benefitting from the grey literature in software engineering research, 2019.
8. Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, New York, NY, 1967.
9. Anselm L. Strauss and Juliet M. Corbin. *Basics of qualitative research: techniques and procedures for developing grounded theory*. Sage Publications, Thousand Oaks, Calif,

1998.
10. Apitchaka Singjai, Georg Simhandl, and Uwe Zdun. On the practitioners' understanding of coupling smells – a grey literature based grounded-theory study. *Accepted for publication in Information and Software Technology*, 134:106539, 2021.
11. Stephen John Warnett and Uwe Zdun. Architectural Design Decisions for the Machine Learning Workflow: Dataset and Code. Zenodo, https://doi.org/10.5281/zenodo.5730291, Nov 2021.
12. H. Washizaki, H. Uchida, F. Khomh, and Y. Guéhéneuc. Studying software engineering patterns for designing machine learning systems. In *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 49–495, 2019.
13. Jan Bosch, Helena Olsson, and Ivica Crnkovic. *Engineering AI Systems: A Research Agenda*, pages 1–19. IGI Global, jan 2021.
14. Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. Adoption and effects of software engineering best practices in machine learning. *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2020.
15. Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. Software engineering for ai-based systems: A survey, 2021.
16. Isaque Alves, Leonardo Alexandre Ferreira Leite, Paulo Meirelles, and Carla Silva Rocha Aguiar. Product engineering for machine learning: A grey literature review, 2020.
17. Wo Chang and Nancy Grady. Nist big data interoperability framework: Volume 1, definitions, 2019-10-21 2019.

Stephen John Warnett is a researcher at the Faculty of Computer Science, University of Vienna, Austria. His research interest is the intersection of software engineering/architecture and artificial intelligence. Stephen received a master's degree in software engineering from the University of Applied Sciences Technikum Wien, Austria. Contact him at stephen.warnett@univie.ac.at.

Uwe Zdun is a full professor for software architecture at the Faculty of Computer Science, University of Vienna, Austria. His research focuses on software design and architecture, empirical software engineering, distributed systems engineering, software patterns, domain-specific languages, and model-driven development. Uwe has published more than 210 articles in peer-reviewed journals, conferences, book chapters, and workshops, and is co-author of several books. Contact him at uwe.zdun@univie.ac.at.