Trustworthy and Robust AI Deployment by Design: A framework to inject best practice support into AI deployment pipelines

András Schmelczer
Leiden University
Leiden, The Netherlands
andras@schmelczer.dev

Joost Visser
Leiden University
Leiden, The Netherlands
j.m.w.visser@liacs.leidenuniv.nl

***Abstract*—Trustworthy and robust deployment of AI applications requires adherence to a range of AI engineering best practices. But, while professionals already have access to frameworks for deploying AI, case studies and developer surveys have found that many deployments do not follow best practices.**

We hypothesize that the adoption of AI deployment best practices can be improved by finding less complex framework designs that combine ease of use with built-in support for best practices. To investigate this hypothesis, we applied a design science approach to develop a new framework, called *GreatAI*, and evaluate its ease of use and best practice support.

The initial design focusses on the domain of natural language processing (NLP), but with generalisation in mind. To assess applicability and generalisability, we conducted interviews with ten practitioners. We also assessed best practice coverage.

We found that our framework helps implement 33 best practices through an accessible interface. These target the transition from prototype to production phase in the AI development lifecycle. Feedback from professional data scientists and software engineers showed that ease of use and functionality are equally important in deciding to adopt deployment technologies, and the proposed framework was rated positively in both dimensions.

***Index Terms*—AI engineering, robustness, trustworthiness, deployment, best practices**

I. Introduction

Even though industry professionals already have access to numerous frameworks for deploying AI correctly and responsibly, case studies and developer surveys have found that a considerable number of deployments do not follow best practices [1]–[5]. Utilising state-of-the-art AI models has become reasonably simple; deploying them correctly is as intricate and nuanced as ever.

We hypothesize that the primary reason for the low adoption rate of best practices is the short supply of professionals equally proficient in the domains of data science and software engineering. Nevertheless, practitioners could rely on frameworks to achieve some level of automation and maturity in their deployment processes. However, the barrier of entry for using such existing libraries is too high, especially when compared with the simplicity of AI-libraries. Moreover, the support for best practices in these frameworks is either lacking or requires considerable effort and advanced skills.

Therefore, we used the design science approach [6] to develop a new framework that combines ease of use with built-in support for AI engineering best practices, and evaluated the frawework on its accessibility and best practice support. Basically, to use the framework, users place a couple of annotations into their code, may use some convenient utility functions, and subsequently obtain a rich set of capabilities, including versioning, logging, tracing, parallelisation, and more. The framework's initial design is focused on the domain of natural language processing (NLP), but with generalisation in mind.

Through literature review, framework design and evaluation, we attempt to answer the following research questions:

- RQ1 What are the barriers for adopting AI engineering best practices with existing deployment frameworks?
- RQ2 What AI deployment framework design could help to lift those barriers?
- RQ3 To what extent can an AI engineering framework automatically implement best practices?
- RQ4 How do practitioners perceive the utility, ease of use and generalisability of the new framework design?

The paper addresses these questions as follows. Section II investigates the adoption problem (RQ1). Section III outlines our design science approach to framework design. Section IV presents and justifies the actual design (RQ2). Section V covers the evaluation of the design on best practice coverage (RQ3) as well as utility, ease of use, and generalisability (RQ4). Section VI presents overall conclusions.

II. Background

Before designing our framework, we review the accessibility of AI development, the current state of AI deployment practices, and existing solutions that support AI deployment.

A. Accessible AI

In recent years, there has been a proliferation of highly accessible AI-libraries, many providing reusable models. For example, FLAIR [7] and Hugging Face's transformers [8] let developers access state-of-the-art models and methods in a couple of lines of code. Using transfer-learning, Hugging Face enables its users to leverage vast amounts of knowledge

learned by pretrained models (such as BERT [9] and its many variations) and fine-tune them for their specific use case. The API exposing this is also very accessible. Other prominent examples are SpaCy [10], Gensim [11], scikit-learn [12], and XGBoost [13]. The situation is similar in all subdomains of artificial intelligence: some domain expertise is — admittedly — beneficial but not a hard-requirement. This, combined with the exponentially increasing computing power affordably available to consumers and businesses alike [14], results in AI that is accessible by many.

B. State of the AI deployment practices

In contrast, the software landscape around packaging, deploying, and maintaining AI/ML, and in general data-heavy, applications paints a different picture.

When looking at AI/ML code in practice through the lens of technical debt, Sculley et al. [5] emphasise the repercussions of writing *glue code* between the algorithms and different systems or libraries and define it as an anti-pattern. The consequence of this is the advice against using generic libraries because their rigid APIs may inhibit improvements, cause lock-in, and result in large amounts of glue code.

Haakman et al. [2] interviewed 17 people at ING, a large fintech company undergoing a digital transformation to embrace AI. They found that the existing tools for ML do not meet the particularities of the field. For instance, a Feature Engineer working in the Data & Analytics department explained that regularly spreadsheets are preferred over existing solutions like MLFlow for keeping track of experiment results. The reason behind this is simplicity. Additionally, multiple other interviewees described the need to self-develop (or highly-customise) dashboards for monitoring deployed models, resulting in many non-reusable solutions across the company for the same problem. The authors conclude that there is a research gap between the ever-improving state-of-the-art (SOTA) techniques and the challenges of developing real-world ML systems.

In a case study at Microsoft, Amershi et al. [3] interviewed 14 people and surveyed another 551 AI and ML professionals from the company. One of the main concerns surfaced was relating to automation which is a vital cross-cutting concern, especially for testing. At the same time, a human-in-the-loop is still favoured. The survey data pointed out the difficulty posed by integrating AI, especially in the case of less experienced respondents. This was elaborated on by describing the preferences of software engineers as striving for elegant, abstract, modular, and simple systems; in contrast, data tends to be of large volume, context-specific and heterogeneous. Reconciling these inherent differences requires significant effort. Nevertheless, Microsoft manages to overcome this with a highly sophisticated internal infrastructure.

Using AI is not unique to large corporations; in a study conducted with three startups [4], the aim was to fill in the gap of understanding how professionals develop ML systems in small companies. Overall, the results showed they have similar priorities to that of large companies, including an emphasis on the online monitoring of deployed models. However, less structure is present in the development lifecycle. Similarly, Thiée [15] described the slow but ever-growing rate of ML adoption by small and medium-sized enterprises (SMEs).

Serban et al. [1], [16] described the results of their global surveys aiming to ascertain the SOTA in how teams develop, deploy, and maintain ML systems. In [1], they compiled a set of 29 actionable best practices. These were analysed and validated with a survey of 313 participants to discover the adoption rate and relative importance of each. For example, they determined the most important best practice to be *logging production prediction traces*; however, the adoption was measured to be below 40%. In more than three-quarters of the cases, newcomers to AI reported that they *partially* or *not at all* follow best practices. This tendency decreases with more years of experience, reaching a maximum adoption rate of just above 60%. In [16], Serban et al. identified another 14 best practices that specifically concern trustworthy AI. They strove to complement high-level checklists with actionable best practices. Analysing 42 survey responses revealed a familiar pattern: most best practices had less than 50% adoption.

John et al. [17] compared and contrasted recent scientific and grey literature on AI deployments from which they extracted concrete challenges and practices. They also observed that most companies are placing many more models into production than in previous years. Additionally, they pointed out that numerous deployment techniques are absent from contemporary literature, which is speculated to be caused by the immaturity of deployment processes employed in academia. Because for instance, most models in scientific literature experience only initial deployment and are not constantly replaced or refreshed as their performance degrades over time.

Finally, in a follow-up study to [17], Bosch et al. [18] organised and structured the problem space of AI engineering research based on their 16 primary case studies. The authors noted the increasing and broad adoption of ML in the industry while also emphasising that the *transition from prototype to production-quality deployment* proves to be challenging for many companies. Solid software engineering expertise is required to create additional facilities for the application, such as data pipelines, monitoring, and logging. They defined *deployment & compliance* to be one of the four main categories of problems and described it as the source of ample struggle.

C. Existing solutions

It is noticeable that given enough resources and at the scale of 4195 AI professionals, Microsoft managed to create a comprehensive in-house solution. A similar impression is given by Uber [19]; they built a highly sophisticated infrastructure using techniques from distributed and high-performance computing. Though the authors note that this solution still has shortcomings in the form of rigidity (number of supported libraries and model types), it also allows for the easy extension of the system. It is not surprising that both high-tech Fortune 500 companies overcame the problems presented by deploying AI. We can learn from their approaches; nonetheless, using

them may be infeasible for individuals and SMEs. Thus, the issues remain for the majority of practitioners.

Luckily, the open-source scene of AI/ML/DS tools, libraries, frameworks, and platforms is thriving. Additionally, there is a considerable number of closed-source — usually platforms-as-a-service (PaaS) — solutions next to them. Let us look at some prominent examples. Table I shows a high-level comparison of frameworks along the dimensions in which practitioners reportedly face difficulties in the *Deployment* stage of the CRISP-DM model [20].

IBM's AutoAI [21] promises to provide automation for the entire machine learning lifecycle, including deployment. It is a closed-sourced, paid service which — from their documentation — seems to focus primarily on non-technical users by providing them with a graphical user interface (GUI) for authoring models. The restrictions caused by the encapsulation of the entire process can be severe: the challenges of integration were emphasised above [5]. Additionally, an engineer working on Microsoft's comparable solution, the Azure ML Studio, highlighted that once users gain enough understanding of ML, such visual tools can get in their way, and they may need to seek out other solutions [3]. Unfortunately, the main value proposition of Azure ML Studio is also to provide a GUI for laypeople, and it has also been set to be retired by 2024. Its successor is Azure Machine Learning which shares many similarities with AWS's SageMaker suite [22].

SageMaker offers the most comprehensive suite of tools and services; most importantly, it has a set of features called *AWS SageMaker MLOps*. This provides easy and/or default implementations for multiple industry best practices described in [1], [16], [23]. Among others, it promotes using CI/CD, model monitoring, tracing, model versioning, storing both data and models on shared infrastructure, numerous collaboration tools, etc. Nonetheless, SageMaker does not enjoy broad adoption, as indicated by the survey data. The cause of this may be the lack of a self-hosting option and its relatively high prices: many companies prefer on-premise hosting for privacy, and financial reasons [18]. Additionally, vendor lock-in and possibly — in the case where it is not already used for the project — the initial effort required for setting up AWS integration could be likely deterrents.

When it comes to open-source libraries, we can find the MLOps libraries of both TensorFlow and PyTorch: TensorFlow Extended (TFX) [24] and TorchX. TFX comes with a more mature set of features with the caveat that initial time investment is needed for their setup. The features of TorchX only concern the distributed deployment to a wide range of providers, including Kubernetes (K8s), AWS Batch, or Ray [25]. There is no augmentation for most deployment best practices. Given the tight coupling between these libraries and their corresponding ML frameworks, they cannot generalise to models or algorithms of other frameworks and technologies.

Open-source platforms also exist, such as MLflow and Seldon Core. They both rely on Kubernetes to provide their features. MLflow emphasises the training phase (in deployment, it lacks a feedback loop which is essential for reaching many of the best practices), while Seldon Core focuses on the deployment stage. The latter comes integrated with a powerful explanation engine, Alibi Explain [26]. It also boasts the most comprehensive suite of features, including outlier detection, online model selection (with multi-armed bandit theory), and distributed tracing. Thus it seems an ideal candidate for the title of *framework for robust end-to-end AI deployments*. Its only downside is the amount of complexity propagated to its clients: it is built on top of Kubernetes and relies on Helm, Ambassador/Istio, Prometheus, and Jaeger for its features. Hence, the first step in using it is setting up a K8s cluster with all the required components; then, when it comes to model deployment, a Kubernetes configuration file must be created to use Seldon's Custom Resource Definition. These are minor obstacles if the project is already built on top of K8s; however, even then, software engineers with solid cloud and DevOps backgrounds are actively required to use Seldon Core.

Additionally, increasing attention is given to ML deployments in embedded systems both from a theoretical [23] and practical [27] point of view. Prado et al. [27] survey the available deployment frameworks and end-to-end solutions, including those for embedded devices. They note the inefficiencies of these that come from the lack of features and too much rigidity. They introduce their framework for embedded AI deployments, which can be used out-of-the-box but also lets users easily replace and extend its pipeline to fit their changing needs and advancements in the field. At the same time, Meenu et al. [23] present and compare different architectural choices for large-scale deployments in edge computing. They also note that: *"...there is a need to consider and adapt well-established software engineering practices which have been ignored or had a very narrow focus in ML literature"*.

In summary, while surveys and case studies have shown the industry's continuous struggle to evolve prototypes into robust and responsible production-ready deployments, platforms aiming to help overcome this challenge lack widespread adoption. Lack of adoption is likely due to the complexity and rigidity of current frameworks, making them are inadequate for many contexts, especially in cases where teams lack extensive expertise in cloud, operations, and more generally, software engineering (RQ1).

III. Methods

The chosen methodology for this study is Design Science which emphasises the need to design and investigate artifacts in their contexts [6]. It consists of a design and an empirical cycle. The aim of the former is to improve a problem context with a new or redesigned artifact, while in the latter, the problem is investigated, and its potential treatment is validated.

A. Design cycle

We carried out the design cycle by first formulating framework requirements, then designing and implementing an initial version, and then iteratively refining the design by applying the framework in two subsequent case studies and adapting the framework based on case study findings. The resulting design,

TABLE I
HIGH-LEVEL COMPARISON OF POPULAR AI DEPLOYMENT PLATFORMS AND LIBRARIES.

	AutoAI	Azure ML	SageMaker	TFX	TorchX	MLflow	Seldon Core
Open-source[1]				✓	✓	✓	✓
Self-hosted[1]				✓	✓	✓	✓
Vendor-agnostic[2]				✓	✓	✓	✓
AI-agnostic[2]		✓	✓			✓	✓
E2E feedback[3]		✓	✓				✓
Distributed monitoring[3]		✓	✓	✓	✓	✓*	✓
Online model selection[3]	✓*	✓	✓				✓
Versioning[3]	✓	✓	✓	✓	✓	✓	✓
Quick setup[4]	✓	✓					
No DevOps dependencies[4]					✓		

1 For privacy and accountability reasons. [18]
2 Minimising required glue code. [5]
3 Implementing best practices. [1], [16], [17]
4 Easy integration into existing processes. [2], [15]
* Only partial support.

described below in Section IV provides a tentative answer to RQ2, which is then evaluated in Section V.

In this paper, we will not report on the details of the case studies. In short, they focus on individual components of a growing commercial platform which aims to find tech-transfer opportunities in academic publications. The primary input of the system as a whole is a set PDF files, while the output is a list of metrics describing various aspects of each paper, such as interesting sentences, scientific domains, and contributions. The result also includes a predicted score used for ranking. This ranking is subsequently processed by the business developers of Technology Transfer Offices (TTOs) of multiple Dutch and German universities, who later give feedback on the results.

Overall, this practical problem context carries the properties of typical industry use cases: it utilises a wide range of natural language processing (NLP) methods, contains complex interactions between the services, benefits from the integration of end-to-end feedback, and has to provide the clients with a platform that they can rely on within their organisation's core processes. Since the final ranking affects real people, explainability and robustness are also central questions.

B. Empirical cycle

To answer RQ3, we compare the features of our framework with best practices found in literature and encountered in the case studies. For each supported best practice, we will establish a degree of support, ranging from *Fully automated*, to *Supported*, and *Partially supported*.

To answer RQ4, we conduct interviews with software engineers and data scientists with varying levels of professional background. The interview candidates were recruited through the personal network of the first author, in the second degree: direct acquaintances were asked to seek out people from their professional networks with any connection to AI/ML. After the first few interviews, participants were also asked to suggest other candidates, preferably from different subfields. After two iterations of reaching out to potential interviewees personally, ten engineers and researchers eventually responded positively and participated in the study. Albeit the sample size is small, it still represents a wide range of organisation types: experts were included from startups, consultancies, government organisations, and research companies.

First, before their interview, participants are requested to complete a questionnaire about their last completed AI project; the questions refer to the best practices implemented by our framework. They are also advised to take a quick look at the tutorial page of the documentation.

The interviews are divided into two parts. In the first part, after a brief introduction, interviewees are asked to solve a real-world deployment task by finishing a partially completed example project using the framework. This is a more straightforward instance of the AI development lifecycle presented in the tutorials. They are also encouraged to think aloud so their feedback can be noted. Successfully completing the task creates a system implementing a known number of best practices. This way, the added value —in terms of a larger number of implemented best practices— can be quantitatively analysed by comparing the qualities of the finished implementation with the previously given answers to the questionnaire. The target duration for the interviews is approximately one and a half hours.

We follow the guidelines proposed by Halcomb et al. [28] for collecting information from interviews and reporting it. This reflexive, iterative process starts by recording participants (with their permission) and concurrent note-taking. Reflective journaling is immediately done post-interview, which is subsequently extended and revised by listening to the recordings. Afterwards, we interpret the gathered information by applying the methodology of thematic analysis [29].

The second half of the one-on-one sessions consists of a short survey allowing us to evaluate our framwork against the Technology Acceptance Model (TAM) [30]. TAM has been widely applied in literature [31], and due to its general psychological origins, it proves to be effective in other areas of technology, not just software [32]. We employ the parsimonious version of TAM, which has been measured to have similar predictive power to that of the original TAM while having fewer variables [33]. Parsimonious TAM observes three interconnected human aspects that influence the actual behaviour (adoption): *perceived usefulness*, *perceived ease of use*, and *intention to use*. Participants are asked ten

questions corresponding to these aspects of their experience using our framework. The internal consistency of the answers is calculated using Cronbach's Alpha [34], after which we reflect on the responses.

IV. Designing the framework

Within the AI Engineering lifecycle, the design of our framework will be focussed on the deployment activities, which we have seen in Section II to be critical for transitioning from prototype to production, yet challenging for many practitioners. This focus is highlighted in Figure 1.

It is interesting to mention that there is a proliferation of platform/software as a service (PaaS/SaaS) products for deploying AI. These may look intriguing, but they tend to only focus on getting code easily deployed in the cloud: AI best practices are not prioritised in this setup. Nevertheless, in many cases, it may be a suitable option to use such a service, and these can also complement our framework, as illustrated in Figure 1: first, the prototype is transformed into a service in the form of a common software artifact that adhere to best practices. Then, it is either deployed using a deployment SaaS or the organisation's existing software deployment setup.

A. Requirements

The best practices we aim to support are a subset of those compiled by Serban et al. [1], [16] and John et al. [17]. These best practices are addressed in requirements described below.

a) General: Our framework should be broadly applicable. In particular, we want our framework to be compatible with a wide range of AI libraries. Large projects frequently end up depending on numerous packages, each of which may impose some restrictions on the code: since these all have to be satisfied simultaneously, this can result in severe constraints. The open-source scene of data-related libraries is vibrant. To take the example of data validation, there are at least four popular choices which offer varying but similar features: Alibi detect, Facets, Great Expectations, and Data Linter [35]. The responsibility of choosing the most fitting solution falls on the user. Our framework should not limit them with respect to such choices.

A limit to generality will be posed by the choice of programming language, which we restrict to Python. Fortunately, Python is currently almost the de facto standard programming language for data science, so implementing the framework in it should not too severely limit its general applicability.

b) Robustness: In software development, robustness can be achieved by preparing the application to handle errors gracefully, even unexpected ones [36]. Errors can and will happen in practice: storing and investigating what has led to them is required to prevent future ones. In the case of ML, errors might not be as obvious to detect as in more traditional applications (see the above-mentioned data validators). Even if only a single feature's value falls outside the expected distribution, unexpected results can happen. In cases where this might lead to real-world repercussions, extra care has to be taken to construct as many safeguards as practicable. The framework should support its users in this.

c) End-to-end: In this case, it refers to end-to-end feedback. That is, feedback should be gathered on the system's real-world performance, which should be taken into account when designing/training the next iteration of the model. Static datasets may fail to capture the changing nature of real life and can become outdated if they are not revised continuously. A well-packaged deployment should make it trivial to integrate new training data.

d) Automated: The available time of data scientists and software engineers is limited and expensive. For this reason, humans should only be involved when their involvement is necessary. Steps in the development process that can be automated without negative consequences must be automated in order to achieve efficient development processes and let the experts focus on the issues that require their attention the most.

e) Trustworthy: As detailed in the *Ethics guidelines for trustworthy AI*, human oversight, transparency, and accountability are key requirements for trustworthy AI applications. For increasing public acceptance and trust while minimising negative societal impact, trustworthiness is essential.

B. Design principles

We follow the Unix philosophy [37], [38] of software design, especially the design goal to *write programs that do one thing and do it well*. Apart from providing a clear and simple picture of the intended use cases for the library, this is also in line with the main notion of *A Philosophy of Software Design* [39]: APIs should be narrow and deep.

A narrow width refers to having a small exposed surface area, i.e. having a small number of functions and classes in the public API. In contrast, depth implies that each accomplishes an involved, complex goal. In a way, the width of an API is the price users have to pay (the effort required for learning it) to use it, while the depth is analogous to the return they get from it. Having to learn little and being provided with a lot of functionality maximises return on investment (ROI), hence, developer experience (DX).

Moreover, the theoretical frameworks presented in *The Programmer's Brain* [40] provides us with explanations and vocabulary from psychology for arguing about the cognitive aspects of API design. In the following, two of them will be used for detailing the design principles: cognitive dimensions of code bases (CDCB) which is an extension of the cognitive dimensions of notation (CDN) framework [41], and linguistic anti-patterns [42]. The former comes with a set of dimensions describing different (often competing) cognitive aspects of code that influence one's ability to perform specific tasks.

Linguistic anti-patterns provide guidelines for improving consistency and decreasing the false sense of consistency when there is none. Also, choosing the right names for identifiers can help activate information stored in the long-term memory, making it quicker to comprehend and easier to reason about the code [43]. Finding the most accurate and useful names is more challenging than it first seems. Accuracy and usefulness are already often competing goals: the more precise the name, the longer and, therefore, less convenient to use [44]. In

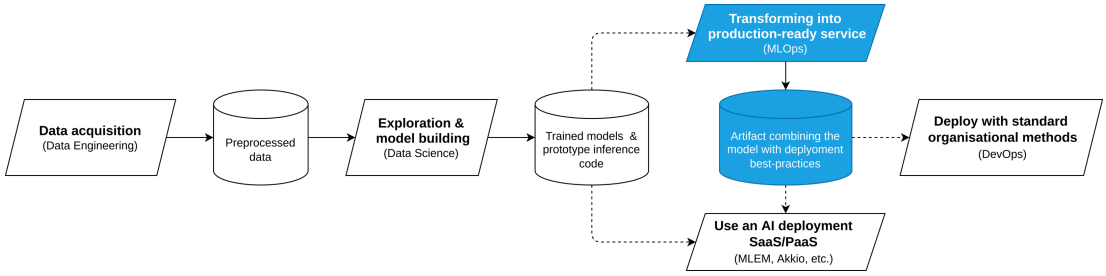

Fig. 1. Usual process steps (based on [17]) in the development lifecycle of a data-heavy software solution. The dashed arrows denote optional paths: after a prototype has been completed, there are multiple options for its deployment. The steps with blue background show the primary scope of our framework.

short, good names are essential to good APIs; consciously considering the implications of names must be an integral part of the design process.

Nonetheless, simple APIs come with a high technical cost. The library has to implement these in a way that still allows for high performance in production [45] and avoids being tied to specific libraries or technologies. Inspiration for the latter may be gained from the ML pipelines of Prado et al. [27]: they show that more freedom can be achieved with plug-and-play steps and preconfigured defaults.

C. Default configuration

Existing frameworks frequently suffer from the entanglement of numerous levels of abstractions. Instead of exposing each implementation detail and encouraging users to interact with most of them, these can be abstracted away in a more high-level layer. Even where configuration may be helpful for advanced users, default values can still be chosen automatically while providing an override option where necessary.

For example, tracing the evaluations and the model versions used in a distributed fashion is very much expected of a trustworthy system. Hence, turning this feature on by default but allowing opting-out from it can result in less scaffolding required from the library's users. It also decreases their upfront cognitive load, which by definition flattens the learning-curve [40]. Similar features can be imagined for providing a service API for the algorithms, giving feedback, marking outliers, and more.

Being *automated* is listed as a requirement, but it is imperative to only automate for simplifying and not for hiding decisions. More precisely, guessing must not be a part of automation. For instance — an otherwise handy WebGL library — TWGL.js, has a feature for automatically guessing the type of vectors based on their names. Suppose it matches the `/colou?r/i` pattern. In that case, it is treated as a vector with three components. It is easy to imagine that this can help in certain scenarios. Still, it does so at the cost of immense confusion when correctly renaming a variable breaks the application. In CDCB, this equates to scoring high on the dimension of *Hidden dependencies* and low on *Visibility*.

Learning from this, any guessing must be avoided to create a pleasant API. However, this conflicts with providing defaults for each configuration value. Even if these would be reasonable defaults derived from educated guesses, they are still merely guesses. Nevertheless, if the users were required to specify each configuration option, that would lead to vastly more boilerplate code. This verbosity is captured by the *Diffuseness* dimension of CDCB and should be minimised.

To resolve this conflict, our framework should have recommended values instead of defaults. This can mean a context object (as suggested in [39]), which contains the result of each design consideration that has to be made for a service's deployment. If not configured manually, the recommended values are applied automatically, just like defaults. However, the values chosen for each parameter must be clearly highlighted. Coming from the library's single responsibility, the number of parameters should not be immense; hence, the user can be expected to comprehend them instead of just being overwhelmed and skipping them.

This way, the library attempts to notify its user about the existence of these decisions but does not force them to decide manually. As a result, no initial configuration is needed for starting out with the library (high *Provisionality*, low *Diffuseness*), and the dependencies are not hidden since they are explicitly highlighted.

D. Documentation

The library must have quality documentation for all categories. Accordingly, for structuring it, the *Diátaxis* philosophy is preferred [46] which prescribes dividing documentation into 4 parts along 2 axes: practical-theoretical and passive-active consumption. The four quadrants derived from this are tutorials, how-to guides, references, and explanations.

Once again, we might notice two competing interests: the level of detail and the length of the documentation. For example, FastAPI, a popular Python web framework, has extensive descriptions and explanations on all topics related to Python's import system, the HTTP protocol, concurrency, deployment, and more. The actual framework's documentation is sprinkled over these overly broad topics. This is undoubtedly helpful for beginners to acquire knowledge from a single place. Yet, this high level of accessibility actually hinders the process of finding the relevant sections; in CDCB, this shows a trade-off between the support of *Searching* and *Comprehension* tasks. Diátaxis' take is that linking to external resources about the

library's domain is welcome, but the documentation must have a single responsibility: describing the library itself.

A large portion of software documentations is automatically generated from source code, and this has the advantage of always keeping it in sync with code changes. However, it might also signal that the API is too large because it is inconvenient for the developers to document it by hand. Striking the right balance between handcrafted and automatically extracted documentation may be a vital component of good documentation.

When it comes to example code, showing at least a minimal starter code and the way of customising it has to be showcased front and centre. It is a well-known observation that developers only read the documentation when they are stuck, and there might be some merit to this. Helping them not get stuck, by providing a starter code from which they can explore the API using IntelliSense-like solutions, should be preferred. Take the example of another popular Python web framework, Flask, at this time, has 324 homogeneously styled links on its landing page. Out of these, only two lead to the quick-start code. Of course, it is not hidden, but we argue that the DX could be improved by displaying where to start more prominently.

E. Developer experience

A key component of good DX is *Progressive evaluation* through which development can become a highly iterative, experimental process. This is well-understood by popular data science tools, such as Jupyter Notebooks. Our framework also has to support some level of this, for example, in the form of auto-reload on code changes. Further key ingredients of good DX are consistency and discoverability. To give one more example, the MySQL connector's Python implementation has a cursor object which exposes a `fetchone` method. Even though this naming scheme is not conventional in Python since it does not follow PEP 8 at least the API is intuitive: changing `sql_cursor.fetchone()` to `sql_cursor.fetchall()` returns all items instead of just one. Using good and consistent names is the key to good DX.

At the same time, Python codebases are rarely strictly object-oriented (OO). They are a mix of the functional, data-driven, and OO paradigms. Consequently, relying on classes for grouping related functions is not always desirable; therefore, it is even more imperative to name similar functions similarly. This helps discoverability and chunking [40], which amounts to quicker comprehension.

F. Architecture

As laid out in Section IV-B, we strive for narrow and deep interfaces; thus, it is time to address the *depth* component.

Our framework stands on the shoulders of numerous open-source packages and integrates them to provide its various features. These include: FastAPI, Plotly, mongoDB, Amazon S3, Pandas, and matplotlib. Given a Python script or a Jupyter notebook, our framework transforms the specified prediction functions into a production-ready deployment, deployable either as a Docker image, WSGI-server, or an executable relying on `uvicorn`.

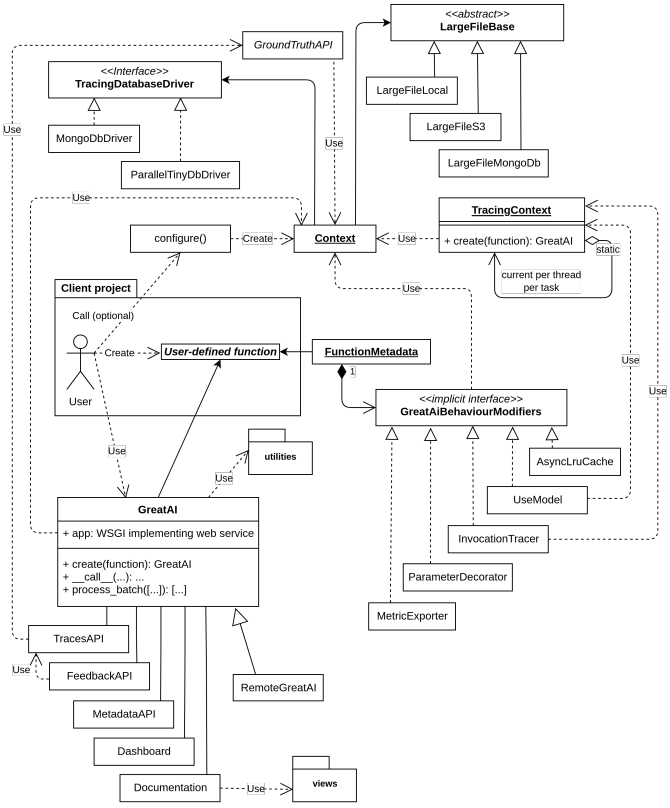

Fig. 2. The core architecture of the *GreatAI* framework illustrated with syntax loosely-based on UML2 [47]. Given its framework nature, the expected client project and the actor integrating it are highlighted; the associations between the framework and the client project are achieved through the use of decorators.

The general theme in the implementation is that each best practice has its distinct, loosely-coupled functions or classes. When collaboration opportunities arise, such as persisting model versions into prediction traces, there are three primary conduits for realising them: the `context` object responsible for global configuration per process, the `FunctionMetadataStore` specifying expected behaviour of each prediction function, and the `TracingContext` created anew for each prediction input (session).

After refining the framework with feedback gathered from case studies and users, we ended up with the core architecture presented in Figure 2. The implementation is mixed-paradigm, combining the expressiveness of functional and the design patterns of object-oriented programming (OOP) in order to maintain an overall low complexity. Reflection is also utilised, especially for run-time type-checking and generating the API definitions and dashboard components. Regardless, the architecture is still presented with a syntax similar to the class diagrams of UML2 [47] because it provides the freedom to express even the non-OOP design aspects.

For brevity, Figure 2 does not show all fields and some related entities have been combined, e.g. the *GroundTruthAPI* box represents the `add_ground_truth`, `query_ground_truth`, and `delete_ground_truth`

functions. The client project can access most presented entities, but these optional dependency arrows are not shown in the diagram. The `utilities` submodule is also left unexpanded; almost all of its functions are orthogonal with the exception of `parallel_map`, which follows a textbook producer-consumer model facilitated by queues and event signals [48].

V. Results & discussion

We evaluated our final design by comparing its features to best practices from literature, and by practitioner interviews, tasks, and surveys. We discuss our findings regarding ease of use, generalisability, and overall technology acceptance.

A. Features

Table II summarises the implemented best practices in the context of methods found by prior surveys of scientific and grey literature [1], [16], [17]. A *Level of support* is determined for each best practice on a scale of *Partially supported*, *Supported*, and *Fully automated*.

For instance, *Use static analysis to check code quality* is *Supported* because the entire public interface of our framework is correctly typed (including generics and asynchronous coroutines) and compatible with `mypy` and `Pylance`. This means that when our framework is used in any Python project, various tools can be applied to statically check the soundness of the project's integration. However, if the library's user does not use type hints in their code and it contains a more complex control flow, it can only be partially type-checked. In short, this best practice is supported, and a considerable part of it is already implemented, but users should still keep in mind that they might also need to make an effort to implement it fully. This is not the case for *Log production predictions with the model's version and input data* because, by default, it is automatically implemented when calling `@GreatAI.create`. Users can still specify the exact expected behaviour, e.g., where to store traces, additional metrics to log, or disabling the logging of sensitive input. Nevertheless, the best practice is already implemented reasonably well without user intervention.

In Table II, we added six additional best practices, which are generally well-known software engineering considerations that are also applicable to AI/ML deployments. These had not explicitly made it into the aforementioned surveys; however, according to the insights gained from our case studies, implementing them has a positive effect on deployment quality. In future research, attention could be given to their level of industry-wide adoption and quantitative utility.

In short, a large number of best practices (17) can be given a *Fully automated* implementation by our framework's design, and many others (16) can be augmented by the library. This proves the feasibility of designing simple APIs using the techniques of Chapter IV for decreasing the complexity of correctly deploying AI services while still implementing various best practices (RQ2).

B. Best practices survey

The practitioners were first asked to fill out a questionnaire about their latest AI/ML project involving deployment. This point-in-time measurement served as a baseline for the deployment quality they are used to. Analysing the results show that the amount of software engineering experience has a moderately strong correlation ($r_{Pearson} = 0.67$ with $p = 0.0033$) with the overall number and extent of implemented deployment best practices. Interestingly but unsurprisingly, there is no similar statistically significant relationship regarding the amount of data science experience.

Best practice adoption is calculated by discarding the *Not applicable* answers and projecting the 5-point Likert scale to a range from 0 to 1, which is subsequently averaged over all questions. The overall mean adoption rate/extent is just above 0.5, which equates to the *Neither agree nor disagree* label. These data are in line with the findings of Serban et al. [1].

Because the 15 survey questions were compiled from the *Fully automated* rows of Table II, they are all implemented automatically when using our framework. Consequently, the adoption rate/extent is doubled immediately just by wrapping the inference function with `@GreatAI.create`. Moreover, this provides further evidence for answering RQ3 showing the extent of automatically implemented practices over deployments without our framework.

C. Technology acceptance

The participants filled out a form after finishing their first deployment with our framework to provide data for assessing technology acceptance. The survey contained ten questions from three categories, which could be rated on a 7-point Likert scale. The summary of the answers is presented in Table III. The high Cronbach's alpha values indicate strong internal consistency [49] for each TAM dimension; thus, averaging the responses per category is semantically meaningful.

Following the methodology of [50], the connections between the Perceived Utility (PU), Perceived Ease Of Use (PEOU), and Intention To Use (ITU) dimensions of TAM were analysed. Two statistically significant ($P \leq 0.05$) correlations were uncovered: between PU and ITU ($r_{Pearson} = 0.81$ with $p = 0.0048$); and PEOU and ITU ($r_{Pearson} = 0.80$ with $p = 0.0068$). Learning from the findings of prior case studies, it is reasonable to believe that the *perceived utility* and the *perceived ease of use* play an equally important role in influencing users' *intention to use* the deployment framework.

The assessment of *ease of use* lags behind the rest, but it is still quite high. It may be possible that PEOU would go up with further use. Nevertheless, the high *perceived utility* implies that our framework shows its value early on. This, combined with the correlations uncovered within the context's technology acceptance model, validates the hypothesis that focusing on good API design is just as necessary as providing practical features.

D. Task solving & exit interviews

To give qualitative depth to the presented quantitative results, it is time to discuss the main segment of the interviews. The participants' backgrounds covered a fascinating cross-section of industrial AI/ML. The financial sector was represented by a researcher working on market prediction models

TABLE II

BEST PRACTICES AND THE LEVEL OF SUPPORT PROVIDED FOR THEM, WHERE *Fully automated* (✓✓) MEANS THAT NO ACTION IS REQUIRED FROM THE USER, *Supported* (✓) ONLY AUTOMATES THE REASONABLY AUTOMATABLE ASPECTS, AND *Partially supported* (∼) PROVIDES SOME USEFUL FEATURES, BUT THE USER IS EXPECTED TO BUILD ON TOP OF THESE.

Best practice	**Implementation**	**Support**
Use sanity checks for all external data sources[1]	`@parameter`	✓
Check that input data is complete, balanced, and well-distributed[1]	`@parameter`	∼
Write reusable scripts for data cleaning and merging (for NLP)[1]	`utilities`	✓✓
Make datasets available on shared infrastructure[1]	`large_file`	✓✓
Test all feature extraction code (for NLP)[1]	`utilities`	✓✓
Employ interpretable models when possible[1]	`views`	∼
Continuously measure model quality and performance[1, 2]	Feedback API	✓
Use versioning for data, model, configurations and training scripts[1, 2]	`@use_model`, versioning	✓✓
Run automated regression tests[1]	`*_ground_truth`	✓
Use continuous integration[1]	Docker Image, WSGI application	✓
Use static analysis to check code quality[1]	Fully typed API with generics	✓
Assure application security[1]	Code is automatically audited	∼
Automate model deployment, enable shadow deployment[1, 2]	Docker Image & scripts	✓
Enable automatic rollbacks for production models[1, 2]	Docker Image & scripts	∼
Continuously monitor the behaviour of deployed models[1, 2]	Dashboard, metrics endpoints	✓✓
Log production predictions with the model's version and input data[1]	`@GreatAI.create`	✓✓
Execute validation techniques: error rates and cross-validation[2]	`*_ground_truth`	✓
Store models in a single format for ease of use[2]	`save_model`	✓✓
Rewrite from data analysis to industrial development language[2]	Jupyter Notebook deployment	✓
Equip with web interface, package image, provide REST API[2]	`@GreatAI.create`	✓✓
Provide simple API for serving batch and real-time requests[2]	`@GreatAI.create`	✓✓
For reproducibility, use standard runtime and configuration files[2]	`utilities.ConfigFile`, Dockerfile	✓
Integration with existing data infrastructure[2]	GridFS, S3 support	✓✓
Select ML solution fully integrated with databases[2]	MongoDB, PostgreSQL support	✓✓
Querying, visualising and understanding metrics and event logging[2]	Dashboard, Traces API	✓✓
Measure accuracy of deployed model to ensure data drifts are noticed[2]	Feedback API	✓
Apply automation to trigger model retraining[2]	Feedback API	∼
Allow experimentation with the inference code[3]	Development mode & auto-reload	✓✓
Keep the model and its documentation together[3]	Dashboard and Swagger	✓✓
Parallelise feature extraction[3]	`parallel_map`	✓✓
Cache predictions[3]	`@GreatAI.create`	✓✓
Allow robustly composing inference functions[3]	All decorators support async	✓✓
Implement standard schemas for common prediction tasks[3]	`views`	✓

[1] SE4ML best practices from Table 2 of [1], and Table 1 of [16].
[2] Reported state-of-the-art and state-of-practice practices from Tables 2, 3, and 4 of [17].
[3] Additional software engineering best practices applicable to AI/ML deployments encountered while designing and using our framework.

TABLE III

TECHNOLOGY ACCEPTANCE MODEL SURVEY RESULTS PER VARIABLE. THE INPUT VALUES RANGE FROM 1 TO 7. SAMPLE SIZE = 10.

	Perceived ease of use	**Perceived utility**	**Intention to use**
Median	5.8	6.4	6.3
Mean	5.5	6.1	6.0
Standard deviation	1.0	0.9	1.3
Cronbach's alpha	0.77	0.88	0.95

.

for the Hungarian State Treasury and two people building an upcoming digital bank's core services. Image processing contexts were illustrated by professionals predicting Sun activity at the European Space Agency and different ones creating pose-recognition at a startup for people with disabilities using 3D cameras. Other activities of interviewees included investigating companies' AI use as part of due diligence processes and intrusion detection from network packet traces.

Stemming from this diversity, these semi-structured interviews could be expected to provide valuable insights into the generalisability of our framework design. The methodology of Section III-B was followed by applying reflective journaling and thematic analysis. After labelling each aspect of the feedback, and two iterations of merging redundant or related topics, we ended up with three overarching themes: *Functionality*, *API*, and *Responsibility to adopt*. As we will soon see, these correspond to the *perceived utility*, *perceived ease of use*, and *intention to use* components of TAM.

a) Functionality: The framework's feature-set was complimented during most interviews, with one participant noting that, although the overall number of features is relatively small, many are utilised in most cases. Similarly, the `utilities`

submodule was appreciated for helping greatly in the interview task, but non-NLP researchers noted its likely inadequacy for their area. Still, they would like to see similar modules for their fields because it would save them from a lot of copy-pasting.

The effortless parallel feature extraction and large file handling support were highlighted multiple times for the reason that the particular interviewees had not encountered other libraries providing these features. Other concrete features, such as the searchable *exceptions* column in the Dashboard's table and the *feedback* mechanism, were also popular. One professional highlighted the latter for coercing users to consider a human-in-the-loop approach which was said to be often expected in modern systems.

When reflecting on the framework from a bird's eye view, the generality and extensibility of the API were emphasised. As explained by a senior engineer, this is mainly because once you commit to using it, it is important not to find yourself at a dead end for a specific use case forcing you to look for a different library. However, two participants also noted that for complete generality, `MATLAB` support would be necessary. Regarding non-functional features, private hosting (especially in banking and government), open-source auditability, and good scalability (by means of an external database) were the top subjects of praise.

b) API: Regarding the surface through which clients interact with the library, the feedback is also positive but more nuanced. Many participants liked that the functions' behaviour is easy to guess from their names. The decorator syntax caused minor confusion but consulting the documentation solved the issues in all three cases. The CLI app was appreciated for having a close to trivial signature; the participant noted that she strives to use as few CLI commands as feasible. Surprisingly, even the practitioners with more data science background appreciated the Docker support. Nonetheless, one expert had a feature request for a configuration GUI because his colleagues are used to handling MATLAB App Designer applications.

The recurring theme of the discussions focused on the question of "*How simple is too simple?*". The argument is that an API cannot be simpler than the domain in which it exists. More precisely, it can only be simpler at the cost of losing transparency. Let us take the example of saving models using `save_model()`. If a project is set up correctly, it either has an initial `configure()` call to the storage provider backend, or it has an appropriately named credentials file in the project's root, for instance, `s3.ini` or `mongo.ini`. Once set up, it is trivial to use as long as we do not divert from the happy path. However, if an issue arises, such as an upgrade or migration of MongoDB, debugging the application is non-trivial for its lack of transparency.

In other words, the average (cognitive) complexity is low while the worst-case is as high — if not higher — than without using `save_model()`. This proved to be somewhat controversial. However, ultimately, optimising the happy path of the AI/ML development lifecycle was deemed worthwhile by the participants in most cases. With the argument that the majority of the time spent during a project is spent on this path anyway. However, this raises the question of who exactly are the target users and who will fix arising issues?

c) Responsibility to adopt: The question of who is responsible for adoption of engineering practices came up in many discussions. Various companies were mentioned that for multiple years used to or still have an R&D department consisting solely of data scientists. In one extreme case, the staff was described as more than 30 data scientists and 0 other technical employees. In such a setup, it is unreasonable to expect even professionals to have the capabilities and focus to set up the required foundation for handling all best practices. All but one interviewee verified this assumption. They also referred to their previous projects, which usually required many researchers and experts from various fields, and too often, software engineers had not been prioritised to be included. Adopting engineering best practices without (sufficient) software engineers is difficult, even when using a framework like ours.

E. Discussion

The overall takeaway from this is that most features were well-received, and the high mean value of *perceived utility* is credible. Also, *perceived ease of use* is relatively high, especially given the short time for participants to become acquainted with the framework. Thus, we can give a positive answer to the first two parts of RQ4 (utility and ease of use).

Regarding generalisability, the criticism of being NLP-centric is justified: the initial scope of the proof-of-principle framework was limited to this domain. Nonetheless, learning the experts' opinion that they wish to have a similarly specific solution to their problem contexts is reassuring because it proves that the API is not only generalisable but is expected to be generalised. At the same time, it is crucial to admit that no one-size-fits-all solution can exist for such a diverse domain. Therefore, allowing customisability and easy extension of the system must remain central design questions. Combining the high value of *intention to use* from Table III, the generally positive feedback regarding the library's added value, and the numerous feature requests for fitting it to specific needs, we conclude that with appropriate adaptations, generalisation is possible.

F. Threats to validity

The main threat to validity of our work lies in the small sample size of practitioners that have participated in our evaluation, and in the limited duration. This threat is alleviated to some extent by the in-depth nature of the interviews. Still, user studies over a longer period of time, with more participants and a wider variety of tasks, would surely be valuable future work.

VI. Conclusion

Transitioning from prototypes to production-ready AI/ML deployments is a source of adversity for small and large enterprises alike. Even though several frameworks and platforms exist for facilitating this step, surveys on the execution of best

practices continue to expose the industry's shortcomings. This signals that existing libraries are underutilised, which may lead to poor deployments that underperform or develop issues that go unnoticed and might inflict societal harm.

We hypothesised that presenting a library which implements best practices and is also optimised for ease of adoption could help increase the overall quality of industrial AI/ML deployments. To test this, we designed and implemented a framework based on the principles of cognitive science and the prior art of software design. Subsequently, we tested and refined the design in an iterative process.

During the refinement of the framework, six previously unaddressed AI/ML deployment best practices were identified. Including these, the framework fully implements 17 best practices while it provides support for another 16. We validated the value provided by implementing or helping to implement these practices through interviews with ten industry professionals from various subfields.

The interview participants completed two questionnaires, the results of one of which indicated that using our framework in an example task increased the number of implemented best practices, on average, by 49% compared with their latest project. We also calculated the technology acceptance; a significantly strong correlation was measured between the *perceived ease of use*, the *perceived utility* and the *intention to use* dimensions. Overall, proving that ease of use is just as important as core functionality when adopting AI deployment frameworks.

The open-ended exit interviews revealed that value can be derived from the library even in its current form and that the API's design has the opportunity to generalise to other fields of industrial AI/ML applications. However, they also highlighted that adoption issues do not necessarily come from a lack of willingness but a lack of awareness. Even if the returns achievable from good deployments are well worth the investment. Nevertheless, this value proposition needs to be conveyed and proved to data science professionals and technical decision-makers.

While our framework is useful in its own right, its design could potentially also be used to inspire improvements of the existing frameworks, such as those discussed in Section II.

Source code and documentation of our framework are available under an open source license at https://github.com/schmelczer/great-ai. Full documentation, including user guides, a tutorial, and several example applications are available at https://great-ai.scoutinscience.com/.

References

[1] A. Serban, K. van der Blom, H. Hoos, and J. Visser, "Adoption and effects of software engineering best practices in machine learning," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–12.

[2] M. Haakman, L. Cruz, H. Huijgens, and A. van Deursen, "Ai lifecycle models need to be revised," *Empirical Software Engineering*, vol. 26, no. 5, pp. 1–29, 2021.

[3] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.

[4] E. de Souza Nascimento, I. Ahmed, E. Oliveira, M. P. Palheta, I. Steinmacher, and T. Conte, "Understanding development process of machine learning systems: Challenges and solutions," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–6.

[5] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[6] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.

[7] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf, "Flair: An easy-to-use framework for state-of-the-art nlp," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, 2019, pp. 54–59.

[8] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[10] B. Srinivasa-Desikan, *Natural Language Processing and Computational Linguistics: A practical guide to text analysis with Python, Gensim, spaCy, and Keras*. Packt Publishing Ltd, 2018.

[11] R. Rehrek, P. Sojka *et al.*, "Gensim—statistical semantics in python," *Retrieved from genism. org*, 2011.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[13] T. Chen and C. Guestrin, "XGBoost," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, aug 2016.

[14] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing cpu and gpu design trends with product data," *arXiv preprint arXiv:1911.11313*, 2019.

[15] L.-W. Thiée, "A systematic literature review of machine learning canvases," *INFORMATIK 2021*, 2021.

[16] A. Serban, K. van der Blom, H. Hoos, and J. Visser, "Practices for engineering trustworthy machine learning applications," in *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE, 2021, pp. 97–100.

[17] M. M. John, H. Holmström Olsson, and J. Bosch, "Architecting ai deployment: A systematic review of state-of-the-art and state-of-practice literature," in *International Conference on Software Business*. Springer, 2020, pp. 14–29.

[18] J. Bosch, H. H. Olsson, and I. Crnkovic, "Engineering ai systems: A research agenda," in *Artificial Intelligence Paradigms for Smart Cyber-Physical Systems*. IGI global, 2021, pp. 1–19.

[19] L. E. Li, E. Chen, J. Hermann, P. Zhang, and L. Wang, "Scaling machine learning as a service," in *International Conference on Predictive Applications and APIs*. PMLR, 2017, pp. 14–29.

[20] R. Wirth and J. Hipp, "Crisp-dm: Towards a standard process model for data mining," in *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, vol. 1. Manchester, 2000, pp. 29–40.

[21] D. Wang, P. Ram, D. K. I. Weidele, S. Liu, M. Muller, J. D. Weisz, A. Valente, A. Chaudhary, D. Torres, H. Samulowitz *et al.*, "Autoai: Automating the end-to-end ai lifecycle with humans-in-the-loop," in *Proceedings of the 25th International Conference on Intelligent User Interfaces Companion*, 2020, pp. 77–78.

[22] A. V. Joshi, "Amazon's machine learning toolkit: Sagemaker," in *Machine Learning and Artificial Intelligence*. Springer, 2020, pp. 233–243.

[23] M. M. John, H. H. Olsson, and J. Bosch, "Ai deployment architecture: Multi-case study for key factor identification," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 395–404.

[24] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc *et al.*, "Tfx: A tensorflow-based production-scale machine learning platform," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1387–1395.

[25] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[26] J. Klaise, A. Van Looveren, G. Vacanti, and A. Coca, "Alibi explain: algorithms for explaining machine learning models," *Journal of Machine Learning Research*, vol. 22, no. 181, pp. 1–7, 2021.

[27] M. D. Prado, J. Su, R. Saeed, L. Keller, N. Vallez, A. Anderson, D. Gregg, L. Benini, T. Llewellynn, N. Ouerhani *et al.*, "Bonseyes ai pipeline—bringing ai to you: End-to-end integration of data, algorithms, and deployment tools," *ACM Transactions on Internet of Things*, vol. 1, no. 4, pp. 1–25, 2020.

[28] E. J. Halcomb and P. M. Davidson, "Is verbatim transcription of interview data always necessary?" *Applied nursing research*, vol. 19, no. 1, pp. 38–42, 2006.

[29] M. I. Alhojailan, "Thematic analysis: A critical review of its process and evaluation," *West east journal of social sciences*, vol. 1, no. 1, pp. 39–47, 2012.

[30] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS quarterly*, pp. 319–340, 1989.

[31] N. Marangunić and A. Granić, "Technology acceptance model: a literature review from 1986 to 2013," *Universal access in the information society*, vol. 14, no. 1, pp. 81–95, 2015.

[32] C. K. Riemenschneider, B. C. Hardgrave, and F. D. Davis, "Explaining software developer acceptance of methodologies: a comparison of five theoretical models," *IEEE transactions on Software Engineering*, vol. 28, no. 12, pp. 1135–1145, 2002.

[33] C.-S. Wu, F.-F. Cheng, D. C. Yen, and Y.-W. Huang, "User acceptance of wireless technology in organizations: A comparison of alternative models," *Computer Standards & Interfaces*, vol. 33, no. 1, pp. 50–58, 2011.

[34] J. M. Bland and D. G. Altman, "Statistics notes: Cronbach's alpha," *Bmj*, vol. 314, no. 7080, p. 572, 1997.

[35] N. Hynes, D. Sculley, and M. Terry, "The data linter: Lightweight, automated sanity checking for ml data sets," in *NIPS MLSys Workshop*, vol. 1, 2017.

[36] M. Bishop, "Robust programming," *handout for*, 1998.

[37] D. M. Ritchie and K. Thompson, "The unix time-sharing system," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1905–1929, 1978.

[38] P. H. Salus, *A quarter century of UNIX*. ACM Press/Addison-Wesley Publishing Co., 1994.

[39] J. K. Ousterhout, *A philosophy of software design*. Yaknyam Press Palo Alto, 2018, vol. 98.

[40] F. Hermans, *The Programmer's Brain: What every programmer needs to know about cognition*. Simon and Schuster, 2021.

[41] A. F. Blackwell, C. Britton, A. Cox, T. R. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre *et al.*, "Cognitive dimensions of notations: Design tools for cognitive technology," in *International conference on cognitive technology*. Springer, 2001, pp. 325–341.

[42] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[43] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[44] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 31–35.

[45] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[46] D. Procida, "Diátaxis documentation framework." [Online]. Available: https://diataxis.fr/

[47] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[48] L. Wang and C. Wang, "Producer-consumer model based thread pool design," in *Journal of Physics: Conference Series*, vol. 1616. IOP Publishing, 2020, p. 012073.

[49] J. C. Nunnally, *Psychometric theory 3E*. Tata McGraw-hill education, 1994.

[50] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2209–2235, 2019.