

Git Workflow for Active Learning: A Development Methodology Proposal for Data-Centric AI Projects

Fabian Stieler^a and Bernhard Bauer^b

Institute of Computer Science, University of Augsburg, Germany

Keywords: Active Learning, Software Engineering for Machine Learning, Machine Learning Operations.

Abstract: As soon as Artificial Intelligence (AI) projects grow from small feasibility studies to mature projects, developers and data scientists face new challenges, such as collaboration with other developers, versioning data, or traceability of model metrics and other resulting artifacts. This paper suggests a data-centric AI project with an Active Learning (AL) loop from a developer perspective and presents "Git Workflow for AL": A methodology proposal to guide teams on how to structure a project and solve implementation challenges. We introduce principles for data, code, as well as automation, and present a new branching workflow. The evaluation shows that the proposed method is an enabler for fulfilling established best practices.

1 INTRODUCTION

More and more AI projects are emerging in companies across all industries, as several surveys¹ have shown. The growing interest in AI systems stems from the promise that, given enough data, machine learning (ML) algorithms can learn to make decisions that are impossible, or at least hard to code manually. Even in critical domains, such as healthcare, new use cases are constantly emerging, and not just because research has shown that ML models are already able to outperform humans at particular tasks (Rajpurkar et al., 2022).


Besides AI projects, an ecosystem of methods, concepts, and tools have been developed around traditional software projects. However, these solutions cannot be directly transferred to AI projects, as AI- and traditional software projects differ fundamentally in one aspect: In traditional software projects, the source code is sufficient to create the artifacts. AI projects, whose artifacts include a trained ML model, have two types of inputs: code and data (Sculley et al., 2015). Since data is usually more volatile than code, the ML artifacts have to be recreated more frequently.


The research community has identified this gap

and is beginning to make its way into the field of MLOps (Lwakatare et al., 2020). In emerging concepts, such as CRISP-ML(Q) (Studer et al., 2021), the focus is on *what* a developer has to do next, but there is a lack of concrete instructions for teams *how* it can be implemented. However, due to the vast spectrum of domains for AI projects, this is understandable. Traditional software projects have answered the *how* in terms of development models and methodologies. For example, modern development methods such as DevOps build on GitFlow or trunk-based development Git workflows (Driessen, 2010). However, there is still a need to adapt established development methodologies for AI projects (Haakman et al., 2021).

This is reinforced by the recent trend in the AI community, which is facing a shift in mindset towards increasing awareness of data dependency in implementing powerful AI systems (Paleyes et al., 2022). In this context, AL is gaining popularity. This method addresses the problem of data labeling, which is often referred to as the most costly and time-consuming part of building an AI system. Here, the annotation process should be made effective by having the model iteratively select the data in a smart way that will contribute to the highest possible information gain during training.

However, such a technique leads to a further increase in the dynamics of artifacts, which is why it is necessary to develop methodologies for implementing AI projects with an AL loop. This paper aims to present a Git workflow for Active Learning (GW4AL)

^a  <https://orcid.org/0009-0004-3827-9809>

^b  <https://orcid.org/0000-0002-7931-1105>

¹(Deloitte, 2020): State of AI in the Enterprise 3rd Edition, (Capgemini, 2020): The AI-powered enterprise, (McKinsey Analytics, 2021): The state of AI

in a way that makes it easier for a development team to solve emerging implementation challenges. To this end, section 2 recaps the basics of the AL lifecycle. Section 3 outlines the relevant software engineering (SE) concepts and presents the proposed workflow. Finally, the methodology is discussed in Section 4.

2 ACTIVE LEARNING LIFECYCLE

In general, AL is defined as a method in which the ML model can select the data for training (Settles, 2009). These data points are then labeled by an oracle and added to the annotated data pool. Given the model's request for annotations, the goal is to make the data labeling process more effective and iteratively train an increasingly powerful algorithm through ongoing feedback. Looking at this interaction from a software developer's perspective, both existing concepts, such as DevOps, and currently emerging concepts from MLOps and DataOps are involved in the implementation.

We use the foundations of these three concepts to present our approach to an AL lifecycle, which emerged during the development of a three-year data-centric AI project and is shown in Figure 1. It is intended to synchronize the established practices and show individual phases, iterations, and their interrelationships in the AL loop. The core cycle gives a system view of AL: All steps to train the ML model are found in the ML iteration. Traditional phases of software development have been implemented in the development iteration. Operational tasks are adapted to the circumstances of an AI project. The selection and annotation of new data take place in the data iteration shown in blue.

2.1 Data Iteration

Currently, various specifications of a data pipeline for AI projects can be found in the literature. In (Fischer et al., 2020), data preparation in the AI Modeling Cycle is defined as curation, labeling, and augmentation. The terminology of data collection as a stage of the data iteration is used in (Idowu et al., 2021; Tamburri, 2020; Ashmore et al., 2019; Amershi et al., 2019), while this is usually followed by a phase called data transformation (Morisio et al., 2020), sometimes more specifically named to the respective tasks processing, formatting or cleaning (Idowu et al., 2021). (Amershi et al., 2019) identify in their ML workflow the three data-oriented phases of collection, cleaning, and labeling, which were also used by (Studer et al.,

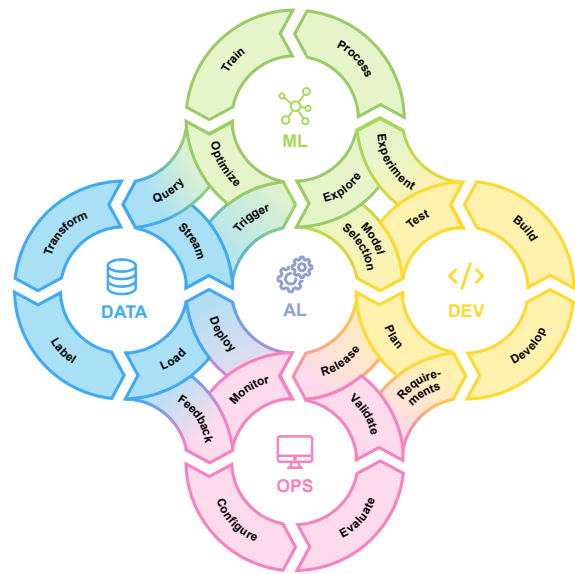


Figure 1: Active Learning Lifecycle.

2021) for their ML process methodology for the modification of CRISP-DM (Wirth and Hipp, 2000). We conclude that the core is about the three phases *Extract*, *Transform* and *Load* of the ETL process and bundle the data engineering activities on them.

According to (Settles, 2009), we will merge data extraction with differentiated AL query scenarios and distinguish between batch-based and stream-based scenarios: The former can be classified as pool-based sampling, which describes a scenario in which data points are selected from an unlabeled dataset to be annotated by the oracle (Lewis and Gale, 1994). These **Query**-scenarios represent the link between ML and data iteration in the AL lifecycle. The other query scenario differs from the batch-based method and involves stream-based sampling (Cohn et al., 1994). Thereby it is evaluated for each data point individually whether the label should be queried by the oracle. Based on the deployed application, we find the **Stream** of new data in the AL lifecycle as the second possible stage to enter data iteration.

As soon as new data is provided, it can be made versioned available for the subsequent phases. For this purpose, we follow the usual order and combine procedures from the area of Data **Transformation**. Rule-based tasks are applied to the extracted raw data. This includes data cleaning, which occurs in most data pipeline proposals, and primarily improves the quality of the data (Li et al., 2021; Ilyas and Chu, 2019).

In an AL setup, one or more query strategies (QS) exist, regardless of whether they are single batches or a data stream. There are essentially three oppos-

ing forces in deciding which samples are best for the model: informativeness and representativeness (Du et al., 2017), and the diversity of a new data point. Some QS are based on certainty (Angluin, 1988) and focus on the model's predictive ability for unknown data points. Other approaches are more decision-theoretic in nature and combine multiple models (Seung et al., 1992). Once the data has been selected by the QS and passed to data transformation, the new instances must be labeled by an oracle. This stage, represented as **Label**, can be done by a human or the model itself. The latter is often referred to as automated data labeling or semi-supervised learning (Zhu, 2008). Based on the initial human annotations, if the model reaches a certain threshold in its prediction, labels are continuously assigned automatically. However, the human-in-the-loop is especially necessary for domains e.g., with medical tasks, on the one hand, to build up a sufficiently large ground truth dataset and, on the other hand, to catch edge cases in the quality assurance process as well as to assign the correct label for rarely occurring cases (Karimi et al., 2020). Therefore, combining human- and automated annotation is a promising method (Desmond et al., 2021).

To complete the data iteration, the goal of **Load** is to make the results of the data iteration available in other stages of the AL lifecycle in a standardized way. Additional aspects need to be undertaken regarding the split into training, test, and validation datasets, as these could be subject to dynamics as well. With this result, we follow the AL lifecycle clockwise to the ML iteration.

2.2 Machine Learning Iteration

As new data or labels are available, the ML model would be continuously (re-)trained in the AL loop. Usually, the required ML pipeline consists of different substeps, which can differ depending on the problem domain and application. All tasks related to the ML model are part of the green-colored ML iteration in figure 1. During the implementation of an AL project, they can be executed periodically or event-based in an automated manner, represented as **Trigger**, as well as experimentally, which is the connection from the direction of the develop iteration. **Experimental** tasks typically fall within the scope of a data scientist, who may want to test the performance of a new model architecture or the configuration of new parameters (Kreuzberger et al., 2022).

Regardless of the application-related stages of an ML pipeline, the data required for the model are first analyzed. This phase, identified as **Exploration** in the AL lifecycle, can be executed manually manually

for the experimental case or manifested in the form of an analysis during the automated execution of the ML iteration. Although the tasks are data-driven, the focus here shifts in the model-oriented direction, for example, the recognition of attributes, statistical evaluations, or outlier detection.

Subsequently, further steps take place, which are summarized in figure 1 under **Process**. The data preprocessing for the ML model is also very domain- and use-case-specific (Studer et al., 2021). Techniques such as normalization and standardization are used, complemented by weighting and resampling, which are more generally summarized as feature engineering.

Training means optimizing the ML model to its objective function, reflecting the defined problem solution. In this process, the previously preprocessed data is fed into the model to identify patterns. The result is an algorithm that is used in an AI application as a prediction service and is successively retrained in an AL project. The path in the AL lifecycle then separates, where the ML iteration can be preceded by an optimization or - if a further improvement is to query new data and labels - to return to the data iteration. As a continuation or additional ML iteration, a model-oriented **Optimization** usually includes activities that focus on repeating the training process itself and improving the model performance e.g., by adjusting hyperparameters. (Ashmore et al., 2019)

Once a suitable version of a trained model is found, it is chosen via **Model Selection** Strategy and made available to the next iteration or to dive into the development process.

2.3 Develop Iteration

Different objectives could be pursued with the execution of the ML pipeline, resulting in requirements for the implementation of the ML pipeline to converge both, for fast feedback loops in the sense of agile software development and to converge to a good model for the problem (Amershi et al., 2019). In the area of SE for ML systems, there is growing interest in the research community. Regarding **Development**, (Arpteg et al., 2018) describe SE challenges of Deep Learning applications and identify the strong data dependency compared to traditional software development. (Nascimento et al., 2020) provide a comprehensive literature review in software development for AI.

Another characteristic of developing systems with ML, especially with an AL loop in addition to data-driven AI development, is the continuous feedback from stakeholders in the form of new data and **Requirements**, which the team of software develop-

ers, data engineers, and data scientists respond to and **Plan** changes in their system.

Figure 1 shows the yellow-colored Develop iteration next the **Build** phase, in which the new increment is merged with the project. Developing a ML application with continuous integration is investigated by (Karlaš et al., 2020) in their practice-oriented research, where they called this methodology a "de-facto standard for building industrial-strength software".

Like in traditional software development, a successful build is followed by a phase in which a series of manual and automated **Tests** are performed. In AL projects, these go beyond the application code and also include data dependency and the selected model (Rukat et al., 2020). Within the context of ML Testing, (Zhang et al., 2019) published an extensive survey. (Breck et al., 2016) provide a set of actionable tests for AI projects with their rubric-based ML Test Score.

We enter the Operations by completing or skipping the development iteration once we pass the tests with the selected retrained model and proceed to the release stage.

2.4 Operations

All operational tasks, colored pink in figure 1, will typically be carried out from the release to the deployment of a ML model and will be performed continuously in a production environment. Here, **Feedback** may come not only in the form of direct feedback from AL stakeholders but also through hidden feedback in form of measurable behavioral differences of model consumers. A possible response would be to reconfigure the current release. (Sculley et al., 2015) describe in their research the importance of **Configuration** in ML systems and establish principles, such as the necessary ability to implement a reconfiguration as a small change.

Monitor is about managing the productive model and data. **Evaluation** and **Validation** tasks result from the (re-)configuration and can also be characterized as continuous jobs of the productive AL system. Usually, this includes techniques for concept drift detection. It can identify both data and model drift and derive possible requirements for the operations team. To this end, (Renggli et al., 2021) give a data quality-driven view of MLOps in their research. Drift detection is also addressed by (Klaise et al., 2020), who also present concepts of explainability for deployed models.

An overview of operations in the end-to-end AI lifecycle is given by (Arnold et al., 2020), where **De-**

ployment is defined as "a stage of the seamless roll-out of ML models". In the study by (Paleyes et al., 2022), the deployment of ML models is divided into several stages in more detail.

The AL loop's inherent character of continuity necessitates the consideration of continuous training to automate ML iterations, continuous integration as a de facto standard of software development, and continuous deployment of the newly trained model. For its implementation, we propose the methodology presented in the following section.

3 GIT WORKFLOW FOR ACTIVE LEARNING

While the proliferation of DevOps principles and best practices has spawned methods such as Git-flow and trunk-based software development, specific approaches are just emerging in developing projects with ML. Derived from the previously defined AL lifecycle, we propose GW4AL, an agile development methodology for AL projects that provides guidelines for a team of developers and data scientists to structure their work, focusing on a data-driven development.

A central idea is the fusion of runners for Continuous Integration (CI), -Delivery (CD), and -Training (CT). Furthermore, we present a branch-based workflow concept in which we introduce data- and code-focused levels as well as new types of branches. To realize this, we first present the necessary principles related to data, code, and the runners.

3.1 Basic Concepts

In an AL project, a wide range of different ML frameworks are used. We have therefore created a development approach as generic as possible. Below, we present concepts enabling the branching workflow described in the following section 3.2.

3.1.1 Data Principle

From a process perspective, AL usually starts with collecting the raw data. Similar to the code, the data will change over the project's lifetime - either due to additional data or changing requirements, which in turn lead to new data being collected. In more concrete terms, new raw data may be imported, or new labels may be acquired through the next iteration. Thus, the artifacts of an AL project depend not only on the code, but essentially on the data, which means that the data has to be versioned. Although this can be

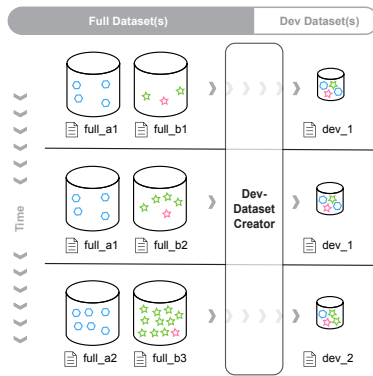


Figure 2: Development Dataset Concept.

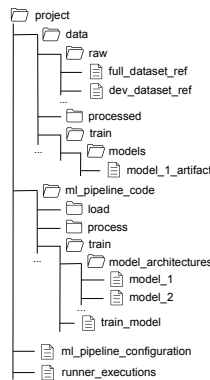


Figure 3: Proposed Project Structure.

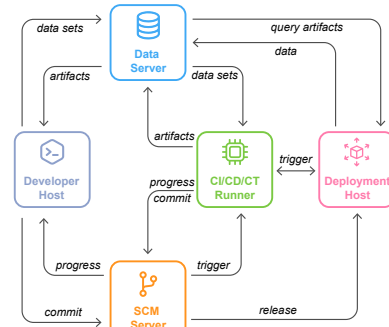


Figure 4: Minimal Infrastructure Setup.

done manually (e.g., by naming different versions of the dataset with timestamps), it is advisable to use a data versioning tool.

Developing a ML pipeline with complete, often massive, datasets renders an implementation inefficient from many points of view. Both, the data selection and the long computation time slow down the developer’s work, therefore we recommend creating a development dataset. This is a small subset of the original data with the goal that the entire ML pipeline takes only a few minutes to execute. This allows developers to use the development dataset for local development on their machine and to provide quick feedback to the runner about whether a recent commit breaks the ML pipeline.

The development dataset described above has to be reproducible and version-safe and, therefore, should not be created manually. The ideal solution is a separate code module that samples from the complete original dataset and enables automatic updates to the subset as the original dataset changes. Suppose this code module enables the possibility to include specific (groups of) samples into the development dataset. In that case, this could serve as a basis for regression testing, where samples that previously broke the ML pipeline are now checked from the beginning of the model training.

In addition to the established requirements for datasets in ML engineering (Hutchinson et al., 2021), there are additional requirements for this development dataset considering the highly dynamic nature of the data in an AL project. It should ideally approximate the distribution of the underlying full datasets and contain outliers and corrupted samples in some scenarios. Figure 2 illustrates the concept of development datasets. The full datasets are shown on the left, with changes over time at three iterations. The *Dev-Dataset-Creator* code module ensures that the development dataset is generated automatically. This can

be triggered manually, or fully automated as soon as the distribution has changed significantly between the iterations, as shown in the figure.

3.1.2 Code Principle

Usually, a traditional ML pipeline consists of different stages, regardless of the used ML framework or use case. As shown in the AL lifecycle, an ML iteration includes, among other tasks, the logic for pre-processing as well as model training. At a very early stage, for example, for prototyping, it is often common to implement all these steps in a single script or notebook (Rule et al., 2018). These artifacts are often local to the developer’s workspace or are poor at identifying version differences.

To enable a team of developers to collaborate on an AI project, it has become established practice to split up the sub-steps of a pipeline (O’Leary and Uchida, 2020). Individual code modules reflect the stages of an ML pipeline, as shown in figure 3. Therefore, it is advisable to identify the required stages when setting up an ML pipeline and then define the necessary dependencies between the individual stages. It is essential to ensure that data flows through the pipeline early in the development process, even if neither the data itself nor the resulting artifacts (e.g., the trained model) produce no meaningful results. In this way, developers are able to work in parallel on the different stages.

A sustainable implementation of the ML pipeline code is to enable configuring the stages via feature-flags and parameters. In this way, different implementations and settings of each stage can be systematically compared later on, while several developers collaborate on the same ML pipeline stage at the same time. In this context, the configuration of the ML pipeline must be stored in files that can be tracked using a version control system (e.g., Git).

To meet other requirements related to traceability, the "version number" of the input dataset must also be treated as part of the configuration. Some data versioning tools provide this by storing hashes of the records as version numbers in the configuration files. Other concepts use references, e.g., to a data catalog. In addition these tools often have the capability to cache any artifact of the ML pipeline stages (e.g., preprocessed data, trained model, evaluation report) and can restore these artifacts if an execution with identical code and data occurred in the past.

This setup creates synergies in terms of ML pipeline runtimes across the AL Project in the local development environment, when experimenting on the entire dataset, or in production. For example, if newly annotated data flows into the pipeline, the model may be re-trained and, if necessary, optimized and evaluated. On the other hand, the computationally intensive preprocessing is not necessarily repeated on the entire dataset. Skipping this step enables a fast re-deployment of the new model.

3.1.3 Automation Principle

An increasing area of MLOps research comprises the management of different environments and effective scaling of hardware resources as well as the associated concepts and tools (Ruf et al., 2021; Giray, 2021). We designed our development proposal as agnostic as possible to the underlying technologies, where figure 4 outlines a minimal infrastructure setup. It shows the necessary components and depicts a trivial implementation of a distributed system for AL projects.

First, the raw data and the created dev-dataset must be provided to the data server. Usually, this upload does not happen directly from the developer's client, but is done via an import of the respective data source in case of large datasets. The development dataset remains on the developer's client and should be small enough to allow quick iterations for ongoing work. Developers commit and pass their code to the source control management (SCM) system at regular intervals. An SCM application is then able to trigger a runner, which we call CI/CD/CT-Runner. This can be hosted on a powerful machine (e.g., with GPUs). Here, the scheduling, management, and scaling of the computational resources required for ML pipeline execution are taken into account, which is necessary as soon as the required resources exceed the capacities of the developer's computer, for example, training the model on the entire dataset.

Each job is uniquely associated with a triggering commit, so it is easy to decide whether certain long-running jobs can be aborted. Existing tools and

frameworks from traditional software projects can be reused to manage and monitor the actual servers hosting the runners. The runner checks out the version of the dataset specified in the configuration files and executes the ML pipeline. After executing the ML pipeline, the runners upload the artifacts to the data server. This could occur when the developer provides new code, when an experiment is computed on the entire dataset, or when an automatic job is triggered, such as a nightly scheduled re-training.

The SCM application provides access to the runner's logs, which is the simplest solution for monitoring the progress of the ML pipeline. This can be extended with suitable tools such as MLfLow (Zaharia et al., 2018), which may require new components (Chen et al., 2020). Again, this infrastructure concept is just the minimum and can be enhanced in several ways: Separate runners for CI, CD, and CT can be replaced by powerful clusters, a feature store and/or a model registry can be added to the data server, and much more.

Also, the implementation of the deployment, as represented in figure 4, could be more sophisticated. The deployment host covers the part of an AL system dedicated for data labeling. The interface to the oracle, in this case, could be realized in the form of a model serving component in the stream-based scenario or via the deployment of a query set for the annotation UI in the pool-based labeling scenario. Jobs for execution in the runner can be re-triggered. Data, such as acquired labels or new raw data from the client, are uploaded to the data server.

3.2 Branching Workflow

The core idea of GW4AL is to introduce different namespaces for the branches. Depending on which of these levels a new branch lives in, it focuses on the code or data dimensions. The runners behave differently depending on which branch namespace they were triggered to. To explore the branching workflow, we consider an example project in figure 5, which can begin once the setup described in Section 3.1 exists and the code for the initial ML pipeline has been committed to the main branch.

3.2.1 Main Branch

The main branch contains, as usual for software projects, the most complete version of the code. At the beginning of the AL project, this may consist only of stubs and interfaces for each ML pipeline stage. Later on, the essential requirement for the main branch is consistently to provide a clean, executable and stable version of the ML pipeline. Re-

garding to this, the runners in its namespace focus on the quality of the code: They run the ML pipeline on the development dataset as a form of integration testing. Since the configuration files contain information about the allowed values of all parameters, they are able to check the code across the allowed combinations, which could turn out to be time-consuming, even with the development dataset. In addition, traditional unit tests and other code analysis steps should be performed to keep the quality in the main branch at the desired high level.

3.2.2 Feature Branches

As soon as the development team plans a new feature or the need arises from changed requirements, a feature branch is created following the established concept of traditional software projects. Using figure 5 as an example, an initial function is to be developed. This need is documented in an issue in the SCM application and the feature branch is created through committing $(A1)$, where the implementation takes place. The developer creates a new feature flag as well as the necessary parameters and implements the function using the development dataset. After each push commit, the runner verifies that the ML pipeline is working as expected by referring to the development dataset for code execution. Static code checks as well as unit tests are executed, similar to a CI pipeline in classic software development. However, the results of the ML pipeline are not important, which is why any experiment tracking service used in the project remains disabled.

Once the developer considers the issue resolved, they start a merge request in the SCM application. These requests allow other team members to provide feedback and review the code before it is merged with the main branch and becomes version (1.0) from commit $(A2)$. At this point, it is not always necessary that the new feature actually improves the performance of the model. As illustrated by the example of the initial feature branch, sometimes, feature implementation and testing suffice, since this allows for other team members to build on the new features as quickly as possible. In some cases a combination of different features is required to actually enhance ML performance. Since the project follows the code structure described in Section 3.1, it is well feasible for multiple developers to collaborate on different features in different branches.

However, when implementing certain features in an AL project, their execution on the entire datasets is necessary, even before starting the merge request. This could be the case when implementing a new model architecture, as illustrated in our example of

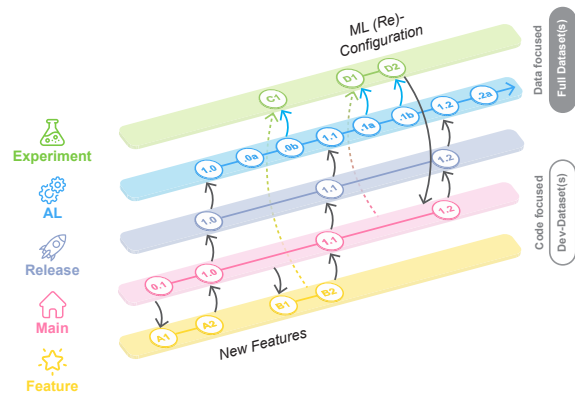


Figure 5: Branching Workflow in an AL Project.

figure 5 in the second feature branch at $(B1)$. There, the developer decides to start an experiment and branches off with the respective code version into one or more experiment branches.

3.2.3 Experiment Branches

GW4AL introduces the concept of custom branch namespaces. These include experiment branches where runners execute the ML pipeline on the complete datasets rather than the development datasets. To continue with the example: Subsequent to the implementation of the executable code of a new model architecture is its training and evaluation of a performant model. The developer creates one or more branches under the Experiment namespace and configures the ML pipeline by modifying the configuration files by enabling the feature flag and specifying the parameters. While one experiment is reserved for hyperparameter optimization, another experiment might have enabled an extension, such as additional data augmentation. Commit $(C1)$ contains a modified configuration of the actual feature commit $(B1)$ and triggers the runner with access to the complete datasets. When the ML pipelines have been completed, the runner stores the artifacts on the data server using the data versioning system, as visualized in figure 4. This ensures that for each artifact in the ML pipeline, all circumstances such as data version, code version, and execution environment used to create it, are documented.

Commit $(C1)$ could be perceived as a feasibility experiment that provides early feedback to the developer on whether further pursuing the current feature is desirable. It could be useful to trigger proof-of-concept experiments, where the initial focus shifts from fine-tuning the model to roughly evaluating ML performance. In case of failure, the overhead of a merge request and code reviews is omitted.

In addition, experiment branches support the re-

configuration of the ML pipeline. Here, the development team can directly fork a new branch (D1) from the main branch into the experiment namespace to modify the configuration file. Runners triggered by the commit of an experiment branch should enable an experiment tracking service by default. This workflow makes the development team’s reconfiguration decision transparent and reproducible at all times.

3.2.4 Release Branches

If the results of an experiment outperform the current best model, it is time to open a merge request from the main branch to the release branch. At this stage, both the code and the artifacts can be reviewed by other team members. In our example, a new runner is triggered on the release branch when the merge request is accepted at version (1.1). Using the caching feature of a data versioning tool, this runner can restore the artifacts from the experiment run (C1) from the data server. If errors or undesirable conditions occur during or after deployment, familiar countermeasures from traditional software projects, such as rolling back the release branch to a previous commit, can be applied.

To address the continuous change of data and labels in an AL project, we introduce the second special branch, that lives in the data-centric level of the project as a code-version-twin to the release branch.

3.2.5 Active Learning Branch

The AL Branch in GW4AL is defined as the mirrored provided code version of a branch, referencing a possibly more recent version of the corresponding data artifacts. The characteristics of the AL Branch correspond to those of a deployed data branch. This condition arises from the consistent use of the runners, which are also used for continuous training, transferring the data artifacts to the data server after execution of the ML pipeline, and committing the version reference of the hashes back to their AL Branch.

Making this description more concrete, we continue referencing figure 5 and the previously presented example for experiments (D1) and (D2), whereas we now focus on the new blue-colored AL Branch of our Release Branch. AL simulations or Live-Experiments go beyond classical ML experiments, e.g., they have to be performed to evaluate new label QS. Here, it is no longer sufficient to switch from development datasets to full datasets, but the runner of the experiment branches must be able to check out the updated published dataset version.

Now we leverage the synergy mentioned above of the data versioning tool’s caching feature and artifact

reuse in the other direction. When a runner in an experiment branch is triggered, it is able to pull itself the current version of the data reference file from the AL Branch and download any existing data artifacts from the data server. The code of the experiments is now executed with the current data and offers the development team the capability to make their decision based on consistently traceable and reproducible results. If the team decides to introduce further modifications to the code, the branching workflow remains the same: A merge request is derived from (D2), and the newer version of the ML-configuration is propagated through the merge in the Main Branch as version (1.2), deployed via the Release Branch in the AL Branch.

4 EVALUATION

In order to assess the practicality of GW4AL, its evaluation is performed in two phases: While in section 4.1, the comparison with existing literature provides an alignment to established concepts, we use the results of interviews in section 4.2 to incorporate a practice-oriented assessment with people from the industry. A detailed overview of all best practices compliance is provided in table 3 in the appendix.

4.1 Best Practices

To evaluate the proposed principles and the branching workflow, we draw on the best practices collected from present literature by (Serban et al., 2020). The authors focus on peer-reviewed publications that propose, collect, or validate engineering best practices for ML. Their method resulted in 29 engineering best practices categorized into data, training, coding, deployment, team, and governance. We consider which of the available aspects are (a) entirely fulfilled by GW4AL, (b) can enable a team to follow the best practice, but further action is required, or (c) not viable.

- (a) We consider 12 of 29 Best Practices to be fully satisfied. These include, for example, "Use Continuous Integration", "Enable Parallel Training Experiments", and "Use Versioning for Data, Model, Configurations and Training Scripts".
- (b) GW4AL does not per se fulfill 17 of the 29 enumerated best practices in its implementation. However, our proposed development methodology can be an enabler for their implementation. These include, for example, "Run Automated Regression Tests" (cf. 3.1.1), "Continually Measure

Model Quality and Performance” (cf. 3.1.3), and “Use Static Analysis to Check Code Quality” (cf. 3.2.2).

- (c) None of the 29 best practices are constrained in their implementation by GW4AL. In other words, teams using GW4AL are not be impeded in adhering to the specific best practices, although further methodological steps are required to achieve full compliance.

4.2 Interviews

In addition to academia, experts from industry have been interviewed in various studies to find answers to practice-oriented questions, such as in the area of SE for ML (Giray, 2021). For our evaluation, we follow that lead and exploit knowledge of such experienced experts to discuss GW4AL in semi-structured interviews. All interviewees from various industries and different company sizes, as seen in Table 1, received slides² with information about GW4AL in advance. In a 60 min face-to-face interview, the subjects were asked about their implementation of projects and their assessment of the applicability of GW4AL.

Maturity models are often used to categorize projects in the traditional SE context. In order to the interviewees professional background, we asked them to assign their projects to a maturity model with various aspects. Table 2 in the appendix provides a detailed overview of their profiles, whereby the assessment reflects a subjective perception and is not based on any hard criteria. It is noticeable that the interview partners from R&D agreed on data science-driven processes and that technology aspects correspond to a lower maturity level. The distribution of the eight interviewees is balanced in technology- and process-related implementations. On the other hand, concerning people-related factors, a large proportion attributes a higher level of project maturity to themselves. This investigation confirms, limited to the reduced group of interviewees, the thesis that methodological problems in the development of AL projects currently prevail mainly in process and technology and less in people-related working practices.

During the discussions about GW4AL and its **Technology**-related tasks, $[\alpha, \beta, \gamma, \varepsilon]$ suggested that a large focus should be placed on requirements for the selection of suitable development datasets. $[\alpha]$ continued the discussion on the data principle further in the interview and suggested a kind of A/B testing with different-sized development datasets up to the full datasets could be a compromise between fast feed-

Table 1: Profiles of interviewees.

No.	Role/Position	Industry (Employees)
α	SE Team Lead	Technology (> 100k)
β	SE Quality Manager	Technology (< 500)
γ	AI Project Manager	R&D (\approx 30k)
δ	Data Scientist	R&D (\approx 2k)
ε	Head of AI	Consulting (< 500)
ζ	ML Engineer	Consulting (< 500)
η	Data Scientist	Automotive (> 100k)
θ	Head of AI	Automotive (> 100k)

back and testing on complete datasets. Thus, the general idea to provide minimal datasets for development on the developer client and inside the code-related CI-runners nevertheless appears promising.

In terms of the **Process** and its scalability, $[\alpha, \beta, \eta]$ pointed out that it could be confusing to have a wide range of experiment branches and possibly several parallel-living AL branches on large projects. This issue should be discussed on an organizational level to develop a satisfactory solution. $[\beta, \gamma, \delta, \varepsilon, \zeta]$ expressed skepticism regarding the methodology’s realization for smaller projects with team sizes of less than five members. The main issue to consider here is the significant overhead in the early stages, as small ML and AL projects today are often still proof of concepts.

Related to **People**, $[\delta]$ said that, the necessary variety of technologies and implementation of automated processes for GW4AL, result in an increase in complexity for team members, for which some developers might lack the expertise. In this regard, $[\zeta]$ suggested that when implementing projects with GW4AL and away from the experiment branches, a kind of laboratory environment for notebooks should be provided for data scientist-driven tasks.

Through the interviews, it became apparent that GW4AL creates a solid basis for developers of data-centric AI projects with AL. The mentioned requirements for the development dataset offer potential for discussion. Furthermore, it should be stated that GW4AL introduces an unavoidable overhead into a project that may outweigh the benefits, especially in the early exploratory phase. The team needs a member capable of setting up the required infrastructure, and each team member needs to be familiar with both code and data version control systems. Developers or data scientists not used to working with a data version control system will face a steep learning curve and common pitfalls, such as overly large merge requests. Feature branches will likely require a different configuration than the current best model, such as a smaller number of training epochs to facilitate the rapid feedback loop. These two configurations will require additional management. Teams can still benefit from the principles provided, and traceability improves.

²mediastore.rz.uni-augsburg.de/get/poze0tjOHv

5 CONCLUSIONS

When implementing projects with an AL loop, addressing the requirements that arise with the additional dimension of data dependency is essential. To this end, we first presented the AL lifecycle, which orchestrates current concepts from DataOps, DevOps, and MLOps for these human-in-the-loop projects. Second, we introduced GW4AL, a proposed development methodology to help teams realize data-centric AI projects with AL. In particular, it enables the growth process of projects from feasibility studies to mature projects, as realized by us in a three-year data-centric AI project.

GW4AL provides a transparent method to collaborate for all stakeholders in developing data-centric AI projects. In doing so, projects can benefit from tools and best practices from the traditional SE domain. Different branching namespaces enforce a strict separation between the "data experiments" and the "feature implementation". The AL branch implements the loop characteristic so that the currently data version can be more upstream compared to the current released code version. Future investigations could address how to implement quality gateways along the AL lifecycle. Additional suggestions from the interviews on requirements, related to the development dataset and reducing the organizational overhead for Data Scientists provide room for future research.

ACKNOWLEDGEMENTS

This work was funded by the German Federal Ministry of Education and Research (BMBF) under reference number 031L9196B.

REFERENCES

- Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. (2019). Software Engineering for Machine Learning: A Case Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300, Montreal, QC, Canada. IEEE.
- Angluin, D. (1988). Queries and concept learning. *Machine Learning*, 2(4):319–342.
- Arnold, M., Boston, J., Desmond, M., Duesterwald, E., Elder, B., Murthi, A., Navratil, J., and Reimer, D. (2020). Towards Automating the AI Operations Lifecycle. arXiv:2003.12808 [cs].
- Arpteg, A., Brinne, B., Crnkovic-Friis, L., and Bosch, J. (2018). Software Engineering Challenges of Deep Learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 50–59. arXiv:1810.12034 [cs].
- Ashmore, R., Calinescu, R., and Paterson, C. (2019). Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges. arXiv:1905.04223 [cs, stat].
- Breck, E., Cai, S., Nielsen, E., Salib, M., and Sculley, D. (2016). What's your ML Test Score? A rubric for ML production systems. *30th Conference on Neural Information Processing Systems (NIPS 2016)*.
- Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S. A., Konwinski, A., Mewald, C., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Singh, A., Xie, F., Zaharia, M., Zang, R., Zheng, J., and Zumar, C. (2020). Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, Portland OR USA. ACM.
- Cohn, D., Atlas, L., and Ladner, R. (1994). Improving generalization with active learning. *Machine Learning*, 15(2):201–221.
- Desmond, M., Duesterwald, E., Brimijoin, K., Brachman, M., and Pan, Q. (2021). Semi-Automated Data Labeling. *Journal of Machine Learning Research*, 133:156–169.
- Driessen, V. (2010). A successful Git branching model. <https://nvie.com/posts/a-successful-git-branching-model/>.
- Du, B., Wang, Z., Zhang, L., Zhang, L., Liu, W., Shen, J., and Tao, D. (2017). Exploring Representativeness and Informativeness for Active Learning. *IEEE Transactions on Cybernetics*, 47(1):14–26.
- Fischer, L., Ehrlinger, L., Geist, V., Ramler, R., Sobiech, F., Zellinger, W., Brunner, D., Kumar, M., and Moser, B. (2020). AI System Engineering—Key Challenges and Lessons Learned. *Machine Learning and Knowledge Extraction*, 3(1):56–83.
- Giray, G. (2021). A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software*, 180.
- Google Inc. (2020). MLOps: Continuous delivery and automation pipelines in machine learning. <https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning>.
- Haakman, M., Cruz, L., Huijgens, H., and van Deursen, A. (2021). AI lifecycle models need to be revised: An exploratory study in Fintech. *Empirical Software Engineering*, 26(5).
- Hutchinson, B., Smart, A., Hanna, A., Denton, E., Greer, C., Kjartansson, O., Barnes, P., and Mitchell, M. (2021). Towards Accountability for Machine Learning Datasets: Practices from Software Engineering and Infrastructure. arXiv:2010.13561 [cs].
- Idowu, S., Strüber, D., and Berger, T. (2021). Asset Management in Machine Learning: A Survey. arXiv:2102.06919 [cs].

- Ilyas, I. F. and Chu, X. (2019). *Data Cleaning*. Association for Computing Machinery, New York, NY, USA.
- Karimi, D., Dou, H., Warfield, S. K., and Gholipour, A. (2020). Deep learning with noisy labels: Exploring techniques and remedies in medical image analysis. *Medical Image Analysis*, 65.
- Karlaš, B., Interlandi, M., Renggli, C., Wu, W., Zhang, C., Mukunthu Iyappan Babu, D., Edwards, J., Lauren, C., Xu, A., and Weimer, M. (2020). Building Continuous Integration Services for Machine Learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2407–2415, Virtual Event CA USA. ACM.
- Klaise, J., Van Looveren, A., Cox, C., Vacanti, G., and Coca, A. (2020). Monitoring and explainability of models in production. arXiv:2007.06299 [cs, stat].
- Kreuzberger, D., Kühn, N., and Hirschl, S. (2022). Machine Learning Operations (MLOps): Overview, Definition, and Architecture. <https://arxiv.org/abs/2205.02302>.
- Lewis, D. D. and Gale, W. A. (1994). A sequential algorithm for training text classifiers. In *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3–12, London. Springer-Verlag.
- Li, P., Rao, X., Blase, J., Zhang, Y., Chu, X., and Zhang, C. (2021). CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. arXiv:1904.09483 [cs].
- Lwakatere, L. E., Crnkovic, I., Rånge, E., and Bosch, J. (2020). From a data science driven process to a continuous delivery process for machine learning systems. In Morisio, M., Torchiano, M., and Jedlitschka, A., editors, *Product-Focused Software Process Improvement*, pages 185–201, Cham. Springer International Publishing.
- Microsoft Corporation (2021). MLOps with Azure Machine Learning - Accelerating the process of building, training, and deploying models at scale.
- Morisio, M., Torchiano, M., and Jedlitschka, A., editors (2020). *Product-Focused Software Process Improvement: 21st International Conference, PROFES 2020, Turin, Italy, November 25–27, 2020, Proceedings*, volume 12562 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham.
- Nascimento, E., Nguyen-Duc, A., Sundbø, I., and Conte, T. (2020). Software engineering for artificial intelligence and machine learning software: A systematic literature review. arXiv:2011.03751.
- O’Leary, K. and Uchida, M. (2020). Common Problems with Creating Machine Learning Pipelines from Existing Code. In *Workshop on MLOps Systems*.
- Paley, A., Urma, R.-G., and Lawrence, N. D. (2022). Challenges in Deploying Machine Learning: a Survey of Case Studies. arXiv:2011.09926 [cs].
- Rajpurkar, P., Chen, E., Banerjee, O., and Topol, E. J. (2022). AI in health and medicine. *Nature Medicine*, 28(1):31–38.
- Renggli, C., Rimanic, L., Gürel, N. M., Karlaš, B., Wu, W., and Zhang, C. (2021). A Data Quality-Driven View of MLOps. arXiv:2102.07750 [cs].
- Ruf, P., Madan, M., Reich, C., and Ould-Abdeslam, D. (2021). Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools. *Applied Sciences*, 11(19).
- Rukat, T., Lange, D., Schelter, S., and Biessmann, F. (2020). Towards automated ml model monitoring: Measure, improve and quantify data quality. In *MLSys 2020 Workshop on MLOps Systems*.
- Rule, A., Tabard, A., and Hollan, J. D. (2018). Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, Montreal QC Canada. ACM.
- Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.
- Serban, A., van der Blom, K., Hoos, H., and Visser, J. (2020). 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)*. arXiv:2007.14130 [cs].
- Settles, B. (2009). Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison.
- Seung, H. S., Opper, M., and Sompolinsky, H. (1992). Query by committee. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory, COLT ’92*, page 287–294, New York, NY, USA. Association for Computing Machinery.
- Studer, S., Bui, T. B., Drescher, C., Hanuschkin, A., Winkler, L., Peters, S., and Mueller, K.-R. (2021). Towards CRISP-ML(Q): A Machine Learning Process Model with Quality Assurance Methodology. arXiv:2003.05155 [cs, stat].
- Tamburri, D. A. (2020). Sustainable MLOps: Trends and Challenges. *22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 17–23.
- Wirth, R. and Hipp, J. (2000). CRISP-DM: Towards a Standard Process Model for Data Mining.
- Zaharia, M. A., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Xie, F., and Zumar, C. (2018). Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41:39–45.
- Zhang, J. M., Harman, M., Ma, L., and Liu, Y. (2019). Machine Learning Testing: Survey, Landscapes and Horizons. arXiv:1906.10742 [cs, stat].
- Zhu, X. (2008). Semi-Supervised Learning Literature Survey.

APPENDIX

Table 2: Inventory of project maturity levels and self-assessment of interviewees: Aspects, based on maturity models from (Microsoft Corporation, 2021) and (Google Inc., 2020), were divided into three categories: Technology, Process, and People. The levels increase from Level 1 (= early project stage, most manual) to Level 4 (= highest mature, fully automated). Interviewees were asked at the beginning of the interviews to use the table to rank the maturity of their projects.

	Level 1	Level 2	Level 3	Level 4
Technology	Poor SCM γ, δ	Standardized SCM ϵ	Integrated Monitoring β, η, θ	Pipeline as product α, ζ
	Untracked Artifacts β, δ	Artifact Mgmt. Tools γ	Toolset Integration ζ, η, θ	Fully Automated α
	No Automation δ	Monitoring Tools γ, ϵ, ζ	Analytic Tools β, η, θ	Integrated Resilience α
	Manual Build γ, δ, ϵ	Standardized Builds	Autom. Builds ζ	Autom. Test-Envs α, β, η
Process	Ad Hoc Development α, δ	Requirement Mgmt. ϵ	Agile Development β, ζ	Lean Development α, η, θ
	Manual Handwork γ	Manual Release δ, ϵ, ζ	Autom. Deliveries	Continuous Deliveries $\alpha, \beta, \eta, \zeta$
	Stand-alone solutions γ, δ	Modularity ϵ	Integrated Reporting β, ζ, η	Predictive Pipeline
	”Trail and error”	Manual Testing γ, δ, ϵ	Integrated Testing β, ζ, η	-Maintenance α
People	Knowledge Silos	Semi-Cooperative δ	Knowledge Mgmt. β, γ	Inter-Team Transfer $\alpha, \epsilon, \zeta, \eta, \theta$
	Poor Communication	Written Knowledge	Fast Feedback-Loops α, ζ	Consult other Teams $\beta, \gamma, \delta, \epsilon$
	No Priority-Awareness	Regular Communication	Continuous Education β	Ownership Mindset $\gamma, \delta, \epsilon, \zeta, \eta, \theta$
	Low Innovation	Innov. by Requirement β, ϵ	Innovation Strategy δ	Innovation as Vision $\alpha, \zeta, \eta, \theta$

Table 3: Best practices, collected by (Serban et al., 2020) and their fulfillment ranking for GW4AL: +++ matches a complete fulfillment of GW4AL. Scores ++ and + are intended to provide an evaluation of whether GW4AL is an enabler for achievement. For the best practice marked with *o*, GW4AL is no enabler, but in our view the compliance would not be hindered.

Nr.	Title	Fulfillment	Reference
1	Use Sanity Checks for All External Data Sources	++	Sec. 3.1.1
2	Check that Input Data is Complete, Balanced and Well Distributed	++	Sec. 3.1.1
3	Write Reusable Scripts for Data Cleaning and Merging	+++	Sec. 3.1.1
4	Ensure Data Labelling is Performed in a Strictly Controlled Process	+++	Sec. 3.2.5
5	Make Data Sets Available on Shared Infrastructure (private or public)	++	Sec. 3.1.3
6	Share a Clearly Defined Training Objective within the Team	+	
7	Capture the Training Objective in a Metric that is Easy to Measure and Understand	+	
8	Test all Feature Extraction Code	++	Sec. 3.2.2
9	Assign an Owner to Each Feature and Document its Rationale	+++	Sec. 3.2.2
10	Actively Remove or Archive Features That are Not Used	+	
11	Peer Review Training Scripts	+++	Sec. 3.2.2
12	Enable Parallel Training Experiments	+++	Sec. 3.2.3
13	Automate Hyper-Parameter Optimisation and Model Selection	+++	Sec. 3.2.3
14	Continuously Measure Model Quality and Performance	+	
15	Share Status and Outcomes of Experiments Within the Team	+++	Sec. 3.2.3
16	Use Versioning for Data, Model, Configurations and Training Scripts	+++	Sec. 3.1.1
17	Run Automated Regression Tests	++	Sec. 3.2.2
18	Use Continuous Integration	+++	Sec. 3.1.3
19	Use Static Analysis to Check Code Quality	++	Sec. 3.1.1
20	Automate Model Deployment	+++	Sec. 3.2.4
21	Continuously Monitor the Behaviour of Deployed Models	++	Sec. 3.2.5
22	Enable Shadow Deployment	++	Sec. 3.2.4
23	Perform Checks to Detect Skews between Models	++	Sec. 3.2.5
24	Enable Automatic Roll Backs for Production Models	+++	Sec. 3.2.4
25	Log Production Predictions with the Model’s Version and Input Data	+++	Sec. 3.2.5
26	Use A Collaborative Development Platform	++	Sec. 3.1.2
27	Work Against a Shared Backlog	+	
28	Communicate, Align, and Collaborate With Multidisciplinary Team Members	+	
29	Enforce Fairness and Privacy	<i>o</i>	