
Proceedings Paper:
Kourouklidis, Panagiotis, Kolovos, Dimitris, Noppen, Joost et al. (2023) A Domain-Specific Language for Monitoring ML Model Performance. In: Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2023, 01-06 Oct 2023 Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. Institute of Electrical and Electronics Engineers Inc., SWE, pp. 266-275.

https://doi.org/10.1109/MODELS-C59198.2023.00056

A Domain-Specific Language for Monitoring ML Model Performance

Panagiotis Kourouklidis
University of York
British Telecom
Ipswich, United Kingdom
panagiotis.kourouklidis@york.ac.uk

Dimitris Kolovos
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

Joost Noppen
British Telecom
Ipswich, United Kingdom
johannes.noppen@bt.com

Nicholas Matragkas
Université Paris-Saclay
Paris, France
nicholas.matragkas@york.ac.uk

***Abstract*—As machine learning (ML) starts to offer competitive advantages for an increasing number of application domains, many organisations invest in developing ML-enabled products. The development of these products poses unique challenges compared to traditional software engineering projects and requires the collaboration of people from different disciplines. This work focuses on alleviating some of these challenges related to implementing monitoring systems for deployed ML models. To this end, a domain-specific language (DSL) is developed that data scientists can use to declaratively define monitoring workflows. Complementary to the DSL, a runtime component is developed that implements the specified behaviour. This component is designed to be easily integrated with the rest of an organisation's ML platform and extended by software engineers that do not necessarily have experience with model-driven engineering. An evaluation of the proposed system that supports the validity of the approach is also presented.**

***Index Terms*—Machine Learning, Model-Driven Engineering, Dataset Shift, MLOps**

I. Introduction

In recent years, machine learning (ML) approaches have found a lot of success across various application domains. According to Delloite's latest state of AI report [1], 94% of business leaders agree that AI is critical to success over the next five years. However, while interest in the technology is very high, organisations are finding it challenging to deploy production AI systems, with the above report listing "lack of technical skills" as a major challenge and Gartner reporting that while launching AI pilot projects is relatively easy, turning them into production systems is "notoriously challenging" [2].

These challenges are not a new phenomenon. Researchers from Google, an organization that was an early adopter of ML in production, argue that ML systems are challenging to implement as they have all the challenges of traditional software systems in addition to ML-specific issues [3]. Additionally, they claim that in ML systems, only a fraction of the code is ML-specific, with the rest of the code concerning typical software system aspects such as data management or infrastructure configuration. This duality of ML systems often means their development needs multi-disciplinary teams comprising software engineers and data scientists. Nahar et al. [4] find that this collaboration between people from different disciplines brings unique challenges. One reported challenge is the potential mismatch between the responsibilities assigned to a person and their capabilities and preferences. Specifically, data scientists prefer to receive support with software engineering tasks rather than doing it all themselves. On the other hand, software engineers find it challenging to perform ML tasks due to a lack of domain knowledge. Additionally, the use of different terminology in the two disciplines can lead to ambiguity, misunderstandings and inconsistent assumptions. Finally, they report that many of these conflicts stem from the lack of clear responsibility boundaries and recommend that teams should carefully define them.

The work presented in this paper focuses on the last stage of the ML lifecycle, the continuous monitoring of deployed ML models to ensure that they perform as expected. Monitoring is important as complex interactions between algorithms and data can lead to ML models failing unexpectedly, potentially exposing their operators to adverse economic and legal consequences. Organisations consider these AI-related risks to be major barriers to AI adoption [1].

Despite its importance, Nahar et al. report that most organisations do not perform monitoring as it is considered difficult [4]. In light of this, we propose a model-driven engineering (MDE) approach with the aim of enabling data scientists to deploy ML monitoring workflows while maintaining a clear boundary between data science and software engineering tasks. The proposed solution, named Panoptes, comprises a domain-specific language (DSL), which data scientists can use to produce high-level specifications of monitoring workflows and a runtime component that implements the specified workflow. This runtime component makes minimal assumptions about its deployment environment and can interface with other components developed by an organisation's software engineers. Based on an empirical study conducted with the participation of data scientists, we find that they can, with little effort, familiarise themselves with the developed DSL and utilise it effectively to specify monitoring workflows. Additionally, participants' qualitative evaluations indicate that the proposed solution has the potential to significantly reduce the effort required for the implementation of monitoring. Finally, to ensure that the proposed solution can cover a wide range of scenarios, three case studies based on externally developed ML models of different input and output modalities and implementation technologies were investigated.

The rest of this paper is structured as follows: Section II provides an overview of the work. Section III introduces an illustrative example used throughout the paper and discusses the theoretical considerations of the ML monitoring domain. Section IV introduces Panoptes Description Language (PDL), the DSL developed for data scientists to define ML model monitoring workflows. Section V explains how PDL models are utililsed to implement the runtime behaviour of the monitoring system. Section VI presents an evaluation of our proposed solution based on an empirical study. Section VII discusses three case studies that support the generality of the proposed solution. Finally, Section VIII summarises the paper and discusses future work.

II. Proposed Approach Overview

To support data scientists throughout the typical ML workflow [5], an organisation's software engineers develop ML platforms comprising multiple components [6]–[9]. Figure 1 (top third) illustrates an example ML platform with a notebook server for exploratory data analysis and ML model training, an ML model registry, a data warehouse for storing raw data, a feature store for transforming raw data into usable features, an ML model prediction server, and a ground truth data ingestion service.

Given the reactive nature of monitoring workflows, continuous observation of specific ML platform components is required. We adopt an event-based architecture, where ML platform components emit messages when relevant events occur, such as serving predictions or ingesting ground truth data. Figure 1 (middle third) depicts the MDE layer of our proposed solution, which includes the Panoptes orchestrator as the main component. The orchestrator receives these messages and also the monitoring workflows defined by data scientists in the form of MDE models. Based on this information, the orchestrator initiates the stages of a monitoring workflow by emitting messages.

Figure 1 (bottom third) shows the services that receive messages from the orchestrator requesting the execution of a monitoring workflow stage. These services, developed by an organisation's software engineers, are responsible for interfacing with the various ML platform components to successfully execute each stage of a monitoring workflow.

This overall approach allows the proposed solution to be agnostic to the underlying ML platform technologies. Additionally, an organisation's software engineers do not need to have MDE knowledge but only adhere to the message format of the orchestrator. Further technical details are provided in Section V.

III. ML Performance Monitoring Domain

To clarify the concepts relevant to the ML model monitoring domain, we introduce a simple hypothetical scenario in which various parameters could negatively affect the performance of an ML model used by a production system. In our scenario, there is a customer support call centre that customers can contact when facing an issue with a product they purchased. Every

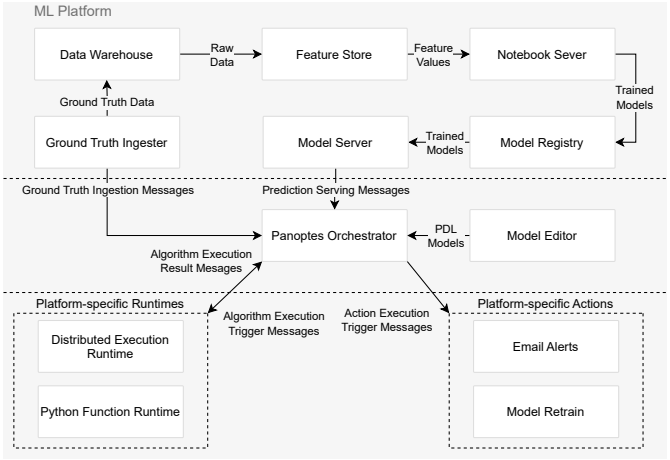

Fig. 1. Example ML Platform with an Integrated Panoptes Deployment.

incoming call goes through the following stages: Initially, the customer is put in a waiting queue until a call centre worker is available to service them. Next, they speak with a worker who tries to resolve their issue. Finally, the call ends with the customer's issue potentially being resolved.

From a data science point of view, the aim is to predict whether customers were left satisfied with the service they received based on their call's wait time, service time and issue resolution. This scenario falls under the supervised learning subset of ML [10], [11]. In such scenarios, the objective is to extract a mapping from a set of labelled samples (the training dataset) that can be used to predict the value of a target variable Y, given the value of an observed variable X. The variables X and Y follow an unknown joint probability distribution which is usually assumed to remain unchanged between the time we train the model and the time that we use it to make predictions [12].

In the call centre example, this would correspond to gathering a dataset of calls that includes wait duration, service duration, issue resolution, and customer satisfaction measurements. Applying ML algorithms to this dataset, we can obtain a mapping (a classifier model) that predicts future customers' satisfaction given their call data as long as the mapping remains unchanged.

Unfortunately, in real applications, hidden variables can affect the statistical distribution of the observable variables and how they map to the target variable, which can negatively affect the predictive accuracy of ML models. Such scenarios have been studied for several years using numerous terms, including concept drift/shift [13], [14], covariate/sampling shift [12], [14], [15], prior probability shift [12], [14] and others. In recent years, the umbrella term "dataset shift" has been introduced in [16] and further standardised in [17] and [18]. In this paper, we adopt the term "dataset shift" and its subtypes as defined in [17].

In the call centre example, the following are some of the ways that dataset shift could manifest:

- On a particular day, an unusually high number of work-

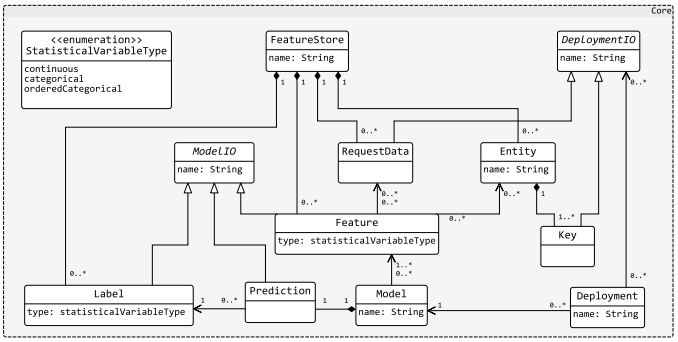

Fig. 2. Core classes of the PDL Metamodel.

```
1  Model callcenterDecisionTree{
2    uses wait_duration, service_duration, is_solved
3    outputs callcenterDecisionTreePrediction
4    predicts is_happy}
5
6  FeatureStore{
7    features wait_duration, service_duration, is_solved
8    labels is_happy}
9
10 Deployment callcenterDeployment{
11   model callcenterDecisionTree}
```

Listing 1. PDL model showcasing the metamodel's core classes.

ers might be on leave, resulting in increased waiting times. Assuming that customers' preferences towards waiting times stay constant, more customers will be left unsatisfied. According to the dataset shift terminology, this scenario is an instance of covariate shift. Theoretically, covariate shift should not affect the ML model's performance since the input-to-output mapping remains constant. However, when the ML model is relatively simple (e.g. Linear classifier), it might approximate the actual mapping well for a specific range of input values but give incorrect results when the input values shift [12]. Given that checking for covariate shift does not require the additional collection of ground truth labels, it might be helpful in scenarios where it is known that a model's performance degrades when the input is outside a particular range.

- After some time, customers might become less tolerant of long waiting times. As a result, even though waiting times remain constant, more customers will be left unsatisfied. In other words, the mapping between input and output has changed. In dataset shift terminology, this is an instance of concept shift. The collection of ground truth labels is necessary to detect this kind of dataset shift since no change in the distribution of input values is observed.
- Additionally, both of the above shifts can coincide. In other words, the average waiting time could increase in addition to the customers becoming less tolerant of lengthy waiting times. This scenario falls under the "other types of shift" category in dataset shift terminology.

IV. Panoptes Description Language

In this section, we present the domain-specific language developed for the specification of strategies for detecting and reacting to the presence of dataset shift.

A. Core

Before discussing anything related to dataset shift, some core classes must be introduced to the DSL's metamodel. These classes lay the foundation by representing the available ML models, the inputs used to train them, their outputs, and whether they are currently deployed and serving prediction requests. Figure 2 shows the classes discussed in this subsection.

The first core class is *Platform*. A *Platform* instance is the top-level element in a PDL model. It represents the infrastructure that hosts all ML-related components. A *Platform* directly contains instances of *Model*, *FeatureStore*, *AlgorithmRuntime*, *Algorithm*, *Action* and *Deployment*. This class is omitted from Figure 2 for conciseness. An ML platform can be utilised to deploy one or more ML models in production. Over time, as more data become available and more advanced ML techniques are developed, data scientists might choose to replace an older ML model with a newer one. This can be hidden from any downstream service that consumes the predictions of an ML model.

A distinction has been made between the *Deployment* and *Model* classes to accurately represent this. *Model* instances represent ML models regardless of whether they are currently being used in production. A *Model* references one or more *Features* and a *Prediction* to express an ML model's inputs and output, respectively. Additionally, a *Prediction* references a *Label* that represents the ground truth values the ML model tries to predict. Since the *Feature*, *Prediction* and *Label* classes are all related to the input and output of ML models, they subclass the abstract class *ModelIO*.

On the other hand, *Deployment* instances represent a specific task that can leverage an ML model, such as predicting customer satisfaction in the call centre example. Each *Deployment* references a *Model* instance, representing the current ML model used to accomplish the task.

Finally, the concept of the feature store is introduced to express that multiple ML models can use each feature, and those ML models might attempt to predict the values of the same variable. Feature stores are represented in the metamodel by the *FeatureStore* class whose instances contain multiple *Feature* and *Label* instances so that they can be referenced by as many *Model* instances as needed.

With the metamodel classes introduced so far, an initial model of the call centre scenario can be constructed. Listing 1 shows this model in the textual concrete syntax developed for PDL using Xtext[1]. Nevertheless, as explained in Section V, the orchestrator is not coupled with any specific concrete syntax, so support for multiple concrete syntaxes is possible.

[1]https://www.eclipse.org/Xtext/

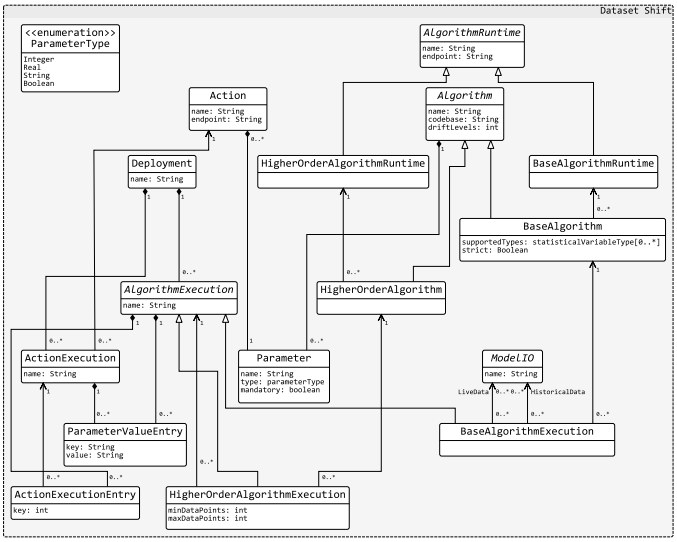

Fig. 3. Classes of the PDL Metamodel related to Dataset Shift.

B. Dataset Shift

This subsection introduces the metamodel classes used to specify the stages of a monitoring workflow. These stages comprise the dataset shift detection stage and a subsequent corrective action stage. The relevant classes for the expression of these stages can be seen in Figure 3.

To detect dataset shift, data scientists implement algorithms using general-purpose languages and create *Algorithm* instances to represent them in a PDL model. For example, Listing 2 shows a Python[2] script that uses the Kolmogorov-Smirnov [19] statistical test to check if two datasets are sampled from the same underlying statistical distribution and lines 3-7 in Listing 3 show the corresponding *Algorihtm* instance. Notice how the script does not seem complete. Specifically, the script only defines a function that expects specific arguments and returns certain values based on the result of the statistical test. There is no code for fetching the relevant data and passing it to the function or any code that shows how the returned result is used.

Filling in the gaps mentioned above is the responsibility of algorithm runtimes. These are components of the ML platform implemented by software engineers capable of fetching data from an ML platform's data store, executing algorithms and handling the results of the execution. The developer of an algorithm runtime documents the requirements that the supported algorithm implementations must fulfil and makes them available to data scientists. In line 5 of Listing 3, for example, we can see that the "kolmogorovSmirnov" *Algorithm* references the "pythonFunction" *AlgorithmRuntime*. This runtime requires the algorithm to be implemented as a Python function with the input/output signature seen in Listing 2. Further technical details about algorithm runtimes are given in Section V.

[2]https://www.python.org/

```
1 from scipy import stats
2
3 def ksTest(trainSet, liveSet, parameters):
4     pValue = stats.ks_2samp(trainSet, liveSet))[1]
5     if pValue < parameters['pValue']:
6         return 1, pValue
7     else:
8         return 0, pValue
```

Listing 2. Dataset shift detecting algorithm implementation in Python.

Given the attributes of the Kolmogorov-Smirnov algorithm, we can apply it to the call centre scenario to monitor the values of the "wait duration" variable against covariate shift. For this, the algorithm needs to receive as inputs the values of the "wait duration" variable in the training set and in recent prediction requests received by the deployed ML model. Information regarding the inputs of an algorithm to apply it in a specific scenario is contained in *AlgorithmExecution* instances, as seen for examples in lines 22-27 of Listing 3. Additionally, algorithms can be parametrised. The names of the parameters are defined in the *Algorithm* definition (e.g. line 7 of Listing 3), and their values are provided in the *AlgorithmExecution* (e.g. line 26 of Listing 3). The algorithm runtime passes the parameter values to the algorithm along with the rest of the inputs. Finally, *AlgorithmExecutions* also include a mapping that links the algorithm execution results to the relevant action execution (e.g. line 27 of Listing 3).

Related to the execution of actions in response to dataset shift are the classes *Action* and *ActionExecution*. An example of an action is sending an email notification to the data scientist that trained the affected ML model. Similarly to algorithm runtimes, action execution functionality is provided by specific software components in the underlying ML platform. These components are represented by *Action* instances in PDL models.

Based on the same principle that links *Algorithms* with *AlgorithmExecutions*, while *Action* instances represent a capability of the underlying platform, its usage is represented by an *ActionExecution* instance that parametrises the *Action* to fit in the context of a particular scenario. As an example, lines 37-39 of Listing 3 show the "emailMe" *ActionExecution* uses the "email" *Action* parametrised with an email address.

1) Base and Higher Order Algorithms: *Algorithm*, *AlgorithmExecution* and *AlgorithmRuntime* are abstract classes. So far, the examples showcase the subclasses *BaseAlgorithm*, *BaseAlgorithmExecution* and *BaseAlgorithmRuntime*. Instances of *BaseAlgorithm* represent algorithms that can detect dataset shift from *ModelIO* values (i.e. features, predictions and labels) in the training and live datasets. On the other hand, *HigherOrderAlgorithm* instances represent algorithms that take as input a set of outputs from the execution of another algorithm.

As Listing 2 shows, the execution of a base algorithm returns a pair of values. The first value is an integer representing the presence and severity of dataset shift. It corresponds to the "keys" of the result-action execution map of the algorithm

```
1  BaseAlgorithmRuntime PythonFunction
2
3  BaseAlgorithm kolmogorovSmirnov{
4    codebase "http://repo.com/kolmogorov-smirnov"
5    runtime PythonFunction
6    severity levels 2
7    parameters pValue}
8
9  HigherOrderAlgorithmRuntime higherOrderPythonFunction
10
11 HigherOrderAlgorithm exponential-moving-average{
12   codebase "http://repo.com/exponential-moving-average"
13   runtime higherOrderPythonFunction
14   parameters alpha,threshold
15   severity levels 2}
16
17 Action email{parameters address}
18
19 Deployment callcenter{
20   model callcenterDecisionTree
21
22   BaseAlgorithmExecution ksWaitTime{
23     algorithm kolmogorovSmirnov
24     live data wait_time
25     historical data wait_time
26     parameter values pValue = 0.05
27     actions 1->emailMe}
28
29   HigherOrderAlgorithmExecution emaWaitTime{
30     algorithm exponential-moving-average
31     observed execution ksWaitTime
32     min observations 3
33     max  observations 3
34     parameter values alpha = 0.5, threshold = 0.05
35     actions 1->emailMe}
36
37   ActionExecution emailMe{
38     action email
39     parameter values address=user@company.com}
40
41   Trigger t1{
42     every 100 samples 100 predictions 100 labels
43     or
44     every one day
45     execute ksWaitTime}}
```

Listing 3. PDL model showing a simple dataset shift detection scenario.

executions that utilise the algorithm. The possible values must be in the [0,N) range where N is the *severity levels* attribute of the relevant *Algorithm* instance. The second return value of the algorithm is the ”raw result” and is meant to be used as input to higher-order algorithms for further analysis.

Listing 3 shows how a *HigherOrderAlgorithm* can be used. It expresses a scenario where a data scientist wants to increase covariate shift detection robustness by considering the last N execution results of the abovementioned Kolmogorov-Smirnov algorithm. An example algorithm that can be used for this is the exponential moving average that calculates a weighted average of the results, with more recent results having a higher weight (the implementation is not shown for brevity).

From the listing above, one can notice that a *HigherOrderAlgorithmExecution* has a few differences compared to a *BaseAlgorithmExecution*. One needs to define the *AlgorithmExecution* (Base or HigherOrder) whose output will serve as the input of the newly created *HigherOrderAlgorithmExecution* (seen in lines 29-35 of Listing 3). In addition, the minimum and maximum number of observations must be defined (seen in lines 32-33 of Listing 3).

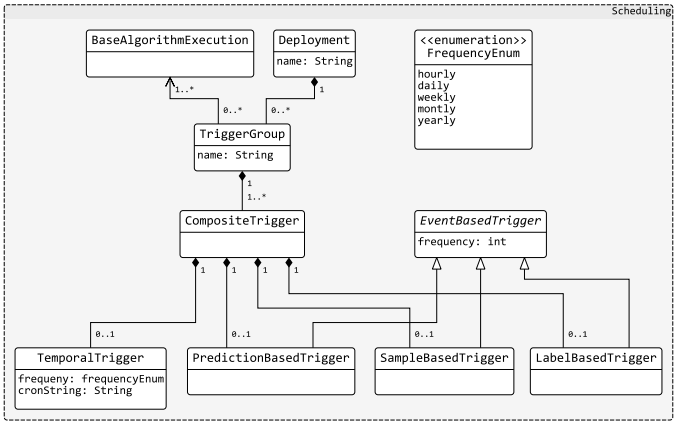

Fig. 4. Classes of the PDL metamodel related to scheduling.

C. Scheduling

An essential aspect of the modelled domain is defining the frequency of algorithm executions and the subsequent execution of corrective actions as needed. For this purpose, classes that represent various triggers and classes that represent a combination of triggers are provided. More specifically, *TemporalTrigger*, *SampleBasedTrigger*, *PredictionBasedTrigger*, and *LabelBasedTrigger* can express the frequency of one or more algorithm executions in terms of how long it has been since the latest execution, how many unlabelled samples have been received, how many predictions have been served and how many labels have been received for previously unlabelled samples respectively. For more complex scenarios, data scientists can create *CompositeTrigger* instances containing up to one instance for each kind of individual trigger and represent scenarios in which the requirements must be met for multiple individual triggers simultaneously. Lastly, *CompositeTrigger* instances can be grouped in a *TriggerGroup* instance. A *TriggerGroup* instance tracks which *BaseAlgorithmExecution* is to be executed when the requirements for at least one of the contained *TriggerGroups* are met.

Lines 41-45 of Listing 3 show a trigger group has been added to the ”callcenter” *Deployment* to showcase all possible combinations. It should be noted that trigger groups only trigger base algorithm executions as higher order algorithm executions are triggered indirectly when the algorithm execution they observe executes (provided that the observed algorithm execution has been triggered at least the minimum number of times required).

D. Validation

Although the metamodel of PDL already captures certain typing and cardinality constraints, there is still the possibility of creating a model that contains errors according to the semantics of the domain. For this reason, a series of constraints based on the semantics of the domain are defined, and every PDL model is validated against these constraints. This section explains the parts of the metamodel relevant to validation. These are not mandatory to specify a monitoring workflow,

```
FeatureStore{
  entities call{keys callID}
  request data rd
  features
    wait_duration:continuous{requires entities call},
    service_duration:continuous{requires entities call},
    is_solved:categorical{requires entities call},
    additional_feature{requires request data rd}
  labels is_happy:categorical}

Model callcenterDecisionTree{
  uses wait_duration, service_duration, is_solved
  outputs happiness_prediction
  predicts is_happy}

BaseAlgorithm kolmogorovSmirnov{
  codebase "http://repo.com/kolmogorovsmirnov"
  runtime PythonFunction
  severity levels 2
  accepts only continuous
  parameters mandatory pValue:Real}

Action email{
  parameters mandatory email:String}

Deployment callcenter{
  inputs callID
  model callcenterDecisionTree

  BaseAlgorithmExecution ksWaitTime {
    algorithm kolmogorovSmirnov
    live data wait_duration
    historical data wait_duration
    actions 1->emailMe
    parameter values pValue = 0.05}

  ActionExecution emailMe{
    action email
    parameter values email=user@company.com}}
```

Listing 4. PDL model utilising validation features.

but including them in a PDL model enables the error detection features.

Listing 4 shows an example PDL model that uses features of the DSL that enable the optional feature and type validation.

1) Feature Validation: Feature validation ensures that every deployment uses a suitable ML model. This is true when every feature used as input by the ML model is retrievable from the deployment's inputs. This validation can prove valuable when a deployment transitions to a newer ML model utilising more or different features to ensure compatibility with the existing prediction consumers.

For instance, in the call centre scenario, for a specific call, one can retrieve the associated "wait_duration", "service_duration", and "is_solved" values. Thus, instead of sending the actual values of the variables, a prediction requester can send just a "call ID" that can be used to uniquely identify the call and retrieve all the necessary inputs for a customer satisfaction prediction. If additional features are available per call, these can be added as inputs to a newer version of the satisfaction prediction ML model while keeping the data that prediction consumers have to send unchanged.

This is reflected in the PDL metamodel with a pattern already familiar to data scientists using a feature store in their workflow. Every *Feature* can reference zero or more *Entities*. An *Entity* represents a concept in the domain within which a data scientist builds a predictive model, for example, a call in the call centre scenario. *Entities* contain one or more *Keys* which can uniquely identify them. *Keys* can also be referenced by a *Deployment* as inputs. For a PDL model to be valid, every *Deployment* must reference the relevant *Keys* such that all features that the *Deployment's* referenced *Model* uses can be retrievable.

Alternatively, some features are calculated on-the-fly based on data only available when a prediction is requested. This, for example, could be a search query that a user submits to a search engine. This is represented in the PDL metamodel by the *RequestData* class. If a *Feature* references a *RequestData* instance, this must also be referenced as input by any *Deployment* that uses an ML *Model* that uses said *Feature*.

2) Type validation: An additional category of semantic validity checks relates to the classes in the metamodel characterised by a type attribute. This includes the *Parameter* and *Feature* classes.

Instances of *Parameter* are contained by *Algorithm* and *Action* instances to denote that they are parameterisable at runtime. The model validation procedure here consists of checking that the *ParameterValueEntries* contained in *AlgorithmExecution* and *ActionExecution* instances have valid parameter names as keys and valid values. For example, in Listing 4, the *kolmogorovSmirnov BaseAlgorithm* contains a *Parameter* named "pValue" of type "Real". Therefore it would be checked that every *AlgorithmExecution* that references this *BaseAlgorithm* contains a *ParameterValueEntry* with "pValue" as the key and a value that is a valid *real* number. Additionally, we check that every *Parameter* denoted as mandatory is given a value and no *ParameterValueEntries* are accidentally left with empty keys or values.

The *type* attribute of *Feature* and *Label* instances can optionally be given a value. This *type* attribute is enumerative. It can take the values *continuous*, *categorical* and *orderedCategorical* that correspond to the different types of statistical variables found in the literature [10]. Additionally, *BaseAlgorithms* can optionally be given one or more values of the same enumerative type in their *supportedTypes* attribute. Based on this information, we can validate whether the *ModelIO* instances referenced by a *BaseAlgorithmExecution* are of suitable statistical type for the *BaseAlgorithm* used. Lastly, suppose the *strict* attribute of a *BaseAlgorithm* is set to false. In that case, only a warning will be generated in case of statistical variable type mismatch instead of an error, which will not cause the rejection of the validated PDL model.

For example, in Listing 4, it is defined that the *kolmogorovSmirnov BaseAlgorithm* only supports *continuous* statistical variables as inputs. This is because while variants of the Kolmogorov-Smirnov statistical test can be used for statistical variables of ordered categorical type [20], the algorithm implementation shown in Listing 2 depends on the Kolmogorov-Smirnov implementation found in the SciPy Python package [3] which only supports continuous statistical variables. This could lead to errors that are difficult to detect since the data

[3]https://scipy.org/

scientist that implemented the algorithm could be someone other than the one using the algorithm to detect covariate shift in a specific context. To avoid this situation, this type of validation is provided.

V. Technical Implementation

As briefly mentioned in Section II, Panoptes' main component, the orchestrator, ingests users' PDL models and communicates via message passing with other components to keep track of events relevant to ML monitoring and trigger the executions of algorithms and actions when necessary. To further explain this mechanism, this section presents the inner workings of the orchestrator as well as those of algorithm runtime and action services. Additionally, technical details of a web editor service that allows data scientists to create and edit PDL models in their browser is presented. For readers that are interested in further technical details, all code repositories of the components presented in this paper are publicly available[4].

A. Panoptes Orchestrator

As seen in Figure 1, the orchestrator ingests the PDL models received from the web editor and is responsible for implementing the runtime behaviour specified in them. While the PDL metamodel is designed so that data scientists can conveniently specify how deployed ML models are to be monitored, it does not provide the orchestrator with a way to keep track of its internal state to determine when algorithm and action executions must be triggered. For this reason, after the orchestrator receives a PDL model, it constructs Finite State Machine (FSM) objects that directly map to the required runtime behaviour.

Specifically, an FSM object is constructed for every *Deployment* in a processed PDL model. As shown in Figure 5, this FSM only has one state, labelled *STANDBY*, and multiple transitions from *STANDBY* back to itself that are triggered based on the messages ingested by the FSM. Whenever a transition is triggered, an accompanying function is executed that will potentially send out a message to either an algorithm runtime or action service under certain conditions. Transitions are added according to the following rules:

- One transition is added for every *TriggerGroup* a *Deployment* contains. These are triggered whenever a base algorithm must be executed and will always send a message to the relevant algorithm runtime.
- One transition is added for every *AlgorithmExecution* a *Deployment* contains. These trigger after an algorithm execution finishes. Based on the result of the execution, if an action needs to be executed, a message is sent out to the relevant action service.
- One additional transition is added if the above *AlgorithmExecution* is a *HigherOrderAlgorihtmExecution*. These trigger whenever the observed algorithm executes. If the observed algorithm has been executed a sufficient number of times, a message is sent out to the algorithm runtime of the higher-order algorithm.

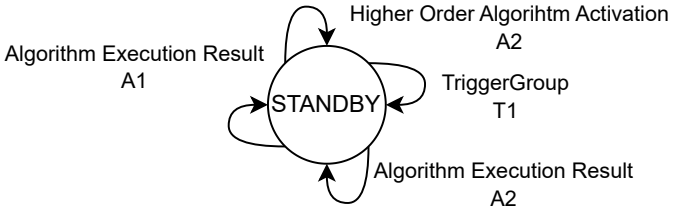

Fig. 5. Example FSM representation of a PDL model used by the orchestrator.

For the transitions to be triggered at the right time, the relevant components of the ML platform need to send a message to the orchestrator whenever a monitoring-related event occurs. Algorithm execution events are sent by the algorithm runtime services. For *TriggerGroups*, there are multiple relevant messages. These correspond to the *EventBasedTrigger* classes. No external messages are needed for *TemporalTriggers* since the orchestrator can independently keep track of the amount of time passed since the last time a *TriggerGroup* transition has triggered. The messages corresponding to *SampleBasedTriggers* and *PredictionBasedTriggers* are emitted by a model server when it responds to a prediction request. These inform the orchestrator that a new sample has been received and was used to produce a new prediction. Messages corresponding to *LabelBasedTriggers* are emitted by any service that ingests a ground-truth label for a previously served prediction. Once enough messages have been accumulated to satisfy the *TriggerGroup*conditions, the transition occurs, and the corresponding counters are reset.

B. Algorithm Runtimes

As mentioned in Section IV, algorithm runtime services are responsible for packaging the algorithms that data scientists develop and executing them with the right inputs when requested by the orchestrator.

For the packaging step, an algorithm runtime service must be able to ingest a message informing it of the creation of a new algorithm. The orchestrator sends this message whenever a PDL model introduces a new algorithm. The message includes the algorithm's git repository from which the runtime can retrieve all the artefacts needed for the packaging. In the Python function runtime, for example, when an algorithm creation message is received, the runtime clones the repository and copies the file containing the algorithm function and the pip[5] requirements file into a Python project template. From that, a docker image is built that, when run, can fetch the relevant data, execute the algorithm, and send a message to the orchestrator with the execution result.

For the execution step, an algorithm runtime service must be able to ingest a message instructing it to execute a previously packaged algorithm with specific inputs. The messages received for this step differ slightly between base and higher-order algorithm runtimes. For base algorithm runtimes, the

[4]https://github.com/pkourouklidis/panoptes

[5]https://packaging.python.org/

message includes the names of the features/predictions/labels to be used as inputs. The runtime triggers the execution of the packaged algorithm and passes the names in as environment variables so the actual values can be fetched from the data repository specific to each ML platform. On the other hand, for higher-order runtimes, the message will include the name of the observed algorithm execution instead. The runtime will similarly trigger the execution of the packaged algorithm with the name passed in as an environment variable. The past execution results of the observed algorithm execution will be retrieved by an API that the orchestrator exposes.

C. Actions

Action services are more straightforward to implement than algorithm runtime services since they do not have to execute any code dynamically. The only message that an action service must be able to ingest is instructing it to perform the action it represents. The message includes values of the parameters that the action accepts and auxiliary information, such as the name of the algorithm execution that has caused the action to execute in case it is useful.

D. Panoptes Web Editor

To provide data scientists with a simple user experience for creating PDL models and sending them to the orchestrator, an Xtext-based web editor was developed. The editor offers various convenient features such as syntax highlighting, error highlighting and auto-completion. Another nice feature of a web-based editor is that it does not require installation. When the user has finished creating or updating a PDL model, the editor backend will parse the textual syntax and generate an equivalent model in XMI. This is then sent to the orchestrator to be processed as described in the previous subsection. Following this pattern, the orchestrator only needs to support XMI, while multiple concrete syntaxes can be developed based on users' preferences.

VI. Laboratory Experiment

This section presents an empirical study conducted to evaluate the proposed solution, with the participation of 10 data scientists working within British Telecom (BT). The study sought to answer the following research questions:

RQ1 Are the concepts of the DSL easily understood by data scientists?

RQ2 Can data scientists effectively utilise the DSL to implement dataset shift detection strategies?

RQ3 How do data scientists evaluate the solution's potential for reducing the effort of implementing ML performance monitoring policies?

A. Study Design

The study was split into three parts, each seeking to answer one of the research questions. In the first part, the participants were given access to documentation material describing PDL, similar to Section IV. After reading the material, the participants were shown an example PDL model utilising every feature of the DSL and given fifteen comprehension questions to answer based on the model. All of the documentation and questions are publicly available[6].

For the second part, the study took the form of a *laboratory experiment* since it was conducted in a highly controlled setting created specifically for the study [21]. Based on the call centre scenario, a system was developed that simulates customers' calls. Additionally, a dashboard was developed for initiating simulations with different parameters that affect customer satisfaction, such as the tolerance of customers to high waiting times. Finally, a dashboard was developed that visualises the various results of each simulation. Participants were given access to the dashboards and asked to complete the following tasks.

- To let participants familiarise themselves with the mechanics of the simulator, they were told to use the example PDL model and a set of settings for the simulator such that no dataset shift is observed. After executing the simulation, the users could observe in the visualisation dashboard that the algorithm execution detected no dataset shift.
- Keeping the PDL model unchanged, the participants were given a set of simulator settings that introduced covariate shift. The algorithm execution defined in the given PDL model detected the shift, and participants received an email notification.
- Finally, participants were given a set of simulator settings that introduced concept shift and asked to modify the PDL model so that the shift would be detected and they would receive an email notification.

Their scripts were collected and analysed to quantify how well data scientists utilised PDL for the given task. One point was awarded for successfully completing each of the following tasks.

1) Defining a *BaseAlgorithmExecution* that uses the right *Algorithm* (accuracy-check).
2) Adding the right data as input to the *BaseAlgorithmExecution*.
3) Adding the correct result-to-action map to the *BaseAlgorithmExecution*.
4) Defining a *Trigger* with the newly created *BaseAlgorithmExecution* as the execution target.
5) Setting the frequency of the *Trigger* every 100 *Labels*.

For the third part, a subjective evaluation was selected due to the difficulty of measuring the effort required to build a monitoring solution and comparing it against a Panoptes-based solution with the same characteristics. Instead, five tasks were selected that were deemed representative of the monitoring domain. Participants were asked for their subjective evaluation in the following way: On a qualitative basis, how much do you think Panoptes could reduce the effort required for the following tasks? Please answer on a scale of one to five:

1) Implementing dataset shift algorithms.

[6]https://github.com/pkourouklidis/panoptes-wiki

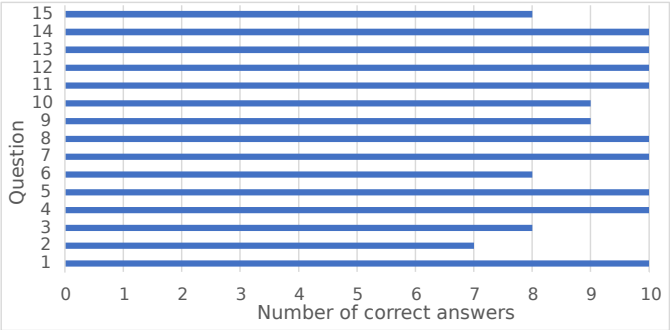

Fig. 6. Results of the first questionnaire.

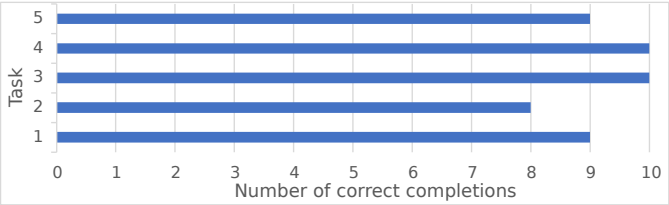

Fig. 7. Results of PDL usage test.

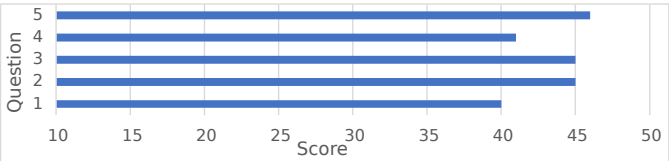

Fig. 8. Effort reduction evaluation results.

2) Fetching the relevant data to check for dataset shift.
3) Scheduling the execution of dataset-shift-detecting algorithms.
4) Executing corrective actions in case of dataset shift.
5) Modifying various monitoring parameters (e.g. algorithm used, frequency of execution) on a live system.

Notably, the questions also address parts of the domain where we did not expect Panoptes to offer significant utility but were included to get a complete picture of the participants' opinions. The implementation of dataset shift algorithms, for example, is streamlined by the runtime mechanism but still requires effort by data scientists.

B. Study results

The study results[7] show that data scientists can quickly familiarise themselves with PDL. They also expect Panoptes to provide significant benefits in terms of effort reduction for ML performance monitoring tasks.

Regarding the first two research questions, the participants spent approximately one hour reading the documentation material before answering the first questionnaire and completing the requested PDL model. Considering that all participants were busy professionals that could not afford to invest much time to complete the tasks, as participation was not part of their job responsibilities, the fact that most of the questions were answered correctly by every participant indicates that data scientists can pick up PDL very quickly. The few mistakes can perhaps even be attributed to people hastily reading the questions rather than a difficulty in understanding PDL. Figure 6 shows for every question the number of participants that correctly answered it. Similarly, Figure 7 shows that most participants were able to complete every task of the second part.

Regarding the third research question, Figure 8 shows the cumulative score for every aspect participants were asked to evaluate. Since the evaluations were given on a scale of one to five, the maximum score was fifty. As expected, certain features of Panoptes were valued more than others. Specifically, the aspects requiring some degree of manual effort, namely developing new algorithms and actions, received the lowest score. On the other hand, fully automated features, such as scheduling, received a higher score. Finally, a correlation was noticed between a data scientist's background and their evaluation of Panoptes. In line with expectations, data scientists from a mathematics background valued the usage of Panoptes more than those from a software engineering background.

VII. Case Studies

To support the claim that Panoptes is general enough to support ML models of different input and output modalities and different implementation technologies, three case studies using publicly available ML models were conducted[8]. The case studies also show the versatility that is made possible by custom dataset-shift-detection algorithms.

For the first case study, an image classifier implemented using TensorFlow was used. After deployment, to simulate dataset shift, 1000 randomly selected images from the Stanford dogs dataset were classified after being made darker. To detect this shift, a detection algorithm was developed for the "pythonFunction" algorithm runtime that calculates the luminance of recent images and checks to see if they are distributed similarly to the luminances in the training set. A second algorithm was also developed that utilises ground truth labels, when available, to calculate the accuracy of the model for recent inputs and compares it against a threshold.

For the second case study, a speech-to-text model implemented using PyTorch was used. To simulate dataset shift, 1000 randomly selected voice clips from the common voice dataset were captioned after white noise was added to them. To detect this shift, a detection algorithm that compared the signal-to-noise ratio of recently received audio clips to those in the training set was developed. A second algorithm that utilises ground truth data was also developed that calculates the word error rate, an accuracy metric commonly used for speech-to-text models, and checks to see if it crosses a set threshold.

[7]https://zenodo.org/record/8140392

[8]https://github.com/pkourouklidis/panoptes

For the third case study, a credit-scoring model implemented using scikit-learn was used. The model was trained on a tabular dataset with multiple columns containing an applicant's income bracket, savings, employment status etc. To simulate dataset shift, 1000 samples were classified after the value of the "gender" column was switched for some of them. To detect this shift, a detection algorithm that calculates the L-infinity distance was implemented. Additionally, the accuracy calculation algorithm developed for the first case study was reused since it applies to both models.

These three case studies indicate that Panoptes can accommodate a wide variety of ML models. Facilitating this is the fact that there is no inherent restriction to the type of ML model as long as the underlying ML platform supports it. The case studies also highlight the benefit of easily deployable detection algorithms that are tailored for each specific model.

VIII. Conclusion & Future Work

In this paper, we have presented Panoptes, an MDE solution for the ML monitoring domain. The proposed solution offers a DSL to data scientists that allows them to specify monitoring workflows at a high level of abstraction. A runtime component is developed that can ingest models of the DSL and implement the defined behaviour by communicating with other components of an ML platform. Additionally, an extensibility mechanism is offered so that software engineers can add new functionality to the monitoring solution. Furthermore, we presented the results of an empirical study that indicates that data scientists can quickly familiarise themselves with the solution and find it useful for implementing ML monitoring systems. Finally, we presented three case studies that indicate that the solution is able to accommodate ML models of different input and output modalities and implementation technologies.

In future work, it would be interesting to develop an additional Python-embedded concrete syntax for PDL and check whether that could benefit adoption, as Python is a language that a lot of data scientists are already familiar with. This would not require any changes to the orchestrator as it is already decoupled from the concrete syntax of PDL. Finally, we would like to evaluate Panoptes in a production environment and have been in contact with production teams to explore possible avenues of collaboration. On a longer-term horizon, the MDE approach followed could be extended to cover more aspects of the ML model commercialisation process. BT's applied research department has invested resources into areas related to AI governance, and we believe that MDE could play a significant role in this domain.

Acknowledgment

The work presented in this paper has been partially supported by the Lowcomote project, which received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 813884.

References

[1] "State of ai 2022 report," accessed: 2023-06-26. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/us/Documents/deloitte-analytics/us-ai-institute-state-of-ai-fifth-edition.pdf

[2] "Future of ai technologies report," accessed: 2023-06-26. [Online]. Available: https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-ai-technologies

[3] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2503–2511.

[4] N. Nahar, S. Zhou, G. Lewis, and C. Kästner, "Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 413–425.

[5] S. Amershi, A. Begel, C. Bird, R. DeLine, H. C. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: a case study," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, H. Sharp and M. Whalen, Eds. IEEE / ACM, 2019, pp. 291–300.

[6] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe *et al.*, "Accelerating the machine learning lifecycle with mlflow." *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.

[7] S. Schelter, F. Bießmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas, "On challenges in machine learning model management," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 5–15, 2018.

[8] K. M. Hazelwood, S. Bird, D. M. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, pp. 620–629.

[9] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data lifecycle challenges in production machine learning: A survey," *SIGMOD Rec.*, vol. 47, no. 2, pp. 17–28, 2018.

[10] J. Friedman, T. Hastie, R. Tibshirani *et al.*, *The elements of statistical learning*. Springer series in statistics New York, 2001, vol. 1, no. 10.

[11] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning*. MIT press, 2018.

[12] A. Storkey, "When training and test sets are different: characterizing learning transfer," *Dataset shift in machine learning*, vol. 30, pp. 3–28, 2009.

[13] J. C. Schlimmer and R. H. Granger, "Incremental learning from noisy data," *Mach. Learn.*, vol. 1, no. 3, pp. 317–354, 1986.

[14] B. Schölkopf, D. Janzing, J. Peters, E. Sgouritsa, K. Zhang, and J. Mooij, "On causal and anticausal learning," in *29th International Conference on Machine Learning (ICML 2012)*. International Machine Learning Society, 2012, pp. 1255–1262.

[15] M. Salganicoff, "Tolerating concept and sampling shift in lazy learning using prediction error context switching," *Artif. Intell. Rev.*, vol. 11, no. 1-5, pp. 133–155, 1997.

[16] J. Quiñonero-Candela, M. Sugiyama, N. D. Lawrence, and A. Schwaighofer, *Dataset shift in machine learning*. Mit Press, 2009.

[17] J. G. Moreno-Torres, T. Raeder, R. Alaíz-Rodríguez, N. V. Chawla, and F. Herrera, "A unifying view on dataset shift in classification," *Pattern Recognit.*, vol. 45, no. 1, pp. 521–530, 2012.

[18] M. Kull and P. Flach, "Patterns of dataset shift," in *First International Workshop on Learning over Multiple Contexts (LMCE) at ECML-PKDD*, 2014.

[19] J. L. Hodges, "The significance probability of the smirnov two-sample test," *Arkiv för Matematik*, vol. 3, no. 5, pp. 469–486, 1958.

[20] T. B. Arnold and J. W. Emerson, "Nonparametric goodness-of-fit tests for discrete null distributions." *R Journal*, vol. 3, no. 2, 2011.

[21] K. Stol and B. Fitzgerald, "The ABC of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 11:1–11:51, 2018.