EMMM: A Unified Meta-Model for Tracking Machine Learning Experiments

Samuel Idowu*, Daniel Strüber*†, and Thorsten Berger*‡
*Chalmers | University of Gothenburg, Sweden
†Radboud University Nijmegen, Netherlands
‡Ruhr University Bochum, Germany

***Abstract*—Traditional software engineering tools for managing assets—specifically, *version control systems*—are inadequate to manage the variety of asset types used in machine-learning model development experiments. Two possible paths to improve the management of machine learning assets include 1) Adopting dedicated machine-learning experiment management tools, which are gaining popularity for supporting concerns such as versioning, traceability, auditability, collaboration, and reproducibility; 2) Developing new and improved version control tools with support for domain-specific operations tailored to machine learning assets. As a contribution to improving asset management on both paths, this work presents *Experiment Management Meta-Model (EMMM)*, a meta-model that unifies the conceptual structures and relationships extracted from systematically selected machine-learning experiment management tools. We explain the meta-model's concepts and relationships and evaluate it using real experiment data. The proposed meta-model is based on the Eclipse Modeling Framework (EMF) with its meta-modeling language, *Ecore*, to encode model structures. Our meta-model can be used as a concrete blueprint for practitioners and researchers to improve existing tools and develop new tools with native support for machine-learning-specific assets and operations.**

***Index Terms*—Machine learning experiments; Management tools; MDE; Metamodeling;**

I. Introduction

Improving the effectiveness of developing intelligent software systems requires new methods and tools for managing the development of machine learning components. The management of machine learning models and other involved assets is essential for two reasons: First, training a machine learning model involves a potentially long sequence of experiments consisting of several iterations, a.k.a., *runs* [1]. Each run employs different versions of assets (e.g., datasets, hyperparameters, source code) within the solution space of the considered task. Explicit management of experimental runs can help model developers avoid redundant effort and recover earlier experimental paths on demand. Second, trained models are integrated into larger software systems, in which their performance is continuously monitored [2]. A typical activity is the retraining of models after more data has become available, possibly from new encountered contexts. To make informed judgments for retraining, it is even essential to understand the data used for training the earlier versions of the model in question, and the experimental paths that have been explored during the training.

Traditional software engineering tools for the management of assets–specifically, *version control systems* (VCSs)–are faced with severe challenges when used for such tasks [1], [3]–[6]. Machine-learning-enabled systems involve a greater variety of asset types than traditional software development, including *resource artifacts* such as datasets, features, and models; *software artifacts* such as source code files and (hyper)-parameters; and *metadata*, including experiment metadata, execution metadata, and performance metrics [7]. VCSs are not geared to support advanced, domain-specific queries on such assets, such as: *which features have been used in a run in which the final model precision was 0.6 or greater*? Consequently, new ways to manage machine-learning model development experiments involving these asset types are needed [5]–[9].

To address this situation, we identify two paths to improve the management of machine learning assets:

First, dedicated *machine-learning experiment management tools* that aim to provide support to manage assets of machine learning experiments effectively. Many of such tools have recently become available, including Neptune.ai, DVC, and MLflow. These tools aim to offer practical ways to maintain an account of the provenance of the assets and processes used during machine learning experiments, supporting concerns such as versioning, traceability, auditability, experiment reproducibility, and collaboration. Fig. 1 shows a high-level illustration of how they work. Yet, as these tools have become available quite recently, they are potentially not fully matured yet. Mora-Cantallops et al. [10] highlight several factors that may hinder the adoption of these tools, including: lack of interoperability across different tools, lack of explicit representation of domain knowledge, and friction or overhead incurred during usage. As a common drawback, users have to carry out a lot of code instrumentation to track assets [11].

Second, new and improved version control tools, building on traditional ones and extending them with domain-specific operations tailored to machine learning assets. This would address the perspective of software engineers, who routinely use standard tools such as Git, thereby addressing the issues of interoperability and usage overhead to some extent. Yet, such tools should ideally be interoperable with existing experiment management tools, which are tailored toward the needs of data scientists. Furthermore, the development of such tools should incorporate the domain knowledge about machine learning experiments and runs, which is already available in machine-learning experiment management tools.

In this work, we contribute toward both paths: the development of next-generation versioning tools and the improvement of existing machine-learning experiment management tools. We present the *Experiment Management Meta-Model (EMMM)*, a meta-model that unifies concepts and relationships extracted from systematically selected experiment management tools, focused on the concept of *experimental runs*.

Our meta-model characterizes two main concerns: (i) machine-learning asset structures as concepts and their relationship as observed in the state-of-the-art tools; (ii) conceptual version control structures that can hold both machine learning and traditional assets. It can be used as a blueprint for practitioners to improve existing tools (targeting data scientists) and for researchers to develop new tools (targeting software engineers) with capabilities to natively support the identified concepts and relationships. We hope such capabilities can foster the realization of tools with native support for machine-learning-specific asset types and natively support machine learning experiment concerns, such as versioning, traceability, auditability, collaboration, and reproducibility.

We evaluate our meta-model on a real case, validating its usefulness and suitability for capturing actual revision histories of machine learning software. In addition, we discuss the improvements enabled by our meta-model in terms of possible use cases. EMMM and the evaluation artifacts are available from an online appendix: `https://github.com/emmm-metamodel/emmm`.

II. Background

A. Machine Learning Experiments

Similar to traditional software engineering processes [12], machine-learning experiments follow well-defined processes designed in data science and data mining contexts such as CRISP-DM [13] and KDD [14]. The workflow, which consists of data-oriented, model-development, and model-operation stages [6], [8], [9], [15], has multiple feedback loops that represent the iterations over sets of steps within the workflow for a variable number of times until the process results in the desired outcome [5]. An asset of a machine learning experiment is an individually storable unit of an experiment that serves a specific purpose [7].

Many incremental iterations are often performed over workflow stages as experiment runs. In some cases, runs can be executed in parallel to improve performance. Each run represents a unique configuration in the solution space of a learning task. The solution space includes datasets from the application domain presenting relevant features for the learning task, a slice or subset of the initial dataset as training data, learning algorithms, and their (hyper)-parameters. A completed run's outcome often includes a trained model, the model performance measurement based on test data, and obtained predictions from an unseen slice or subset of the initial dataset.

We describe two levels within a run—the pre-model and post-model phases (see Fig. 2). These levels follow users' perceptions of activities carried out before generating a model and the evaluation-related activities. The physical representation

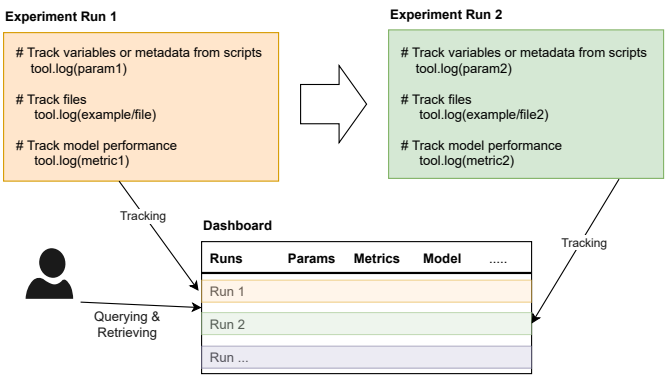

Fig. 1: Illustration of how experiment management tools work

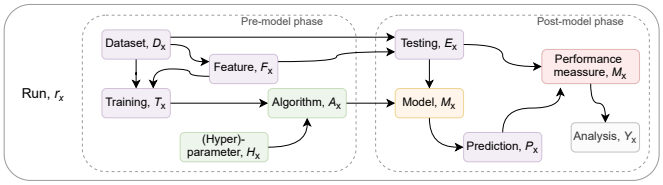

Fig. 2: A representation of a machine-learning experiment run

of assets used during an experiment varies depending on the platforms and tools used. For instance, users may use the exact text file representation for an experiment's (hyper)-parameters, algorithms, and features in small setups. Consequently, our representation of the run and assets here is a logical one.

B. Model-Driven Engineering & Meta-Modeling

Model-driven engineering (MDE) focuses on creating and exploiting models to produce software. With the help of MDE, it is possible to express software design using concepts that closely reflect the problem domain than the actual implementation technology. Meta-modeling is an essential concept in MDE, closely related to domain modeling (e.g., using UML diagrams in the early stages of software development). A meta-model provides a more formal representation of a domain model and precisely describes the possible model instances.

A meta-model determines the abstract syntax of a model, that is, an underlying data structure of concrete model instances [16]. Concrete model instances can be visualized in different concrete syntaxes (e.g., graphical and textual ones). To encode the knowledge collected in our analysis, we encoded it as a formal meta-model. To this end, we used *Ecore* from the Eclipse Modeling Framework (EMF) [17], a popular meta-modeling language in the MDE community [16].

C. Related Work

We now present the two main directions of related work.

Machine Learning Experiment Management Tools: Two main concerns of experiment management tools are traceability and reproducibility. Mora-Cantallops et al. [10] review nine relevant tools for their traceability in connection to building machine learning models and machine-learning-based systems. They found several tools supporting traceability and reproducibility but stated that the tools do not support a common

approach. Hence, the authors recommend that future research fill this gap, enabling interoperability across traceability tools through shared semantics. A survey by Serban et al. [18] that shows the level of adoption of the recognized engineering practices among 300+ practitioners highlights the importance of tracking predictions with model versions and input data (traceability), which is commonly supported by experiment management tools. Several empirical studies [7], [19]–[21] investigate experiment management tools for the support of reproducibility and comparison of their features, including StudioML, MLflow, Weights & Biases, Polyaxon, Comet.ml, Sacred, Sumatra, and DVC.

Artifact Meta-modeling is a model-driven technique for representing assets and their relations. By relying on artifact meta-models, tools may offer better support for working with diverse file types. Some studies adopt meta-modeling techniques to represent and understand assets managed by tools or within projects [22]–[25]

Hillemacher et al. [25] and Atouani et al. [24] introduce meta-models for machine learning and deep learning frameworks, aiming to provide a development foundation for automation and data management for developing software with machine learning components.

Samuel et al. [26], [27] develop the REPRODUCE ME ontology for reproducibility of general experiments using Semantic Web technologies. Including machine learning concepts is mentioned as a future work direction of high-interest [27].

Our work applies a meta-modeling technique comparable to [24], [25]. However, our work differs since we focus on concepts within machine learning experiments and runs, as opposed to finalized machine learning components. The meta-models from both works [24], [25] do not make the revision history of machine learning assets explicit; therefore, they cannot be used in the way our subject tools and ideas for new tools do. To this end, we applied a systematic methodology, extracting concepts from an analysis of our 17 subject tools.

III. Methodology

We now discuss our methodology, including the aspects of tool selection and the extraction of knowledge about critical structures and relationships supported by the subject tools.

A. Tool Selection

For this part of our methodology, we rely on our previous survey of machine-learning experiment management tools [7], which provides a suitable basis: It surveyed available tools and their high-level characteristics, following systematic literature review guidelines. We focus on the persistency and versioning aspects of the tools, which are critical for our considered scenarios. Table I shows the final selection, which includes nine cloud-based and eight standalone software tools.

TABLE I: Considered tools (following the selection from [7]).

Cloud Service	Standalone Software
Neptune.ml (netptune.ml)	Datmo (github/datmo)
Valohai (valohai.com)	Feature Forge (github/machinalis)
Weights & Biases (wandb.com)	Guild (guild.ai)
Determine.ai (determined.ai)	MLFlow (mlflow.org)
Comet.ml (comet.ml)	Sacred (github/IDSIA)
Deepkit (github/deepkit)	StudioML (github/open-research)
Dot Science (dotscience.com)	Sumatra (neuralensemble.org)
PolyAxon (polyaxon.com)	DVC (dvc.org)
Allegro Trains (github/allegroai)	-

B. Domain analysis & Validation

We proceeded with a domain analysis, in which we developed a meta-model, entitled *EMMM*, representing the superset of the concepts supported by our considered tools. The two main components of a meta-model are classes and references showing concepts of machine-learning experiment management tools classes and their relationships. The concepts and relationships were formulated based on a detailed manual analysis of the tools and how they support their features for asset management. When we observed conflicts between observations in two tools, we chose a higher-level concept that encapsulates them, in line with one of the main intentions of meta-modeling—reducing information complexity by abstraction [16]. For example, the specific metadata supported across the subjects are often conflicting in the sense that different tools support different types of metadata; so, we introduce the class `Metadata` that can represent any metadata type.

We carried out the above domain modeling in three phases: First, one author performed the initial design of the meta-model to establish its classes and their relationships. Then, we adopted an iterative process to refine the class relationships from the initial design. This involved weekly meetings of all authors, where we reviewed and iteratively improved the meta-model design until all authors approved the meta-model. We performed a validation phase where we populated our meta-model with concrete experiment information from actual experimental revision histories to reveal design flaws and identify improvement opportunities.

IV. EMMM: Proposed Meta-model

In this section, we present *EMMM* along with justifications for our design decisions. *EMMM* unifies essential characteristics, concepts, and relationships from 17 subject tools (see Table I). Figure 3 shows the resulting structure of our meta-model. Below, we describe *EMMM*'s different aspects.

A. Experiments & Experiment Runs

Experiment: Model development *experiments* consisting of multiple runs are the main concept in our subject tools. An experiment can be described as a project for a specific goal or learning task. For example, Neptune and PolyAxon refer to experiments as *projects*. The analyzed subjects support essential information about experiments with details such as names, descriptions, and requirements. Therefore, we present the class `Experiment` as the top-level asset of the meta-model. Some subjects allow custom metadata to store information such as tags, requirements, and authors. Accordingly, the

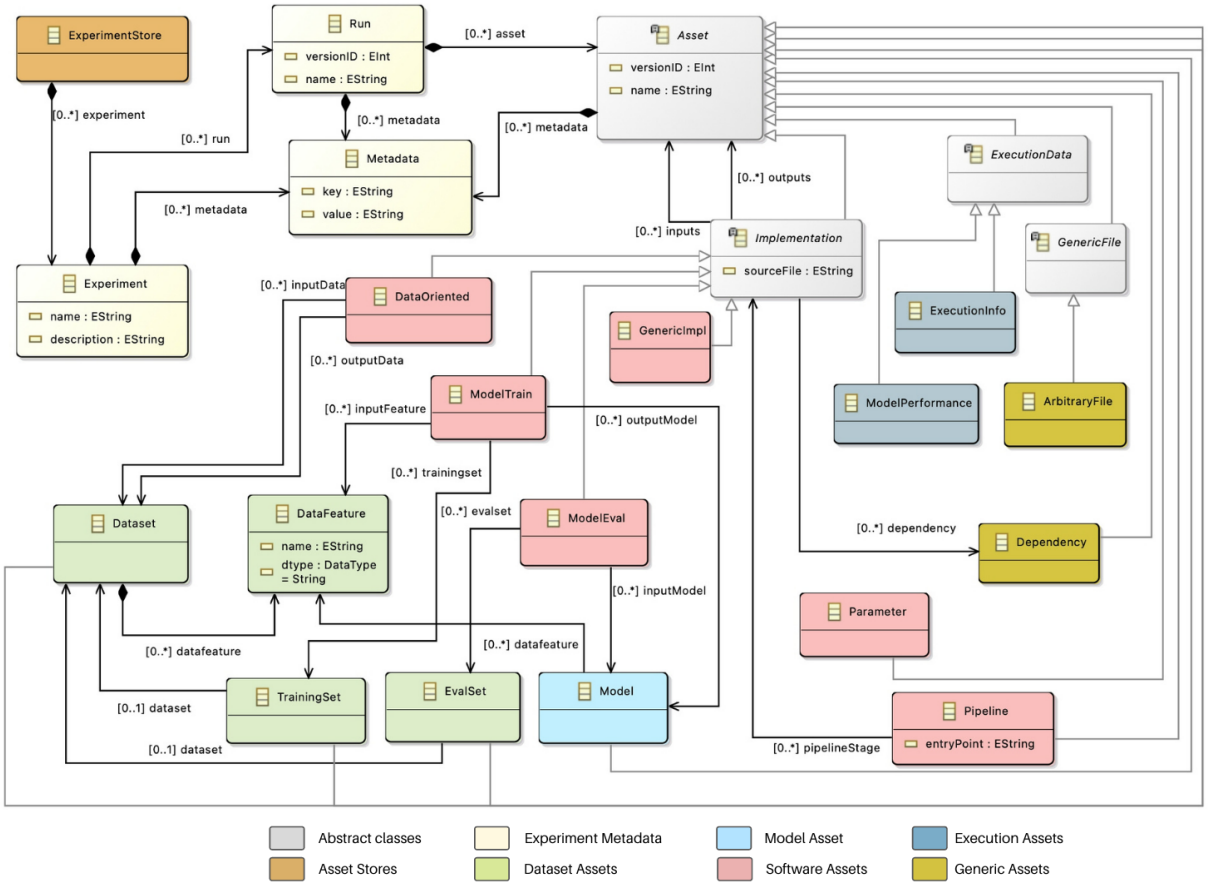

Fig. 3: *EMMM*: Metamodel unifying all asset types and their relationships extracted from the 17 subject tools.

class `Experiment` references multiple instances of the class `Metadata` to represent such information.

Experiment Runs: As the core abstraction of the subject tools, an experiment run represents the main asset type to which other asset types are associated (as illustrated in Figures 1 and 2). Our tools support the book-keeping of the exact versions of the assets used in a specific run. Consequently, our meta-model has the class `Run` with the attribute `versionId`. The class references `Metadata` to support storing additional metadata such as tags, descriptions, and notes on the thought process of decisions made for each run. These metadata instances can be useful for managing, comparing, or analyzing experiment runs. The `versionId` stores the incrementing count of experiment trials and is important for tracking and linking versions of other assets to an instance of the `Run` class. It represents the version id (similar to the unique commit id in traditional VCS) as a machine learning experiment evolves with constant updates and changes to its associated assets.

The class `Run` is related to all other experiment assets through association with the abstract class `Asset`. Through this abstract class, `Run` is associated with the following assets: (a) Datasets and features, (b) Implementation assets and their parameters, (c) Execution results and performance data from a run. A specific version of an asset can be shared by multiple instances of `Run`, or it can be unique to an instance of `Run`. Suppose that an asset is modified between two successive runs, then the previous and current versions of the asset are uniquely associated with the previous and current `Run` instances. On the other hand, if an asset were to be unchanged (while other assets are changed), a copy of the asset version is shared between previous and current run instances. However, the meta-model should not generate a new `Run` instance if nothing changes. Therefore, given the `versionId` of an instance of `Run`, the meta-model retrieves the snapshot version of all associated assets.

A `Run` instance is typically created after the execution of an experiment process finishes. For subjects such as Guild.ai and DVC, a `Run` instance is created when users invoke a command to execute the current `Run` instance. For DVC, "`$ dvc exp run`" executes the experiment process as presented in the current version of assets and creates a `Run` instance with an increment of the `versionId` attribute. Versions of assets modified afterward are associated with the new instance until another `Run` execution command is issued. Likewise, Guild provides the "`$ guild run`" command. An experiment run is associated with the execution of data-oriented, model training, and model evaluation tasks. At the execution point of such tasks, we associate all snapshots of assets with the current run.

B. Assets

The abstract class `Asset` supports modeling various concrete asset types used during experiments. The attribute `versionId`

tracks the different versions of assets created as an experiment evolves. The snapshot version of assets inherited from the abstract class has a one-to-one or one-to-many relationship with an instance of `Run`.

As summarized in Section III-A, there are four core groups of asset types observed in the subject tools: resource, software, metadata, and execution assets, which we represent as follows:

1) Resource assets: The core *resource* asset types are dataset and model. Many subjects prioritize these two primary assets of machine learning experiments. Datasets serve as input at multiple stages (i.e., data processing and model training stages) of machine learning experiments, and models generated from datasets serve as input for the model evaluation phase of experiments. In contrast to datasets and models, generic resources represent arbitrary assets of any type [7].

Dataset & DataFeature: We introduce the class `Dataset` as a kind of `Asset` that represents the input data available for an experiment. Since the subject tools often target supervised machine learning experiments, the datasets can be in tabular form, with features available as headers of each column of the datasets. The class `Dataset` contains multiple `DataFeature`, which store dataset's features. During an experiment, data transformations or modifications create new instances of `Dataset` while incrementing the value of its inherited `versionId` attribute. The list of `DataFeature` instances used from one `Run` instance to another varies when new features are engineered or removed. Irrespective of the state of `Dataset`, the instances of `DataFeature` used during a particular run may be a subset of or all the features contained in the latest `Dataset` instance. This justifies the need for a separate representation of data features as class `DataFeature` and their type. Since class `DataSet`, and `DataFeature` inherit from `Asset`, metadata describing the dataset (e.g., URI, author and data source) can be stored in the meta-model.

Data partitions: A crucial aspect of machine learning experiments involves the selection of data instances for training, testing, and validating the model performance. There are several approaches for this purpose, e.g., cross-validation [28]. Users may use different partitioning ratios between two different experiment runs. Also, the selection of instances can be random, making it difficult to recreate the exact selection later. Consequently, data partitioning methods available in machine learning frameworks often provide a 'random seed' argument to ensure consistent results. Some tools, including DVC, and Neptune, store data partitioning information as experiment parameters. To account for data partitioning, our meta-model presents classes `TrainingSet` and `EvalSet`, which are referenced by `ModelTrain` and `ModelEval` implementations, respectively.

Model: Machine learning models are trained from datasets using learning algorithms typically provided by development frameworks, such as TensorFlow (http://tensorflow.org) and SciKit-Learn. (http://scikit-learn.org) Some subjects, such as Comet.ml, Valohai, and Neptune support direct or indirect tracking and storing models and their metadata as asset types to support asset management concerns, such as traceability analysis. When a model is trained and evaluated within an experiment run, the snapshot versions of other assets (such as software and dataset features) are linked with the specific model. To support certain operations, such as model comparison across different runs, subject tools support the storage and tracking of `Model` as an asset type.

The subclass `Model`, by inheriting from `Asset`, can store model-related metadata. The class `Model` also references `DataFeature` to store information on the features or schema that the model supports. With the `Metadata` inherited by the class `Model`, information such as documentation of the model's characteristics or model's intended use cases and context can be stored. As with other parts of our meta-model, assets connected to a model can be easily retrieved through the class `Run`, which contains all its associated assets. For example, the resulting performance metrics and execution data when training a model can be fetched through the `versionId` of a `Run` instance that is associated with the `Model` instance in question.

Dependency: Information about the dependency of experiments on external libraries and environments is crucial for the subject tools since many offer reproducibility as a key functionality. Since sharing a dependency or environment with source code and dataset provides the highest level of reproducibility [29], subjects such as MLFlow track required dependencies to reproduce the previous experiment runs. Examples of supported dependencies are container or package management information, e.g., Docker container and Conda environments. Additional relevant information includes environment variables, host OS information, hardware details, and library versions used for experiments. Accordingly, we include class `Dependency`, referenced by `Implementation` assets, to store all dependency or environment-related files and metadata.

GenericFile: Pre-determining all information users may require to support their experiments seems impossible. Consequently, many subjects offer ways to track and manage arbitrary files. Examples of such information are credentials to authenticate external services that host other resources or application domain reports to help understand the experiment. Furthermore, some subjects provide a "one-size-fits-all" to track and manage resources (i.e., tracks all asset types without particular consideration for the specific types). Therefore, our meta-model has a concrete class `ArbitraryFile`, which inherits the abstract class `GenericFile`—a type of `Asset`— to store arbitrary files. The abstract class `GenericFile` must be extended to store other custom files.

2) Software assets: Software assets represent the scripts or source code used to implement the experiment's processes as one or more stages of a machine learning workflow. They also typically rely on the supporting machine learning frameworks or model development tools that provide a collection of general machine learning techniques, such as SciKit-Learn, PyTorch, TensorFlow, and Keras. Our meta-model reflects the support found in the subjects to store implementation assets through the classes `DataOriented`, `ModelTrain`, `ModelEval`, `GenericImpl`, `Parameter`, and `Pipeline`.

Implementation: Implementation assets represent text-based files with implementation details to carry out specific machine

learning operations. Version control systems (VCS), such as Git, are essential source code management tools commonly used when managing implementation assets of traditional software systems. For machine-learning experiment assets, there is a desire for alternative approaches that are better suited for the exploratory nature of machine-learning experiments [5]. Accordingly, many subjects integrate or build on existing VCS to offer tailored approaches. For example, DVC extends Git to provide methods to prepare experiment assets in stages and execute experiment runs while automatically tracking the versions of assets as the experiment evolves. Also, subjects such as MLFlow and Neptune interface with Git repositories to track their commit information (i.e., commit hash and messages) along with the states of other assets as experiments evolve.

Consequently, we include the class `Implementation`, pointing to a `sourceFile` taking other `Assets` as inputs or outputs. The classes `DataOriented`, `ModelTrain`, and `ModelEval` are types of `Implementation` based on the aspect of the machine learning workflow they represent. Since these classes inherit from abstract class `Asset`, the classes can store source code-related metadata. Classes `DataOriented`, `ModelTrain`, and `ModelEval` vary based on specific asset inputs and outputs they support. Class `DataOriented` represents implementation instances for preprocessing, transformation, or engineering of datasets, and it references class `Dataset` as input and output. The implementation of the training stage, where machine learning algorithms use training datasets to generate a new model, is represented by the class `ModelTrain`. It references instances of `TrainingSet` and `DataFeature` as its input, with class `Model` as its output. Implementation of the performance evaluation of a model is represented by the class `ModelEval`, which references the instances of `Model` and `EvalSet` as the model to evaluate and dataset to use for evaluation, respectively. The class `GenericImpl` represents other implementations such as model deployment or monitoring that are not represented by `DataOriented`, `ModelTrain`, and `ModelEval`.

Parameter: (Hyper-)parameters are special parameters controlling the model training process of a machine learning algorithm (e.g., learning rate, regularization, and tree depth). Most subjects support hyper-parameters tracking and management through custom metadata, while the subjects Comet.ml, PolyAxon, and Valoh.ai support hyper-parameters as asset types. Such tools offer hyper-parameter-specific features, including hyper-parameter tuning to facilitate the model-oriented stages of machine learning workflow. In addition, the parameter asset type represents other configurable parameters that may influence any aspect of experiment processes.

It is mostly the responsibility of the users to define necessary experiment parameters for the tools to track. Consequently, our meta-model has the class `Parameter` inherited from the abstract class `Asset`. The class `Parameter` can be referenced by `Implementation` as input through inheritance from `Asset`. For example, an instance of `Parameter` can serve as input for different instances of the class `ModelTrain`.

Pipeline: Subjects such as DVC support representing sequences of implementation assets to produce a complete machine learning pipeline. Here, the inputs and dependencies of each implementation stage are defined along with expected outputs in a direct acyclic graph. Consequently, our meta-model includes the class `Pipeline`, referencing sub-classes of `Implementation` as its stages. To represent a concrete pipeline, an instance of class `Pipeline` can store instances of `DataOriented`, `ModelTrain`, `ModelEval` or `GenericImpl` as sequence with their respective inputs and outputs.

3) ExecutionData assets: The abstract class `ExecutionData` represents execution-related information that the subject tools track explicitly or automatically when executing experiment processes. It inherits from `Asset` and can be referenced as output generated by the class `Implementation`.

ExecutionInfo: `ExecutionInfo` stores execution information as generally tracked by the tools, e.g., terminal outputs, logs, book-keeping information (e.g., progress, status, duration, and events), and live hardware consumption (e.g., CPU, GPU, and memory utilization).

ModelPerfomance: Most subject tools track the model performance information generated as the output of model evaluations. This is based on evaluation metrics, as tracked in different forms based on the machine learning task (e.g., sensitivity or ROC values for classification tasks; MSE, MAPE, or R^2 for regression tasks). Our meta-model stores model performance using class `ModelPerformance`, a subtype of `ExecutionData`.

4) Metadata: Metadata describes the descriptive and structural static information about machine learning experiments and their assets. The subject tools support metadata management for software assets, datasets, models, execution, dependencies, and experiments [7]. The supported metadata of assets can be predefined or custom-defined meta information. For example, *tag* is a type of metadata information supported in Neptune; however, users can create additional custom *key-value* metadata information as required to associate with experiment runs. Since metadata is core to the experiment management tools, all concrete classes of asset types have an association with the class `Metadata` through the abstract class `Asset`.

Experiment Stores: Class `ExperimentStore` represents the storage of all experiment-related information and assets. How subjects physically store the actual `Experiment` information differs: some use file systems, others use databases, clouds, or a combination of those. For instance, PolyAxon, DotScience, and MLFlow delegate the storage of software assets to external version control systems, such as Git. Similarly, several tools such as DotScience, MLFlow, and DVC support storage and tracking of datasets from different storage types, including distributed storage systems such as AWS S3. `ExperimentStore` objects can contain `Run` s, containing all assets used during a specific run. The physical storage of assets may differ based on the asset types. For instance, the class `Dataset` can be stored in separate storage specific to data, while the class `Model` can be stored in another storage specific to models and their relevant metadata. Similarly, the class `GenericFile` can also be stored separately.

V. Metamodel Evaluation: Modeling a real case

We validated our metamodel *EMMM* by instantiating it for a real case. In this section, we describe the details of this instantiation and how doing so led to further improvements. All evaluation artifacts are available via our online appendix: `https://github.com/emmm-metamodel/emmm`.

We instantiated our meta-model for a case of a submission to Kaggle, a machine learning competition community. The submission addressed the *Titanic survival* classification task with 18 experimental runs (`https://www.kaggle.com/code/hosamwajeeh/titanic-survival-crossval-81-89-score-0-78708`). To create the instance, we used the EMF-generated source code [17] for *EMMM*, which includes all required classes, packages, and factories. We fetched the Kaggle experiment, extracted the assets for the available runs, and added the assets to the instance of *EMMM*. We persisted the created instance as an XMI file through EMF's resource class. The interchange file viewer provides a way to review the meta-model for consistency with the source experiment data. We modified the meta-model for encountered inconsistencies.

Figure 4 shows an excerpt of the created *EMMM* instance. The root entity is the Experiment Store, which holds the Experiment "Titanic Survival Prediction," containing the 18 experiment runs and their assets. The figure shows the asset under the fifth experiment run, which are of different runs. For example, `Dataset` and `DataOriented` implementation have not been modified since *run 1*, while the second version of `Parameter` and `ModelEval` was used during *run 5*.

Validating *EMMM* with actual experiment data led to several improvements. While modeling the experiment, we solved two problems with prior meta-model versions: generating multiple `Models` from `ModelTrain` and evaluating multiple ones using `EvalModel` in a single run was previously not possible, which we fixed. Similarly, we updated the representation of the data-related assets and their associations with the software assets to easier modeling of experiment data.

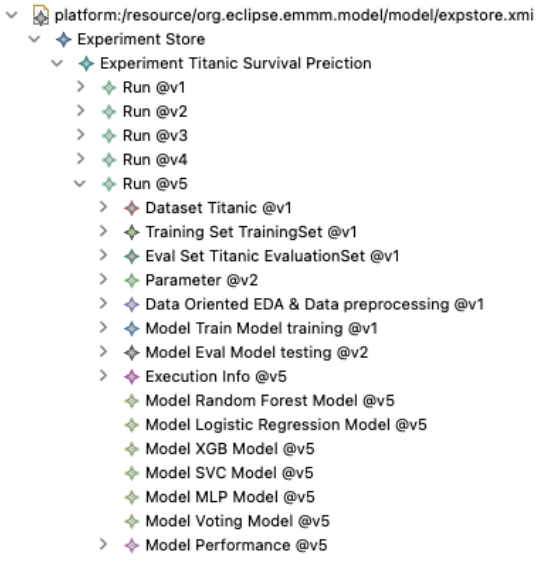

Fig. 4: *EMMM* instance showing an experimental run.

VI. Discussion

We discuss how *EMMM* provides a practical foundation for new and improved tools, including possible uses of *EMMM* and customization opportunities, as well as threats to validity.

Using EMMM: Our meta-model is a ready-to-use software artifact, formalized in Ecore, directly usable to facilitate tool development. Via the EMF-generated code [17] that we provide with the meta-model, it provides APIs and standard editors for manipulating its instances (shown in Section V). We foresee the following uses of EMMM:

(i.) *Enabling interoperability.* Lack of interoperability has been cited as a weakness of existing experiment management tools [10]. Our meta-model provides a foundation for enabling interoperability, offering an empirically informed representation of concepts from 17 tools. Developers of such tools can write import and export functions towards our meta-model; instead of one importer and exporter for each of the other tools, which might be prohibitively expensive.

(ii.) *Blueprint for developing new tools*. Extending available versioning tools such as Git towards native support for machine learning requires a conceptualization of machine learning projects. Our meta-model provides such a conceptualization. Developers of tool extensions could represent the machine-learning-specific information of a revision history as instances of our meta-model.

(iii.) *Connecting to available MDE tools and services*. By formalizing concepts of machine learning experimentation in a meta-model, our contribution bridges two technical domains of MDE and machine learning. In doing so, we make a plethora of MDE work [16] applicable to a new context in machine learning, e.g., tools for model analysis, simulation, refactoring, quality assurance, testing, and many others.

Customizing EMMM: The variety of existing tools [7] can be ascribed to different user needs and scenarios. Even though we have presented a unified meta-model, not all valid uses require the support of the meta-model in its entirety. Instead, it might be desirable that new tools implement support for a subset of the meta-model based on their specific needs. This leads to the notion of a *configurable* meta-model, in which a configuration can be described as views representing subsets of all the concepts and their relationships.

For example, we currently have tools that serve mainly as machine-learning model registries (e.g., ModelDB [30]) or those that integrate model registries (e.g., MLFlow, Vertai, and Neptune model registries). Such tools or aspects of tools focus on the class `Model` and its `Metadata`. So, configuring our meta-model with views on relevant assets can serve tools that require a subset of the meta-model. A configurable meta-data also provides opportunities for new kinds of tools. For example, tools with views on `DatasetFeature` can be used to trace model features back to concrete concepts within the application domain. Such information can be valuable knowledge for domain experts. In future work, we aim to extend *EMMM* to provide such configurable views.

Threats to Validity: Concerning *external validity*, while our meta-model represents concepts from 17 systematically selected tools, some tools may exist that have not been covered by our methodology, especially in a fast-moving field such as machine learning. Our considered tools are generally tailored towards supervised machine learning algorithms instead of other machine learning types, such as reinforcement learning. Future development of such tools will still benefit from the present unification effort since other machine learning algorithms generally share some characteristics with supervised machine learning (e.g., storing information about multiple experiment runs is essential for provenance questions). In addition, our use of just one case for the evaluation may limit external validity. On *internal validity*, our design decision justifications are based on the features and asset types observed in the subject tools. One threat is that our interpretations of different tool features are subjective. We mitigated this threat by adopting a methodology where one author designed the initial meta-model, which was iteratively reviewed and improved by all authors.

VII. Conclusion and Future Work

We proposed *EMMM*, a unified meta-model representing the asset types, their relationships, and their evolution history among machine learning experiments as observed in 17 machine-learning experiment management tools. We propose *EMMM* as a reference for tool developers and researchers seeking to improve existing tools or develop next-generation tools with native support for machine learning. *EMMM* presents a superset of conceptualized structures and their relationships extracted from our subject tools. Our meta-model can foster the improvement of tools and the development of new tools with native support for machine learning assets.

In the future, we plan to extend our meta-model to make it configurable for supporting multiple views for specific tool and user needs. Furthermore, we aim to validate it qualitatively, following evaluation methods from the assessment of taxonomies [31], [32], addressing a need for more comprehensive usefulness and effectiveness evaluations of machine learning asset management research [33]. A recent extension [34] of the survey that informed our tool selection [7] provides new cases (from research papers instead of in-practice tools) which could be addressed to ensure the completeness of the metamodel.

References

[1] X. Bouthillier and G. Varoquaux, "Survey of machine-learning experimental methods at neurips2019 and iclr2020," Tech. Rep., 2020.

[2] G. Hulten, *Building Intelligent Systems*. Springer, 2019.

[3] P. Janardhanan, "Project repositories for machine learning with tensorflow," *Procedia Computer Science*, vol. 171, pp. 188–196, 2020.

[4] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *VL/HCC*, 2016, pp. 162–170.

[5] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Software Engineering Challenges of Deep Learning," in *SEAA*, 2018, pp. 50–59.

[6] F. Kumeno, "Sofware engineering challenges for machine learning applications: A literature review," *Intelligent Decision Technologies*, vol. 13, no. 4, pp. 463–476, 2020.

[7] S. Idowu, D. Strüber, and T. Berger, "Asset management in machine learning: A survey," in *ICSE-SEIP*. IEEE, 2021, pp. 51–60.

[8] J. Jordan, "Organizing machine learning projects: Project management guidelines." Jan 2021. [Online]. Available: https://www.jeremyjordan.me/ml-projects-guide/

[9] L. Visengeriyeva, A. Kammer, I. Bär, and A. Plöd, "ml-ops.org," Jul 2021. [Online]. Available: https://ml-ops.org/content/end-to-end-ml-workflow

[10] M. Mora-Cantallops, S. Sánchez-Alonso, E. García-Barriocanal, and M.-A. Sicilia, "Traceability for trustworthy ai: A review of models and tools," *Big Data and Cognitive Computing*, vol. 5, no. 2, 2021.

[11] A. A. Ormenisan, M. Ismail, S. Haridi, and J. Dowling, "Implicit Provenance for Machine Learning Artifacts," *MLSys'20*, p. 3, 2020.

[12] I. H. Sarker, F. Faruque, U. Hossen, and A. Rahman, "A Survey of Software Development Process Models in Software Engineering," *International Journal of Software Engineering and Its Applications*, vol. 9, no. 11, pp. 55–70, 2015.

[13] R. Wirth, "CRISP-DM : Towards a Standard Process Model for Data Mining," *KDD*, no. 24959, pp. 29–39, 2000.

[14] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "The KDD Process for Extracting Useful Knowledge from Volumes of Data," *Commun. ACM*, vol. 39, no. 11, pp. 27–34, 1996.

[15] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *Ieee Network*, vol. 32, no. 2, pp. 92–99, 2017.

[16] A. Wasowski and T. Berger, *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*, 2022. [Online]. Available: http://dsl.design

[17] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[18] A. Serban, K. van der Blom, H. Hoos, and J. Visser, "Adoption and effects of software engineering best practices in machine learning," in *ESEM*, 2020.

[19] R. Isdahl and O. E. Gundersen, "Out-of-the-Box Reproducibility: A Survey of Machine Learning Platforms," in *eScience*. IEEE, 2019.

[20] R. Ferenc, T. Viszkok, T. Aladics, J. Jász, and P. Hegedűs, "Deep-water framework: The Swiss army knife of humans working with machine learning models," *SoftwareX*, vol. 12, p. 100551, 2020.

[21] T. Weißgerber and M. Granitzer, "Mapping platforms into a new open science model for machine learning," *it - Information Technology*, vol. 61, no. 4, pp. 197–208, 2019.

[22] A. Butting, T. Greifenberg, B. Rumpe, and A. Wortmann, *On the Need for Artifact Models in Model-Driven Systems Engineering Projects*, 01 2018, pp. 146–153.

[23] T. Greifenberg, S. Hillemacher, and K. Hölldobler, *Applied Artifact-Based Analysis for Architecture Consistency Checking*, 12 2020, pp. 61–85.

[24] A. Atouani, J. C. Kirchhof, E. Kusmenko, and B. Rumpe, "Artifact and reference models for generative machine learning frameworks and build systems," in *GPCE*, 2021, p. 55–68.

[25] S. Hillemacher, N. Jäckel, C. Kugler, P. Orth, D. Schmalzing, and L. Wachtmeister, *Artifact-Based Analysis for the Development of Collaborative Embedded Systems*, Cham, 2021, pp. 315–331.

[26] S. Samuel, "A provenance-based semantic approach to support understandability, reproducibility, and reuse of scientific experiments," Ph.D. dissertation, 12 2019.

[27] S. Samuel, F. Löffler, and B. König-Ries, "Machine learning pipelines: Provenance, reproducibility and fair data principles," in *IPAW*, 2020.

[28] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation." *Encyclopedia of database systems*, vol. 5, pp. 532–538, 2009.

[29] R. Tatman, J. Vanderplas, and S. Dane, "A Practical Taxonomy of Reproducibility for Machine Learning Research," *Reproducibility in ML Workshop, ICML'18*, no. Ml, 2018.

[30] M. Vartak, H. Subramanyam, W.-E. E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia, "ModelDB: a system for machine learning model management," in *the Workshop*. ACM Press, aug 2016, pp. 1–3.

[31] D. Bedford, "Evaluating classification schema and classification decisions," *Bulletin of the American Society for Information Science and Technology*, vol. 39, no. 2, pp. 13–21, 2013.

[32] P. Ralph, "Toward methodological guidelines for process theories and taxonomies in software engineering," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 712–735, 2018.

[33] S. Idowu, O. Osman, D. Strüber, and T. Berger, "On the effectiveness of machine learning experiment management tools," in *ICSE-SEIP*, 2022, to appear.

[34] S. Idowu, D. Strüber, and T. Berger, "Asset management in machine learning: State-of-research and state-of-practice," *CSUR*, 2022, to appear.