Towards ML-Integration and Training Patterns for AI-Enabled Systems

Sven Peldszus[1 [0000−0002−2604−0487]], Henriette Knopp[1 [0009−0003−5666−2222]], Yorick Sens[1 [0009−0002−5845−5887]], and Thorsten Berger[1,2 [0000−0002−3870−5167]]

[1] Ruhr University Bochum, Germany
[2] Chalmers | University of Gothenburg, Sweden

Abstract. Machine learning (ML) has improved dramatically over the last decade. ML models have become a fundamental part of intelligent software systems, many of which are safety-critical. Since ML models have complex lifecycles, they require dedicated methods and tools, such as pipeline automation or experiment management. Unfortunately, the current state of the art is *model-centric*, disregarding the challenges of engineering systems with multiple ML models that need to interact to realize complex functionality. Consider, for instance, robotics or autonomous driving systems, where perception architectures can easily incorporate more than 30 ML models. Developing such multi-ML model systems requires architectures that can integrate and chain ML components. Maintaining and evolving them requires tackling the combinatorial explosion when re-training ML components, often exploring different (hyper-)parameters, features, training algorithms, or other ML artifacts. Addressing these problems requires *systems-centric* methods and tools. In this work, we discuss characteristics of multi-ML-model systems and challenges of engineering them. Inspired by such systems in the autonomous driving domain, our focus is on experiment-management tooling, which supports tracking and reasoning about the training process for ML models. Our analysis reveals their concepts, but also their limitations when engineering multi-ML-model systems, especially due to their model-centric focus. We discuss possible integration patterns and ML training to facilitate the effective and efficient development, maintenance, and evolution of multi-ML-model systems. Furthermore, we describe real-world multi-ML-model systems, providing early results from identifying and analyzing open-source systems from GitHub.

Keywords: ML-Enabled Systems · ML Asset Management · ML Training · Maintenance · Evolution.

1 Introduction

Many recent advances in artificial intelligence (AI) allow building software systems for tasks that seemed impossible before. Especially machine learning (ML) has been the driving force behind these advances. Consider the field of autonomous driving, which benefited greatly from these advances. Modern autonomous driving systems use AI in a variety of ways. For instance, AI perceives

traffic lights, detects lanes, helps avoid obstacles on the road, and predicts traffic. Modern software systems can have many AI components, up to a whopping number of 28 ML models in the autonomous driving system Baidu Apollo [27, 4]. Since fully autonomous driving is still far away according to experts [12, 22], software systems incorporating even more ML models in different topologies and architectures can be expected.

Integrating one or many models in software systems poses new software-engineering challenges, since ML models differ significantly from traditional software artifacts. Models are developed (i.e., trained) highly iteratively based on data—the respective data-science methods and tools are entirely *model-centric* and *data-driven*. Traditional software-engineering methods and tools have not been designed for these circumstances [18]. This challenges managing different models in ML-enabled software systems, especially integrating the models and quality-assuring them. Training the models in such multi-ML-model systems gives rise to combinatorial explosion due to interactions [2], also since the models themselves are trained highly iteratively in so-called experiments. As such, managing many different models [16] requires managing many different experiments.

The integration of multiple ML models poses novel challenges, particularly for safety-critical systems. Consider again autonomous driving. Spectacular attacks—such as hackers being able to steer a Tesla into oncoming traffic using an attack as simple as putting some white stickers on the road [19]—illustrate the limitations in properly applying traditional software-engineering technologies in real, safety-critical systems [8]. Such systems typically use various sensors to perceive their environment and process the captured data using ML models to make safety-critical decisions. Each model used in such a system is trained for a specific task, such as detecting the road markings in a camera image. The performance of the final system however, does not depend on a single model, but on multiple, potentially interacting models and non-ML parts, each of which is potentially the weakest link in the chain. Furthermore, in practice, models are likely to be exposed to inputs that were not considered during training, raising the risk of incidents. To this end, it is unclear how we can systematically improve the overall performance and robustness of such AI-enabled systems [30].

Addressing these challenges requires novel *system-centric* methods that go beyond the *model-centric* methods from data science, which focus on individual ML models in isolation. We need best practices on how to integrate multiple ML models and how to decide which combination of models to improve (e.g., by retraining) to improve the overall system performance. Such methods need to support managing the necessary ML experiments, integrate them into a *system-centric* workflow, and consider the benefits to the whole system when training certain combinations of ML models.

As a first step towards such a *system-centric* method, we investigate the interaction between multiple ML models in AI-enabled systems. We discuss their characteristics and derive possible training patterns for *system-centric* training of ML models. These training patterns allow not only to consider the overall system performance as an essential aspect while training ML models, but also

to consider the interactions between ML models while training them. We discuss these training patterns, their potential benefits, but also their limitations in detail using the example of an autonomous racing car as described above. To enable the transfer to any AI-enabled system, we performed an exploratory survey on open-source software systems from GitHub, in which we identified different integration patterns for ML models. For each of these integration patterns, we relate the identified training patterns to them. We derive open challenges from our discussions and conclude with a research agenda.

2 Background

Before discussing how we can evolve from a *model-centric* to a *system-centric* development process, we provide an overview of current practices in ML model development and challenges discussed in the field of AI engineering.

To illustrate these challenges our running example is a Formula Student autonomous race car [1, 33]. It is similar to full-featured autonomous driving systems, such as Baidu Apollo. These race cars have to drive on a track marked by yellow and blue cones, so they do not have to deal with all the complexities of real traffic. However, such a race car still uses various sensors, such as cameras and LiDAR to perceive the track. These sensor information process is using multiple ML models [10]. Typically, the outputs of these models are fused into a world representation that the car can use to make driving decisions. Currently, each of the individual models is trained in isolation for a specific task, such as detecting the cones in the camera images.

2.1 Machine Learning

Recall that, machine Learning (ML) approximates a function from data, the so-called ML model. There are several subcategories of ML that differ in how said pattern or function is learned. For example, in supervised learning, a model is trained from a dataset that consists of input data and a set of labels on that data from which the ML model is approximated. The ML model can then make predictions about the labels of unseen input data [21].

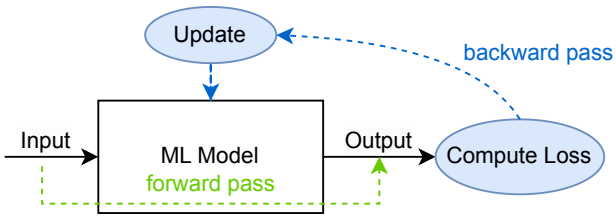


Fig. 1: High-level view on the steps of training an ML model

ML models are approximated iteratively from a set of training data in a process called ML model training that is realized in so-called experiments [21]. As shown in Figure 1, a training algorithm is implemented as a training loop that consists of the following steps. First, in a so-called *forward pass*, the current configuration of the ML model is used to compute the output (i.e., the labels) on a batch of the training dataset. Then, the deviation of the computed output from the expected output (i.e., labels in the dataset), called the loss, is computed. The loss is then used to update the model in a *backward pass*, with the goal of iteratively optimizing the ML model towards the function to be approximated. This training is repeated until an acceptable loss is achieved, i.e., a sufficiently low rate of false positive and false negative outputs.

In practice, models are trained in so-called experiments. An experiment consists of multiple runs where every run optimizes a model with differing sets of parameters. The resulting models from each of the runs are compared until a final model is selected to be used. Managing and comparing different experiment runs is a significant development effort and requires tools, called experimentation management tools, to adequately support it [16, 17, 14].

2.2 ML Frameworks

It is common to rely on ML frameworks to create and train ML models [31]. For this purpose, these frameworks provide a variety of support, ranging from implementing the most basic algorithms to providing fully functional ML models.

While frameworks such as PyTorch [28] and TensorFlow provide developers with the means to train ML models by providing implementations of common algorithms used in ML, the developer still needs to do a fair amount of development to create a working ML model. In contrast, Keras [9] is a high-level framework based on TensorFlow that allows easy implementation of ML models without too much background knowledge.

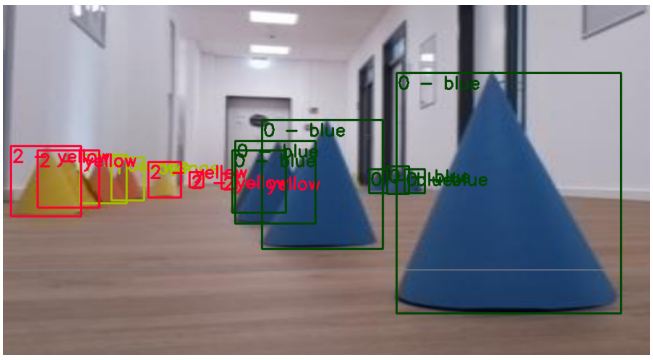


Fig. 2: Cones detected by an image-based perception model

Other frameworks provide pre-trained models and code to run them for specific tasks. For example, in autonomous driving, an essential task is to detect objects in the environment, such as the blue and yellow cones in the Formula Student example. An example of such a framework that focuses on efficiently implementing the task of object detection in images is YOLO [29]. Figure 2 shows an example of the output of such a YOLOv5 ML model for an indoor track oriented on the Formula Student. The objects—i.e., the cones—are detected in the form of bounding boxes that define the area in the image occupied by the detected objects. Usually, pre-trained models such as YOLO have to be trained on the concrete data and labels, e.g., the bounding boxes, that they are supposed to detect. In this case, we trained YOLO version 5 (YOLOv5) [29, 20] to detect the cones.

2.3 Development of ML-enabled Systems

The development of traditional software and training of ML models is very different. In traditional software system development functionality is programmed by hand and features are linked closely to the code. When training ML models their code is visible while the internals of ML models which realize the functionality are merely black boxes. ML model training is an iterative process that goes through different stages, including requirements analysis, data-oriented work, model-oriented work, and the DevOps phase. Several tools support the different phases of ML model development, including asset and experiment management [16, 17]. However, integrating the training of ML models with system development has been identified as a challenge among researchers [5, 3]. Especially the integration of a ML model into a systems environment is often difficult [24].

In particular, the quality of the training data is essential in ML development [24]. If the training data is not properly prepared, the training algorithm may not learn the correct pattern and make incorrect predictions on new, unseen data. Especially in safety-critical domains such as autonomous driving, where the system is frequently exposed to new inputs and new data, the development of robust ML models is critical [19].

2.4 ML Techniques using Multi-ML-Model Systems

Multiple ML models in a system have been considered in related work, but with a focus on the robustness of a single task implemented using ML. The two techniques that come to mind are *Ensemble Learning* and *Federated Learning*.

Ensemble Learning is a technique based on multiple ML models to obtain classifications that are more robust than those of single ML models [34]. The basic strategy is to train multiple classifiers, possibly using different configurations or frameworks, on the same problem and determine a final output using the outputs of all classifiers. While ensemble learning also uses multiple ML models, the main difference with multi-ML-model systems as we consider them is that multi-ML-model systems use multiple ML models to solve different and sometimes independent problems.

Federated Learning is a technique for decentralized training of ML models. Users or clients (i.e., mobile devices) train the ML model themselves with their own data. The individually trained models are then aggregated into a new model, on which training is then continued in a distributed fashion. Particularly relevant in the context of our running example of a race car is research on the benefits of federated learning for ML-based object recognition, especially how Yolov5 (cf. Sec. 2.2) can be implemented or even improved [11].

3 Integration of ML models in ML-Enabled Systems

Multi-ML-model systems are ML-enabled systems that incorporate more than one ML model. While ML-enabled systems are on the rise in a variety of domains, multi-ML-model systems are often found in safety-critical domains such as autonomous driving. Systems that integrate ML models pose new challenges for developers. Bosch et al. discuss challenges related to processes and workflows and outline a research agenda that includes the architecture, development and processes involved in AI engineering and also highlight domain-specific challenges [5]. Nazir et al. present insights on challenges related to the design and discuss different architectural patterns in engineering ML-enabled systems [25]. Lastly, Apel et al. discuss the issue of unexpected interactions of ML models when they are being reassembled after an initial decomposition of a complex task and the resulting integration testing [2]. Altogether, there is a need to obtain a better understanding of multi-ML-model systems.

3.1 Examples from the Autonomous Driving Domain

Baidu Apollo [4] is a real-world example of a multi-ML-model system from the domain of autonomous driving. A case study showed that the Apollo perception and prediction pipeline consists of 28 ML models [27]. For presentation purposes, however, we use a less complex running example: an autonomous Formula Student race car. Like the full-fledged autonomous driving system Baidu Apollo, the race car drives autonomously in its environment, but the latter is less complex, and therefore, requires fewer sensors and less ML models to process the sensor readings. In the remainder, we mainly focus on the ML-based perception architecture of our race car example.

Similar to Baidu Apollo, the autonomous race car uses a camera and a LiDAR to perceive its environment, i.e., the race track. For this task, it uses the YOLO model introduced above to detect cones that mark the race track in the camera images. Figure 3 shows a sketch of the race car's perception architecture. Sensor data flows from left to right, with the system receiving two inputs, *Images* and *LiDAR scans*, which are processed independently of each other using ML models. Later, the data is fused in a sensor fusion, as shown further to the right, resulting in the detected *cone positions* and corresponding *cone labels*, i.e., their colors, as the pipeline output.

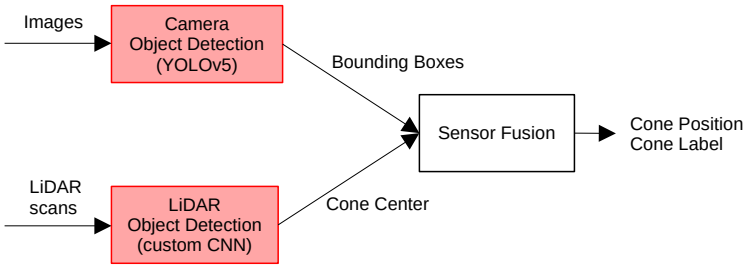


Fig. 3: ML-components of the running example

To train and manage the individual ML models, we used the open-source experiment-management tool MLFlow, which provides a Python API to log various metrics, which measure the models performance (i.e., recall or loss), during the training process. The results are stored and can be visualized using the MLFlow web interface. MLFlow is model-centric and does not support managing multiple ML models in systems, however. As we will show, this tool helps to keep track of individual training iterations of single models, it does not help to improve the overall system.

3.2 Integration Patterns of Multiple ML Models

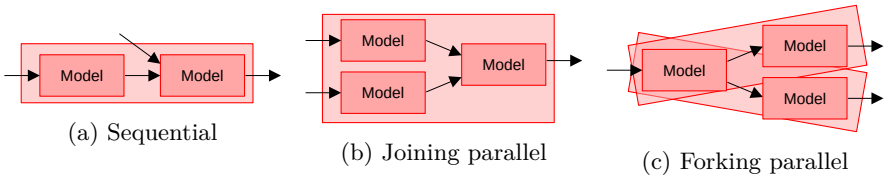
The perception architecture of the race car illustrates a multi-ML model-system that integrates two ML models that process different inputs by merging their outputs. However, it is not representative of all AI-enabled systems employing multiple ML models, and it is unclear in what other ways ML models can be integrated. In the remainder, we will refer to the ways in which multiple ML models are integrated as *integration patterns*.

To obtain a broader overview of how ML models are integrated in practice, we searched on GitHub for repositories that use one of the three major ML libraries TensorFlow, PyTorch, and Scikit-learn [31]. We obtained an initial sample of 763,820 repositories from which we extracted 359 applications by a series of automated filtering steps followed by a manual review of the remaining repositories. Roughly 40% of these systems use more than one ML library. A list of these repositories and the data collected about them is available on GitHub.[3]

To identify the systems that use multiple ML models, we analyzed how many trained models are stored in binary files in the repositories. However, only few repositories (17%) contained such model files. We learned that in practice, ML models are often stored externally and have to be manually downloaded from platforms such as Google Drive. However, we also observed systems that automatically download the ML models from a server. In most cases, however, custom code for training ML models is provided with the system instead of already trained models. Of those projects that store their model files in the repository, a median of 2 and an average of 5 ML models can be found.

[3] https://github.com/isselab/ML-Systems-Datasets

Although most repositories include training code for their models, the training of individual models appears to be independent of each other. Therefore, we focused our further analysis on the concrete implementations of the ML models and their integration. Source code belonging to model implementations was found in almost all subject applications, although the number of model implementations varied greatly.



(a) Sequential (b) Joining parallel (c) Forking parallel

Fig. 4: ML integration patterns observed in multi-ML model systems

We manually examined a sample of these systems, focusing on their integration of multiple ML models. We identified the three underlying integration patterns for interacting ML models shown in Fig. 4.

Sequential: This integration pattern describes an integration of ML models in which one model processes the output of another one, potentially, including additional information (Fig. 4a).

Joining parallel: Following this integration pattern, two or more models are deployed in parallel, and their outputs are merged either by a traditional component or another ML model (Fig. 4b). Our running example represents a system that implements this integration pattern.

Forking parallel: As shown in Fig. 4c, we found parallel models, that process the output of another ML model. While this third pattern could be seen as two overlapping sequential patterns, it comes with particular challenges when considering training on a system-level scope as outlined above.

An example of the *forking parallel* deployment pattern is a system that detects traffic lights and road signs for autonomous driving, which we found on GitHub.[4] The detection is implemented in two steps and involves three different ML libraries. In a first step, a YOLOv5 model is used to detect bounding boxes around potential objects of interest. Thereafter, the forking happens and different models are applied in parallel for classifying these objects of interest. A Scikit-learn model is used to detect traffic lights and their state, and a TensorFlow model created using Keras is used to classify road signs.

4 Training Patterns for Multi-ML Model Systems

In this section, we discuss the interaction of ML models in multi-ML-model systems and possible training patterns for improving the training of such multiple,

[4] https://github.com/JeffWang0325/Image-Identification-for-Self-Driving-Cars

interacting ML models. When considering the training of AI-enabled systems consisting of multiple ML models that are integrated with each other, such as the perception pipeline of our race car example, one can try to improve the training efficiency and effectiveness by not considering the different models independently. The idea is to make the training process more effective and more focused on the actual output of the system. This mainly concerns how the training process is structured and implemented, and what datasets need to be managed.

4.1 ML Model Interactions

To begin with, we will discuss interactions between ML models in our running example to identify potentials for improving their training. Usually, the individual models of our race car example would be trained in isolation and only thereafter be considered in combination in traditional software engineering tasks such as system design or quality assurance. If each of the included ML models is trained individually, as it is currently the case, we need two training datasets that consist of different data. First, we need images from the camera that are labeled with cones, as described in detail above. Here, each label is specified by five data points, first the color of the cone, which is encoded as the *class* of the bounding box, and the bounding box itself, which is described by four numeric values (*x_center*, *y_center*, *width*, and *height*). Second, we need LiDAR scans, again manually annotated with labels representing cones. In our case, each LiDAR scan is a 1D vector of distance measurements, and the cones are encoded by labeling each distance measurement as belonging to a cone or not. In total, cones have to be labeled in two datasets with six different values for each cone.

Not only is it a substantial effort to create the two required datasets, but by training in this way we are trying to optimize each of the models to its best performance in isolation. However, it is unclear how this affects the final output of the system. There are two factors to consider.

System performance: Although each model may performs optimally in isolation, we have no guarantee that the final output of the system will meet the performance requirements of the system, i.e., in terms of precision and recall of the cone detection needed by the race car for autonomous driving. Therefore, we need additional test cases to evaluate whether the performance requirements are met. However, if these are not met, it is not clear which model needs to be improved to meet the performance requirements.
On the other hand, for optimal system performance, it may not be necessary for each model to perform optimally in isolation. For example, in our case, we have precise positions for each cone in the LiDAR scans, and as long as the sensor fusion operates properly, it may not be necessary to optimize the sizes and positions of the bounding boxes for maximum accuracy. Also, false positives may not be a significant issue in the outlined application, since cones not detected by either ML model will not be included in the final output, while false positives can simply be discarded in the sensor fusion. Therefore, it would be beneficial to prioritize recall over precision in training.

While such considerations are obvious in our simplified example, they may not be in more complex systems.

Training cost: Building up on the considerations above, if we do not need optimal performance of the individual ML models, we could even reduce the effort needed for training and only fit the ML models until they reach the required performance. Unfortunately, however, we are not aware of any systematic way to determine this threshold upfront and one has to fall back to systematic and extensive integration tests.

According to the above considerations, it would be beneficial to train both models not against their individual outputs, but against the overall output of the system. First, it allows for a better integration of the two models as described above. Second, it could even help to reduce the maintenance effort of the required datasets. While in the isolated training of the models against their individual outputs we need two datasets described by six values together, we can reduce this to a single dataset if we would train against the final system output. Furthermore, in our running example, we could even describe the cones in this dataset by only three values, the x and y positions of the detected cones, and *labels* representing their colors. These three values are even easy to measure.

4.2 Training Patterns

In the following, we introduce training patterns that allow for a system centric training of the ML models. Based on the structure of the perception pipeline of our running example, we have identified four different training patterns for training multi-ML-model systems. We briefly describe each training pattern and what data is used where in the training process of our running example. Thereafter, we discuss the application to the other identified integration patterns.

Figure 5 shows in detail how the ML models of our running example are integrated and interact during training, and which data is relevant in which step of these interactions. The dashed lines represent the backward pass of the training, where data is propagated back through the system. First, according to the right part of the backward pass highlighted in red, the system output is first used to compute the loss of the pipeline, which we call the *end loss*. Then, the blue part at the top of the figure and the green part at the bottom show how the loss is traced back to each of the ML models, which are then updated. We can compute a deviation from expected outputs and predicted outputs, similar to a typical training algorithm. The figure also shows how the end loss is propagated back through the system. It consists of a combination of the error in predicting the position and color of the cones.

Our four identified training patterns are:

Individual Model Training: This training pattern represents the state of the art, where each ML model is optimized independently of its integration into the system. It is useful for ML models that are used as stand-alone models, such as the model used in the YOLO library. However, when the model is

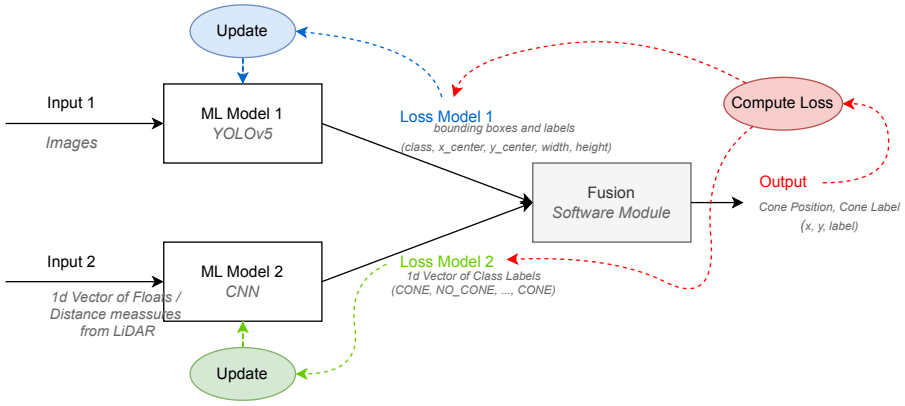


Fig. 5: Interactions among the ML models in our running example. (blue lines: update of ML model 1 / Yolov5 using loss 1; green lines: update of ML model 2 / CNN using loss 2; red lines: back propagation of the *end loss*)

integrated into the system, it may not be necessary to fine-tune every aspect. For example, in the case of the YOLOv5 model integrated into our race car, the training optimizes the position of the bounding boxes as well as the labeling. In this case, the exact position of the bounding box on the image data does not affect the overall system performance because the exact cone position is determined based on the more accurate LiDAR measurements.

Partial End-Loss Training: This training pattern puts a single ML model into the context of the system during training. It differs from the state of the art by considering its integration into a system. The idea is that instead of updating the ML model based on its direct output, as we did in *Individual End Loss Training*, we will follow the data through the system, compute the end loss and use this to update the model. This means that any software components used for pre- or post-processing must be integrated into the training algorithm. In our example, this would mean that the data would be fed into the system and processed by both ML models. After sensor fusion, the final result is used to update the LiDAR model, which is indicated by the backward pass consisting of the red and green markers in Fig. 5. Alternatively, we could also update the Yolov5 model in this way.

Simultaneous End-Loss Training: This training pattern extends the previous training pattern to include all ML models in the system. This means that all ML models are trained in the context of the system. The training infrastructure is the same as in the previous training pattern. The difference is that now all ML models are updated using this end loss. In each training iteration, data is passed through the system and processed by the ML models. The data is then further processed and the final result is used to update the models using the backward pass through the system. In Figure 5, we

update Yolov5 by following the red and the blue arrows and we update the CNN by following the red and the green arrows.

Alternating End-Loss Training: This training pattern is a modification of the previous training pattern. Instead of updating all ML models in each training iteration, we will update only one ML model. This variation reduces the computation time in each steps and allows for better traceability between the improvement in the end loss and model update.

The idea is that in some cases, one of the ML models will perform quite well, while the other will not. In our example, the end loss includes different features, such as the position and color of the cones. At least in our example, each of these features can be traced back to a different ML model. If the position of the cone is predicted incorrectly but the color is correct, it may make more sense to update the LiDAR ML model instead of the Yolov5 model. For reference, in Figure 5, we would follow the red arrows and then either the blue or green arrows, alternating the two updates.

Until now, we only considered our running example that implements a joining parallel integration of multiple ML models. In our investigation of multi-ML model systems at GitHub however, we also identified sequential and forking parallel ML model integration patterns. Like in our running example, the overall system performance of these systems depends on the interaction of the different ML models. For example, in the system that classifies traffic lights and road signs using models deployed following the forking parallel integration pattern, a traffic light or road sign that is missed by the first model, cannot be recovered by the subsequent models executed in parallel, but they can discard potential objects that are not traffic lights or road signs. Therefore, to improve safety, the first model should be trained mainly targeting a high recall or as proposed by us immediately concerning the overall system output. While these considerations are obvious for this example, they may be challenging for different application scenarios or more complicated tasks.

Since the sequential integration pattern can be seen as a subset of the discussed joining parallel integration pattern, the training patterns outlined for the joining parallel pattern can be applied without changes to train multiple ML models that are integrated sequentially. However, when ML models are integrated following the forking parallel integration pattern, the output of the first model is processed by multiple subsequent models, and therefore, the training patterns outlined above cannot be applied immediately to this integration pattern. In principal, the alternating end loss training could be applied and the training process iterates through all subsequent models. While this might work sufficiently well for an initial training, it would imply a huge effort if only one of the subsequent models does not perform well. Furthermore, since the first model would go through much more training iterations than the subsequent models, this may will lead to over fitting. Alternatively, we would need for example a loss function that can merge the outputs of all subsequent models for training the preceding model. However, this would still not solve the issue of only optimizing one of the subsequent models while not impacting the other ones. To

overcome this issue, one could fall back to performing multiple experiments again for the contained sequential patterns and, thereafter, use federated learning [35] for merging the multiple, individually trained, models of the forking parallel pattern into one that fits the needs of all subsequent models.

5 Research Directions

The detailed challenges outlined above give rise to the following research directions. We focus on the actual implementation of multi-ML systems intended to operate in safety-critical domains, such as autonomous driving.

5.1 Research Challenges

We now summarize the challenges we identified and concrete research directions.

Challenge 1: *Integrating multiple ML models into an AI-enabled system.* In our preliminary study of multi-ML-model systems on GitHub, we have identified underlying patterns for integrating multiple ML models into an AI-enabled system, but it is unclear whether the identified list is complete. Furthermore, best practices on how to actually realize these integration patterns as well as what are the advantages and disadvantages of concrete integration patterns are not systematically captured, yet. To effectively implement multi-ML model systems, developers need such best practices for integrating the multiple models with each other.

Challenge 2: *Robustness of AI-enabled systems against adversarial input.* Many AI-enabled systems operate in critical domains where it is essential to ensure their robustness against adversarial inputs to avoid dangerous behaviors such as steering into oncoming traffic [19]. While securing a system against adversaries is a well-known challenge [23, 26, 32], it becomes even more challenging due to the black box behavior of ML models [13]. Individual patterns, such as redundancy, have proven effective in improving robustness to adversarial input. However, we lack a systematic overview of such patterns as well as general techniques and processes for improving the robustness of AI-enabled systems. In particular, it is unclear how optimized training and data management help improve the robustness of AI-enabled systems.

Challenge 3: *Managing experiments in multi-ML model systems.* As discussed above, following the current state of practice in developing AI-enabled systems, one creates one training dataset per ML model and manages its experiments individually. However, the integration of multiple ML models leads to many interactions that need to be considered in the experiments. In practice, we have many training datasets that may be related, such as LiDAR data and camera images of the cones in our running example. It is unclear if and how this relationship needs to be reflected in the experiments. Furthermore, these interactions lead to significant scalability issues, since in the worst case we have to consider the cross product of all experiment runs for all ML models.

Challenge 4: *Training ML models on a system-wide scope.*

Although ML models can be an important part of an AI-enabled system, they are not an end in themselves. Instead, requirements always target concrete functionalities of a system, and ML models are only one technical solution to realize such functionalities. As discussed above, in many systems it is necessary to integrate multiple ML models, but training is still done on the basis of individual models against unclear internal requirements. Therefore, training should be shifted from a model-centric to a system-centric scope. Despite the outline of concrete training patterns that aim at this goal, how to actually achieve it remains an open challenge. This includes how each ML model influences the overall system performance, and how each ML model should be updated from a system perspective, particularly how to calculate a meaningful loss for each model.

Challenge 5: *Optimization of the system performance.*

The development of complex software systems is usually not a one-time task, but involves frequent iterations of implementing new functionality, testing and validating that functionality, and then improving it. Once ML models are integrated into such a software system, improving the functionality provided by the system may involve improving one or more of the models. However, pinpointing specific models that need to be retrained, especially in terms of what is missing, is currently a labor-intensive, manual task that requires tracing system-level tests back to the corresponding models. Providing support for this task remains an open challenge. Furthermore, whenever we retrain a model, it is currently unclear whether the entire system will benefit from this improvement, or whether it remains a local improvement that does not affect the entire system at all. Overall, the challenge is twofold. First, we need to identify architectures that support the overall system benefiting from model improvements, and second, we need to estimate this benefit in advance. The *Alternating End Loss Traing* proposed above could help address this challenge by considering models in combination and scheduling retraining passes according to which improvement would benefit the entire system. However, its feasibility and effectiveness remain to be demonstrated.

Challenge 6: *Testing of multi-ML-model systems.*

Testing of multi-ML-model systems is a frequently mentioned challenge. Because ML-models are trained deductively from examples, instead of inductively from fixed specifications, they are inherently difficult to test. Related works have addressed this issue by proposing ML-specific testing methods [6, 7], but these are not yet widely adopted. More work is needed in that direction, especially to evaluate the feasibility of these methods in actual software systems. Integrating multiple ML models in a software system further complicates the issue of testing, as additional integration tests are needed. In sequentially applied models, errors are propagated through the entire pipeline, which makes it hard to trace their origin.

5.2 Research Agenda

For addressing the challenges and the corresponding research opportunities summarized above, building up on the early ideas presented in this paper, we identified two concrete research directions that we will target next.

Integration patterns in practice. To obtain a better overview of the state of the art regarding the development of ML-enabled software, we are going to extend the mining study already discussed. Currently, the dataset used still contains a number of relatively small and immature software systems, many of which use only a single ML model. This could potentially be improved, by adapting the filtering criteria, such as filtering them by their popularity at GitHub. Furthermore, we will broaden the study by taking projects implemented in other programming languages into account.

Furthermore, although the integration patterns observed in our preliminary study are promising, we have to go deeper into detail. By extracting integration patterns from additional systems we could potentially find additional patterns. Also, by now we only have a high-level overview but have extract details on how to actually implement these patterns. We must assess how often each integration pattern is used in practice and in which context it is used. Based on this information, we can derive advantages and disadvantages of each pattern, especially in terms of performance and robustness as considered in Challenges 1, 2, and 5. Additionally, we can analyze the subject systems for the usage of further techniques to increase adversarial robustness, such as safeguards.

To get insights on how interactions among models are considered during their training and testing, we will study training code and datasets in depth. To this end, we can build upon the work of Idowu et al. [15] and their techniques to identify code that belong to different steps of the training process. This way, we aim to identify whether experiments are conducted together with system development or in separate repositories and how testing at a system-wide scope takes place (Challenge 3). The mapping of API calls to stages of ML development provides a basis for characterizing the relationship between these stages. We can, for example, determine if experiments are conducted on the entire pipeline or only the single models are refined in isolation, or if the output of one model is used as training data for the next one. Furthermore we can analyze the systems for testing practices, for example, the test coverage of ML-and non-ML code or the prevalence of model and data validation. We aim for identifying practices to address the issue of testing pipelines with multiple ML models (Challenge 4). This provides an empirical basis for addressing Challenge 6.

Building on top of this extended mining study, we could potentially conduct a survey with developers regarding the most pressing challenges they face and the reasoning behind the design choices we uncovered in the mining study. This could also help to put the discovered architectures and practices into context, separating good from bad practices.

Training (multiple) ML models on a system-centric scope. In this paper, we have already identified opportunities and discussed initial ideas for shifting the training of multiple ML models from a model-centric to a system-centric scope. In particular, the integrated training of multiple ML models following the outlined training patterns would allow to consider a system-centric scope also in the training of multi-ML systems, as targeted by Challenge 4. In this way, the testing associated with the training process of ML models would be further shifted towards a system scope, thus also addressing Challenge 6. Finally, as outlined above, system-centric training could help to reduce the effort needed to create and maintain training datasets, as well as to reduce the number of individual experiments needed to train the ML models of multi-ML systems (Challenge 3). To this end, we also want to assess how this idea is perceived by developers and how it improves their systems and training processes.

Building on the ideas presented in this paper, the next step is a prototypical implementation of the training patterns to evaluate the advantages and disadvantages of the proposed training patterns, particularly also concerning the robustness of the AI-enabled system to adversarial inputs (Challenge 1) and the impact on improving the system performance (Challenge 5). Here, the first challenge is that the ML model frameworks are not designed to be integrated into a system as we envision it. In other words, for our pipeline to work properly and for our ML models to be able to be trained, we need to merge the typical ML training loop with the execution of our system.

Next, we need to properly define the metrics and losses, and come up with a baseline against which to compare the effectiveness of the training patterns. We propose to use the *Individual Model Training* as a baseline. The losses need to be carefully defined and we will need to track what data each ML model needs and outputs as well as what information might be lost due to further processing. The question is how much of the data is left in the final result. One risk to consider is that the end loss may also not accurately reflect the performance of an individual model. Consider the following example. The end loss is the sum of the deviation in cone position and cone color. The Yolov5 model made an incorrect prediction, while the LiDAR model is already performing quite well and the cone position is predicted close to the expected output. If we update both models with the same loss, we are penalizing the LiDAR model for a good prediction. This is something that *Alternating End Loss Training* could fix and might be a peril in *Simultaneous End Loss Training*.

After properly defining the metrics and losses, we need to evaluate the benefits of this system-centric training. Assuming that we can practically train ML models that have the same performance and robustness as those trained using traditional training, the aspects we need to evaluate and compare are mainly the number of iterations used for training and the computational time. Another factor is the amount of data required. In other words, how much is the dataset reduced when training the pipeline as a whole. We believe that the reduction is significant, but we do not have anything tangible. Another aspect is the gen-

eralization of our proposed training patterns to other integration patterns that differ from our example.

6 Conclusion

We presented an overview of how multiple ML models are integrated in AI-enabled systems and how we can shift their training from a model-centric to a system-centric scope. Our long-term goal is to gain a better understanding of how we can rethink the development and training process for multi-ML model systems and enable software engineers without extensive background in data science to develop high-quality multi-ML systems. To this end, we want to be able to make recommendations on how to simplify the training process based on the context of the system. As a first step, we identified integration patterns of ML models in AI-enabled systems, as well as training patterns for shifting the training of ML models from a model-centric to a system-centric technique. In doing so, we identified open challenges in systematically integrating multiple ML models into a robust AI-enabled system and ensuring its performance and robustness that need to be addressed by future research. To this end, in our upcoming research, we will evaluate the identified training patterns and ML model interactions in real-world multi-ML systems. In addition, we will further evaluate the underlying integration patterns in ML-enabled software systems. We will analyze a large set of ML-enabled systems and present our results in a separate mining study.

References

1. FSG Competition Handbook 2024. Tech. rep., Formula Student Germany (2023)
2. Apel, S., Kästner, C., Kang, E.: Feature Interactions on Steroids: On the Composition of ML Models. IEEE Software **39**(3), 120–124 (2022)
3. Arpteg, A., Brinne, B., Crnkovic-Friis, L., Bosch, J.: Software Engineering Challenges of Deep Learning. In: Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 50–59 (2018). https://doi.org/10.1109/SEAA.2018.00018
4. Baidu: Apollo Auto: An Open Autonomous Driving Platform. https://github.com/ApolloAuto (2024)
5. Bosch, J., Olsson, H.H., Crnkovic, I.: Artificial Intelligence Paradigms for Smart Cyber-Physical Systems, chap. Engineering AI Systems: A Research Agenda, pp. 1–19. IGI Global (2020). https://doi.org/10.4018/978-1-7998-5101-1.ch001
6. Braiek, H.B., Khomh, F.: On Testing Machine Learning Programs. Journal of Systems and Software (JSS) **164**(110542) (2020). https://doi.org/10.1016/j.jss.2020.110542
7. Cheng, C.H., Huang, C.H., Yasuoka, H.: Quantitative Projection Coverage for Testing ML-enabled Autonomous Systems. In: International Symposium on Automated Technology for Verification and Analysis (ATVA). pp. 126–142 (2018)
8. Chernikova, A., Oprea, A., Nita-Rotaru, C., Kim, B.: Are Self-driving Cars Secure? Evasion Attacks against Deep Neural Networks for Steering Angle Prediction. In: IEEE Security and Privacy Workshops (SPW). pp. 132–137 (2019)

9. Chollet, F., et al.: Keras (2015), https://github.com/fchollet/keras
10. Gong, H., Feng, Y., Chen, T., Li, Z., Li, Y.: Fast and Accurate: The Perception System of a Formula Student Driverless Car. In: International Conference on Robotics, Control and Automation (ICRCA). pp. 45–49 (2022). https://doi.org/10.1109/ICRCA55033.2022.9828892
11. Hegiste, V., Legler, T., Ruskowski, M.: Federated Ensemble YOLOv5 – A Better Generalized Object Detection Algorithm. ArXiv (arXiv:2306.17829) (2023). https://doi.org/10.48550/ARXIV.2306.17829
12. Heineke, K., Kampshoff, P., Mkrtchyan, A., Shao, E.: Self-Driving Car Technology: When Will the Robots Hit the Road? Tech. rep., McKinsey & Company (2017), https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/self-driving-car-technology-when-will-the-robots-hit-the-road
13. Hu, Y., Kuang, W., Qin, Z., Li, K., Zhang, J., Gao, Y., Li, W., Li, K.: Artificial Intelligence Security: Threats and Countermeasures. ACM Computing Surveys **55**(2), 20:1–20:36 (2023). https://doi.org/10.1145/3487890
14. Idowu, S., Osman, O., Strueber, D., Berger, T.: Machine Learning Experiment Management Tools: A Mixed-Methods Empirical Study. Empirical Software Engineering (EMSE) (2024)
15. Idowu, S., Sens, Y., Berger, T., Krüger, J., Vierhauser, M.: A Large-Scale Study of ML-Related Python Projects. In: Symposium On Applied Computing (SAC) (2024), https://api.semanticscholar.org/CorpusID:267375897
16. Idowu, S., Strueber, D., Berger, T.: Asset Management in Machine Learning: State-of-research and State-of-practice. ACM Computing Surveys (2022)
17. Idowu, S., Strueber, D., Berger, T.: EMMM: A Unified Meta-Model for Tracking Machine Learning Experiments. In: Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 48–55 (2022). https://doi.org/10.1109/SEAA56994.2022.00016
18. Idowu, S., Strüber, D., Berger, T.: Asset Management in Machine Learning: A Survey. In: International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 51–60 (2021). https://doi.org/10.1109/ICSE-SEIP52600.2021.00014
19. Jing, P., Tang, Q., Du, Y., Xue, L., Luo, X., Wang, T., Nie, S., Wu, S.: Too Good to Be Safe: Tricking Lane Detection in Autonomous Driving with Crafted Perturbations. In: USENIX Security Symposium. pp. 3237–3254 (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/jing
20. Jocher, G.: YOLOv5 by Ultralytics (2020). https://doi.org/10.5281/zenodo.3908559, https://github.com/ultralytics/yolov5
21. Jordan, M.I., Mitchell, T.M.: Machine Learning: Trends, Perspectives, and Prospects. Science **349**(6245), 255–260 (2015). https://doi.org/10.1126/science.aaa8415
22. Liu, L., Lu, S., Zhong, R., Wu, B., Yao, Y., Zhang, Q., Shi, W.: Computing Systems for Autonomous Driving: State of the Art and Challenges. IEEE Internet of Things Journal **8**(8), 6469–6486 (2021). https://doi.org/10.1109/JIOT.2020.3043716
23. McGraw, G.: Software Security. IEEE Security & Privacy **2**(2), 80–83 (2004). https://doi.org/10.1109/MSECP.2004.1281254
24. Nahar, N., Zhang, H., Lewis, G., Zhou, S., Kästner, C.: A Meta-Summary of Challenges in Building Products with ML Components – Collecting Experiences from 4758+ Practitioners. In: International Conference on AI Engineering – Software Engineering for AI (CAIN). pp. 171–183 (2023). https://doi.org/10.1109/CAIN58948.2023.00034

25. Nazir, R., Bucaioni, A., Pelliccione, P.: Architecting ML-Enabled Systems: Challenges, Best Practices, and Design Decisions. Journal of Systems and Software (JSS) **207**(111860) (2024). https://doi.org/10.1016/J.JSS.2023.111860
26. Peldszus, S.: Security Compliance in Model-driven Development of Software Systems in Presence of Long-term Evolution and Variants. Springer (2022)
27. Peng, Z., Yang, J., Chen, T.H., Ma, L.: A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 1240–1250 (2020)
28. PyTorch: Open Source Machine Learning Framework. https://pytorch.org (2023)
29. Redmon, J., Divvala, S.K., Girshick, R.B., Farhadi, A.: You Only Look Once: Unified, Real-Time Object Detection. In: Conference on Computer Vision and Pattern Recognition (CVPR). pp. 779–788 (2016). https://doi.org/10.1109/CVPR.2016.91
30. Shafique, M., Naseer, M., Theocharides, T., Kyrkou, C., Mutlu, O., Orosa, L., Choi, J.: Robust Machine Learning Systems: Challenges,Current Trends, Perspectives, and the Road Ahead. IEEE Design & Test **37**(2), 30–57 (2020). https://doi.org/10.1109/MDAT.2020.2971217
31. Stančin, I., Jović, A.: An Overview and Comparison of Free Python Libraries for Data Mining and Big Data Analysis. In: International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). pp. 977–982 (2019). https://doi.org/10.23919/MIPRO.2019.8757088
32. Tuma, K., Peldszus, S., Strüber, D., Scandariato, R., Jürjens, J.: Checking Security Compliance between Models and Code. Software & Systems Modeling (SoSyM) **22**(1), 273–296 (2023). https://doi.org/10.1007/S10270-022-00991-5
33. Valls, M.I., Hendrikx, H.F., Reijgwart, V.J., Meier, F.V., Sa, I., Dubé, R., Gawel, A., Bürki, M., Siegwart, R.: Design of an Autonomous Racecar: Perception, State Estimation and System Integration. In: International Conference on Robotics and Automation (ICRA). pp. 2048–2055 (2018). https://doi.org/10.1109/ICRA.2018.8462829
34. Webb, G.I., Zheng, Z.: Multistrategy Ensemble Learning: Reducing Error by Combining Ensemble Learning Techniques. IEEE Transactions on Knowledge and Data Engineering (TKDE) **16**(8), 980–991 (2004). https://doi.org/10.1109/TKDE.2004.29
35. Zhang, C., Xie, Y., Bai, H., Yu, B., Li, W., Gao, Y.: A Survey on Federated Learning. Knowledge-Based Systems **216** (2021). https://doi.org/10.1016/J.KNOSYS.2021.106775