Cross-Platform Edge Deployment of Machine Learning Models: A Model-Driven Approach

Albin Karlsson Landgren[1], Philip Perhult Johnsen[1], Daniel Strüber[1,2*]

[1*]Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden.
[2]Department of Software Science, Radboud University, Netherlands.

*Corresponding author(s). E-mail(s): danstru@chalmers.se;
Contributing authors: albk@student.chalmers.se;
perhult@student.chalmers.se;

Abstract

Deploying machine learning (ML) models on edge devices presents unique challenges, arising from the different environments used for developing ML models and those required for their deployment, leading to a gray area of competence and expertise between ML engineers and application developers. In this paper, we explore the use of model-driven engineering to simplify the deployment of ML models on edge devices, specifically smartphones. We present a DSL for the specification of the ML serving pipelines (pre- and postprocessing of data before and after inference), together with a model interpretation approach that allows to make changes to the pipeline during runtime, thus removing the need to re-release an application upon changes to a pipeline. We followed a design science approach, in which we elicited requirements through an initial artifact study and interviews with engineers at an industrial partner. This was followed by the design and implementation of a lightweight, JSON-based domain-specific language designed to describe ML serving pipelines, along with an accompanying Flutter library to execute the pipelines during runtime. A preliminary evaluation with four developers shows the potential of this approach to increase development speed, decrease the amount of code required to make changes to an ML serving pipeline, and make less-experienced engineers more confident contributing to the domain.

Keywords: model-driven engineering, MDE4AI, AI engineering, mobile applications

Author preprint for a paper accepted for publication in Software and Systems Modeling (SoSyM)

1 Introduction

When deploying a machine learning (ML) model[1], the choice of target platform can have a significant impact on the ease of deployment. Models are often designed, trained, and deployed in a Python environment using libraries such as TensorFlow or PyTorch [1], with the same group of engineers maintaining control during the entire process. However, there are cases where the same toolchain or platform is not available throughout the whole process. One such example arises when deploying machine learning models on edge devices, specifically smartphones. In such scenarios, there is a chance that the training and deployment environments are different and that the deployment environment only contains the already trained models. Thus, the engineers deploying the models to applications might not know how to format model inputs and outputs. Therefore, they might land in a gray area of competence and responsibility between those developing and deploying the models. The ML engineers developing the models have most of the knowledge related to the actual models, while the application engineers know the deployment platform, but might lack the crucial context required to effectively deploy the model.

To add to the development and deployment scenario, further complexity arises since the development is happening in a cross-platform environment, a consequence of smartphone applications being developed for separate platforms (typically, iOS and Android). By bridging the gap between the different engineering roles and simultaneously reducing the amount of equivalent code being written twice, the intention is to improve knowledge-sharing across engineering teams, improve developer experience, and facilitate experimentation and flexibility in development.

In this paper, we aim to simplify the process of deploying ML models on edge devices, specifically smartphones. When, for example performing ML inference on image data from a smartphone camera, a series of pre- and post-processing steps is required before and after performing the inference, as different models require inputs of different shapes and output results of different shapes. These steps collectively form a pipeline, referred to as an *ML serving pipeline*.

We explore the use of model-driven engineering to facilitate the development and maintenance of ML serving pipelines on smartphones using model-driven engineering (MDE). Specifically, we propose a domain-specific language (DSL) and accompanying library. The DSL allows developers to easily specify and make changes to ML serving pipelines deployed on smartphones. The accompanying library supports the execution of the pipeline using platform-specific functionality, based on the contents of a model (DSL instance). The DSL acts as a platform to facilitate a shared understanding of an ML serving pipeline between the ML engineer and application engineer.

We conducted this work in collaboration with Wiretronic AB and their AI division in Gothenburg. Wiretronic develops image-based ML products (e.g. object detection and computer vision), including a suite of smartphone applications [2, 3]. As a part of this product palette, the company deploys ML models directly on devices, where the

[1]In this paper, the term “model” generally refers to machine learning models. When we address software models in the sense of model-driven engineering, we generally use the term *DSL instances*, except for specific contexts, e.g., *model interpretation*, which is a fixed technical term, explained in Section 2.

variety in architecture and operating systems can make it more difficult to deploy software effectively. The applications and corresponding libraries developed at Wiretronic are written in Flutter [4], a library for the Dart programming language enabling cross-platform development of native apps for iOS and Android. Whilst a majority of the functionality can be developed in Flutter, some aspects require writing platform-specific code. This often entails working directly with hardware resources on the device, such as the device's camera, or hardware-optimized ML libraries [5, 6]. Thus, the developer is required to write equivalent implementations for two platforms, underlining the challenges in deploying ML models on smartphones. Since the platform-specific code is constrained to a very specific domain, we saw the potential that this the workflow can be enhanced. By lifting the development to a suitable level of abstraction and making use of model-driven engineering techniques our industry partner could avoid having to write equivalent, domain-specific code for multiple platforms.

The deployment of ML models on smartphones or edge devices, in general, can cause problems with maintenance and updates. When deploying an ML model on a centralized server, the developer can have near full control over that server and perform updates as needed, without users noticing or requiring manual work. Meanwhile, when developing and deploying ML models for smartphones, this process is non-trivial. If the ML model and related functionality are bundled and shipped with the application when installed on a user's device, we must re-publish the application to the app store of each platform upon making changes or updates and the user must reinstall the application. An alternative method to this is to view the model as an asset to fetch from the application, allowing for easier updates. However, this still requires developers and users to perform the update process if a new version of the ML model requires a different serving pipeline.

In combination, these issues create problems with knowledge-sharing, developer experience, and flexibility when working with ML models on smartphones. It is unclear who is responsible and most suited to handle the deployment of the ML models, the code often has to be written for two separate platforms, and then subsequently re-deployed for these separate platforms' app stores. By using a DSL and model-driven engineering, we can address communication difficulties, create a single source of truth for the ML serving pipeline, and decrease development as well as deployment efforts. Subsequently, this can also improve the end-user experience since updates to the ML performance can occur without them noticing or having to update the application.

The study is guided by the findings from studying the role of MDE in edge-deployed ML and applied to the specific context of cross-platform mobile development. We assume a scenario in which an ML model is available as a result of a model development process, which may be carried out by training a new model from scratch, or by fine-tuning a pre-trained model. Our focus is on the deployment of this ML model to edge devices with multiple different platforms. Specifically, we will explore how a DSL can be designed and utilized to enhance this deployment, centered around the following research questions:

- **RQ1:** How can we design a domain-specific language (DSL) for an ML model with its required inputs, outputs, and pre- and post-processing stages?

- **RQ2:** How can we best implement and utilize the DSL in a concrete setting, specifically in the development of cross-platform mobile applications?
- **RQ3:** To what extent does the introduction of a DSL and an accompanying library improve the developer experience in the aspects of maintenance, feature development, time-saving, and resource planning?

This paper is accompanied by an online artifact [7] containing the implementation code as well as evaluation artifacts and data.

This rest of this paper is structured as follows: In Sect. 2, we introduce necessary background. In Sect. 3, we introduce our overall methodology. Section 4 is devoted to describing the design and implementation of our solution. Section 5 describes the detailed methodology and results for our evaluation. In Sect. 6, we discuss our results, before discussing related work in Sect. 7 and concluding in Sect 8.

2 Background

We now present relevant background of this work.

2.1 Edge-deployed Machine Learning

Edge-deployed ML refers to deploying ML models on edge devices instead of on a centralized server. An edge device can, for example, be a smartphone or Internet of Things (IoT) device, which generally has far simpler hardware than a server in a data center. The deployment of ML models on edge devices has become significantly more common in recent years thanks to advancements in both software and hardware [8–10]. Despite the less advanced hardware, deploying machine learning models on edge devices presents several advantages when compared to a centralized approach. Transmitting potentially sensitive or private data to a remote server introduces the risk of data leakage, with a fault in the remote system potentially leading to personal or financial consequences [9]. Additionally, eliminating the need for connecting to an external service for ML inference can improve both latency and reliability, as the potential bottleneck introduced by a weak network connection is removed. Despite these improvements, deploying ML models on edge devices, specifically smartphones, is not straightforward. One reason for this is the heterogeneity of underlying architecture [11]. A wide selection of libraries to deploy ML models on smartphones exists, but they each perform differently depending on the device's hardware configuration. A difference in cache size or GPU capacity can cause two libraries accomplishing equivalent tasks to perform differently, and with the wide range of hardware configurations present in the market, it is difficult to develop a solution optimal for every device [11]. Additionally, opting to deploy an ML model on devices instead of in a centralized environment can create obstacles to improvement and maintenance. When deploying an ML model in a centralized environment, the developer has full control over software and hardware and can develop the artifacts surrounding the model for a single environment. If the development is instead targeted at smartphones, the model has to be deployable on both iOS and Android devices, which each have distinct underlying architectures for deploying custom ML models [5, 6].

2.2 Cross-Platform Mobile Development

Mobile developers targeting both iOS and Android users may opt for a cross-platform framework, which enables the creation of separate, native builds for both platforms from a single codebase. The two most widely used frameworks for this purpose are Flutter and React Native [12]. Flutter is an open-source framework for the Dart language maintained by Google, while React Native is a JavaScript framework maintained by Meta. While there are some differences in architecture, both frameworks abstract away platform-specific details, allowing developers to focus on a single, platform-agnostic codebase. This abstraction layer translates the shared code into platform-specific components, aiming to achieve native functionality and performance for both iOS and Android. When requiring access to platform-specific features, often hardware, developers can opt to write native code for each platform. Although React Native has a new architecture in development that will allow for easier communication between the cross-platform and native layers, current implementations of both Flutter and React Native require serialization of data for inter-layer communication [13, 14].

2.3 Model-Driven Engineering (including Model Interpretation)

Model-Driven Engineering (MDE) reshapes software engineering by emphasizing high-level abstraction and model-centric approaches. In particular, the abstractions offered by MDE through modeling languages abstract away platform-specific details, facilitating easier adaptation of new technologies and development for multiple-platform environments [15]. There are several approaches to go from a model to executable software. One such approach commonly applied is code generation, where a model is transformed into a program in a suitable language that can subsequently be executed. This process allows for developer intervention where needed since the generated code can be edited upon generation. Furthermore, as a consequence of the code being generated before execution, this approach does not introduce any run-time overhead. However, while avoiding performance overhead, a disadvantage of code generation is having to re-generate and re-deploy the software if making changes to a model [15].

As an alternative to code generation, software development can be automated in MDE using model interpretation [15]. Model interpretation does not generate code, it instead implements a generic engine, e.g. a library, that parses and executes the model on the fly. This comes with several advantages as noted by Brambilla et al. [15]: it allows making changes to the model or engine without an added code generation step, easier portability between platforms, and not having to interact directly with the source code. Possible disadvantages are a negative impact to performance, due to the overhead of the interpretation engine, as well as potentially making the system more difficult to debug, as bugs could be in the interpretation engine itself.

2.4 Domain-Specific Languages (DSLs) with JSON Schema

Domain-specific languages (DSLs) are widely used in model-driven engineering to support the specification of software at exactly the right level of abstraction [16]. A DSL is a custom-tailored language for a particular domain of expertise. Developing a DSL

involves the definition of an *abstract syntax* that defines the concepts the language consists of, and one or several *concrete syntaxes* defining the appearance of the language (typically textual or visual) and the way it is edited [15]. Developing DSLs can be done in both language workbenches, such as Xtext [17], GEMOC, and MetaEdit+, or by using more lightweight approaches, such as a JSON Schema [18]. Language workbenches support the automated generation of comprehensive editor tooling for a given DSL. However, while the resulting editors are powerful, they typically require the commmitment to specialized technologies and platforms, which can be a source of concern for companies.

As a more light-weight alternative to using a language workbench, JavaScript Object Notation (JSON) can be used to define DSLs [19]. JSON is a data serialization format widely adopted to either store data physically or transfer it over the internet [20]. It is a semi-structured document format, that is possibly the most popular format for data exchange over the internet [21, 22]. It allows developers and IT professionals to transfer data structures across programming languages and environments, without having to worry about said environments. Instead, the data can be serialized or parsed in any language. JSON Schemas and JSON documents differ in their purpose. A JSON document contains the data to be sent or stored, organized in JSON objects. Meanwhile, JSON Schemas are used to define the structure of which a JSON document should adhere to, to ensure compatibility and consistency. The schema can then also be used to validate a JSON document [23].

JSON Schemas are the standard schema language for structuring JSON data. It is based on a combination of structural operators that describe values, arrays, and objects, with logical operators like negation, conjunction, and disjunction [24]. JSON Schema validators have been developed for many programming languages and they are used to make software and data transfer more reliable [20].

3 Methodology

This study employs an engineering research (a.k.a. design science) methodology, as it focuses on "*research that invents and evaluates technological artifacts*". [25] Firstly, we conducted research into the potential role of MDE in the deployment of ML models on edge devices and formulating relevant requirements. We applied these findings to develop an artifact to solve the specific problem of deploying ML models in cross-platform mobile environments, and evaluated it through a controlled experiment with developers from our industrial partner as well as analysing requirement satisfaction. The problem is being addressed through the three aforementioned research questions and three distinct cycles explained in Section 3.1.

3.1 Design Science Cycles

Following the design science process presented by Knauss [26], each cycle was centered around a specific phase in the design science process while iteratively advancing the understanding and progress of each research question. Below is a summary of the work conducted in each cycle of the study.

- **Cycle 1 (RQ1):** This cycle was about a major focus on research and understanding of the domain. We developed small proof-of-concept solutions and evaluated their viability in this context. We identified clear requirements for our artifact, both functional and non-functional, to deepen our knowledge about the domain and Wiretronic's needs. We employed tools such as interviews, frequent conversations with employees, and inspection of source code.
- **Cycle 2 (RQ2):** This cycle primarily focused on applying our findings through the concrete implementation of the artifact, iteratively verifying the development against the requirements specified by Wiretronic. Complemented with research into our domain and specific implementation details.
- **Cycle 3 (RQ3):** We evaluated our developed artifact by conducting a two-fold evaluation. Firstly, we conducted an internal evaluation, focusing on the requirements defined in cycle 1. Secondly, an external evaluation in collaboration with our supervisor from Chalmers and the respondents at Wiretronic. To test the suitability of our artifact we conducted a controlled experiment at Wiretronic, in which two groups performed a set of tasks using the existing approach and the new approach. The two groups were measured with respect to time, lines of code, and correctness. Additionally, a final interview to evaluate their experiences of implementing cross-platform specific code using our new approach compared to the old approach.

3.2 Cycle 1: Domain Understanding and Initial Artifact Definition

As stated, we spent most of the first cycle researching the domain and its specific representation at Wiretronic, using this information to help define our requirements and the scope of the study. This section covers the methods applied during this phase.

3.2.1 Repository Analysis and Exploratory Program Comprehension

We spent part of the first cycle examining an existing library developed at Wiretronic. This library powers all of Wiretronic's machine learning operations on edge devices, here limited to smartphones. This was primarily done as a program comprehension activity, as we required a thorough understanding of the domain and current state of development to have informed discussions with engineers at the company, identify constraints for future requirements elicitation, and find potential areas of enhancement.

The applications are written in Flutter, using Java for Android-specific functionality and Swift for iOS-specific functionality. This resulted in large parts of the program comprehension being conducted twice, as the machine learning code was implemented both in Java and Swift. This gave us two possibilities to understand most of the relevant code instead of one. Aside from serving as a tool to inform our requirements elicitation and development, analysis of the library was used as a tool to better understand the Flutter architecture and how it handles communication between the cross-platform and native layers.

3.2.2 Interviews

At Wiretronic, there were two engineers with experience in applications and libraries relevant to the study, therefore these two were chosen for qualitative interviews. The interviewees had experience both in developing the ML models and deploying them on devices, which allowed us to obtain a holistic view of their current processes with a small number of interviewees. The interviews, conducted as part of our problem exploration in the first cycle, aimed to document workflows and challenges to inform our subsequent requirements elicitation. While the interviews were conducted during the initial phase of the study, they were not conducted immediately. We deemed it necessary to first grasp the theoretical concepts relevant to the study, in addition to performing program comprehension. This was done to ensure we would go into the interviews well-informed and that they would serve their purpose.

Label	Role	Platforms	Experience
Interviewee A	Engineer/System Architect	ML + iOS	4 years
Interviewee B	Engineer/System Architect	ML + Android	4 years

Table 1 List of interviewees participating in initial interviews.

The interviews were conducted in a semi-structured format to obtain both quantitative data, such as tools currently in use, and deeper, qualitative insights into workflows and challenges. To allow for this structure, we used a set of pre-defined questions (available in the appendix) in combination with follow-up questions to elicit more detailed information. Each session began with a standardized introduction to maintain consistency across interviews, regardless of the participants' prior knowledge of the study. When crafting the interview questions, we deliberately included some questions where we expected to already know the answer. This measure was taken to ensure that basic needs, or *must-be* requirements, were not overlooked. These requirements are often taken for granted and go unnoticed if fulfilled, but failing to fulfill them can render the artifact unusable [27].

Upon conducting the interviews, we analyzed them as part of our requirements identification. Since the majority of the interview content was equivalent across the two interviews, they could be directly compared to identify challenges and pain points identified by both engineers. We also followed up the interviews with informal discussions, helping us draw conclusions and inform requirements when opinions or statements presented in the interviews were conflicting.

3.2.3 Requirements Engineering

We used the insights obtained from examining relevant repositories at Wiretronic and interviewing engineers when specifying our requirements. Analyzing repositories gave us a good overview of their existing systems and possible areas of improvement within our scope. Additionally, the interviews were valuable in providing context to our findings, ensuring that they align with the requirements and priorities of the company.

Based on the initial interviews, to move from requirements elicitation to requirements identification, we proceeded in two steps: First, we derived a set of user stories,

centered around a persona representing a developer at Wiretronic. Second, from the user stories, we derived more concrete functional and non-functional requirements. These two steps helped us bridge the requirements elicitation and requirements identification phases, consolidating the information we had obtained without getting caught up in implementation details.

A user story [28] is a concise, simple description of a feature written from the perspective of an end-user, often used in agile settings. We derived our set of user stories from the initial interviews described in Sect. 3.2.2, by examining our notes and recordings taken during the interviews with the aim to identify the intended key features from the perspective of a developer who uses the envisioned artifact. This process lead to a set of user stories that, finally, were validated with our industry partner

After defining a set of suitable user stories we began defining requirements, both functional and non-functional, rooted in the user stories. Naturally, this phase helped in setting up a set of more concrete, measurable goals for the project. Defining the requirements was an important tool in defining the scope of our study and creating a mutual understanding of priorities among ourselves and with the engineers at Wiretronic. Furthermore, the requirements were vital for the third and final cycle of the study, when performing validation and verification of the developed artifact.

3.3 Cycle 2: Artifact Design and Development

The purpose of the second cycle was two-fold: it first involved transforming the data collected in the first cycle to well-informed technology and design choices, and secondly, it involved designing and implementing the artifact. This section covers how we conducted this transformation, as well as how the design and implementation phase was conducted.

3.3.1 Design and Technology Choices

This section and the choices we made can be divided into two separate areas making up our artifact: the DSL and the accompanying library.

When designing the DSL, the primary guiding factor was the interviews with engineers, as no similar project had been conducted at Wiretronic before. The interviews, informed by the initial research activities, helped us narrow down what specific purpose we should aim to solve. The technologies chosen for the library are primarily rooted in practices already in place at Wiretronic to avoid obstacles in the handover of the artifact at the end of the study and to ensure compatibility with relevant applications. This information was elicited through the interviews and our analysis of existing repositories.

The possibility of using a JSON Schema to define the DSL was explored before the actual study began. Through the interviews and subsequent requirements engineering it was deemed a viable and preferable option during the first research cycle. We found that the main role of the DSL would be to describe an ML serving pipeline, and not write the actual implementation and logic, thus making a JSON Schema a fitting choice. After this decision was made, more focus was put into how to best describe the

model metadata and the pre- and postprocessing steps. This involved going through the existing models and comparing which aspects of the current ML serving pipelines that are shared, and which are unique for one or a set of specific models.

After identifying the required content of the DSL, the concrete syntax had to be established. Thanks to the lightweight nature of JSON Schemas in contrast to developing a programming language, we were able to iterate on the syntax quickly and try out several variations of the syntax in a single working session. Additionally, some of the choices of this process were made automatically due to limitations of the JSON specification, as highlighted in Section 4.3.2.

The library was designed in parallel with the DSL, ensuring both that any additions or changes made to the DSL would be feasible to implement in the library and that we could find a suitable place for them. When designing the functionality for preparing and running the actual ML serving pipeline, we made several choices based on our initial research and the interview feedback.

It was clear that, since a camera-based ML application can receive data as a stream of images, the overhead introduced by our library must be minimal. This meant that we wanted to avoid parsing the DSL instance each time, and also avoid conditional statements during execution, based on the parsed DSL instance. Thus, we implemented the pre- and postprocessing as a series of individually contained *steps*, all implementing an interface with the necessary method stubs. Thus, the pipeline lists consist of generic pre- and postprocessors, and not the concrete implementation, according to the dependency inversion principle [29].

This helped us separate the preparation and execution of the pipeline, as all steps had a method for setting it up with all the correct parameters and a separate method for executing it. While this was mainly done to eliminate any DSL-related logic during execution, it also helped when designing the functionality to implement custom pre- or postprocessing steps. By allowing the developer to implement an anonymous class implementing the interface directly in the consuming application, they can be confident that the step will be compatible with the pipeline, as long as the implementation is fault-free. In MDE, this functionality is considered part of the model engine, which is presented in more detail in Section 4.3.3

3.4 Cycle 3: Artifact Evaluation

We evaluated the developed artifact in a threefold way: First, through a comparison with the initial vision and requirements from our industry partner Wiretronic. Second, by a controlled experiment involving several of their engineers. To assess the artifact's impact under controlled conditions, two groups of engineers performed tasks with and without the artifact alternately. Third, through unit and integration testing.

We describe the first two steps, that is, the methodology and results of our requirements validation and user study, in a dedicated section (Sect. 5).

Our unit and integrating testing efforts for focused on those areas of our implementation that are important for supporting the findings of our paper, in particular, in the context of the user study and requirements validation. Our unit test suite includes a total of 20 test cases for the functionality related to loading, setting up and running the pipeline from an available DSL instance. We tested whether the loading and the

provided functionality (e.g., custom pre -and postprocessors) work as intended. Our manual testing worked by testing the deployed framework from the perspective of a potential user, on the same DSL instances that we used for our experiments. The final version of our software passes all test cases.

4 Results

This chapter presents the findings of our study, the subsequent artifact implementation, and the evaluation of the artifact. It lays out the requirements that guided the artifact implementation and evaluation, along with the reasoning behind each requirement.

4.1 Initial Problem Exploration

This section is dedicated to presenting our findings from the first cycle, focused on defining the artifact. This entails our repository analysis and interviews.

4.1.1 Interview Findings

We primarily obtained insights into existing development processes and potential enhancements through the engineer interviews. Interviewee B stated that a DSL and accompanying tools would help in the development and testing of ML serving pipelines, specifically for iOS. Stating that since he does not use MacOS, a requirement for building iOS applications in Swift, he can not currently develop for iOS. Instead, if making changes to an ML serving pipeline, he would have to write and test the changes in Java and then pass development to Interviewee A, who can implement the equivalent functionality for iOS in Swift. He mentioned that with a DSL he could instead define an ML serving pipeline using the DSL and then be confident that the iOS implementation will work, as long as the DSL instance is written correctly. Interviewee A independently pointed this out as well, underlining the fact that native development and related communication are obstacles in their current workflow. Furthermore, the two engineers agreed that an additional problem they would like to solve is having to publish a new version of the library when either making a change to an ML serving pipeline or implementing a new model.

When asked about the language design, Interviewee B stated that they would prefer writing the pipeline steps in a format they are familiar with and can get used to quickly rather than a completely custom DSL since there are only two platforms. They used the reasoning that if they were to learn a new language or platform, they could learn the other platform (in their case, iOS/Swift) instead of a new DSL.

The two interviewees presented slightly different approaches to implementing the DSL in an application. Interviewee B suggested that it could be a part of the build process, i.e. generating platform-specific code for the ML pipeline when compiling the application. Interviewee A, however, noted that he would prefer that the DSL be bundled with the application, loaded and parsed during runtime, and then used to configure the pipeline. This suggestion can be classified as a model interpretation-based approach, as it parses and executes a model during runtime [15]. This process requires

including all possible pipeline operations in the application bundle. He stated that the performance implications would be negligible, especially in comparison to loading an ML model from either the disk or over the network, which is already done in the applications. The suggestion by Interviewee B would satisfy their shared pain point of having to republish the library when making a change to an ML serving pipeline, but it would still require publishing a new version of the application. Interviewee A's suggestion would also remove this step, but it could prove less flexible if a developer needs to add currently non-existent functionality or functionality not general enough to be part of the library.

4.1.2 Impact on Artifact Development

Here we present some decisions made after conducting our initial studies and interviews. While Section 4.3 explores the design and implementation of our artifact in more detail, this section aims to provide relevant context for Section 4.2, which lays out the requirements guiding the artifact development.

When re-examining the problem after our research and interview study, we decided to opt for an approach based on model interpretation. This decision was primarily driven by two factors. Firstly, the interviews along with further discussions with engineers confirmed that the set of operations used for image transformation is limited and overlaps significantly across pipelines, confirming our previous findings from examining repositories. This highlighted that the configuration of arguments would benefit more from abstraction than the development of completely new functionality. Secondly, by opting for a model interpretation-based approach, the need to release a new library or application version upon making changes to the pipeline is removed, as previously highlighted. Instead, the pipeline can be updated dynamically, for example by fetching it from a remote server, thanks to the required functionality being bundled in configurable modules with the application.

While it seems suitable for this scenario, opting for a model interpretation-based approach may bring drawbacks. As highlighted previously, if a new ML model that requires custom preprocessing functions is introduced, this functionality will not be present in the library. In this case, the DSL and library either have to be extended to include this functionality, or we would need to include a way for a developer to reference one-off functions residing in the application in the DSL with suitable syntax. This does in turn introduce a problem of runtime safety. If we fetch a new DSL instance and this includes functionality not present in the application, the pipeline will not be configured correctly.

Upon discussions with engineers, we still deemed the model interpretation-based approach to be most suitable. If using code generation and avoiding runtime configuration, completely new functionality not supported by the DSL would still require substantial maintenance work and manual updating of either the library or applications consuming it.

As highlighted by our interview study, the DSL needs to be easy to learn and adopt compared to mastering a new platform. This, in combination with our specific context of defining a machine learning serving pipeline using pre-defined functionalities, we decided that employing a JSON Schema for the DSL would be an effective approach.

This choice seemed more suitable than opting for a more complex and advanced tool like Xtext since the primary goal is to describe pipeline steps and we do not require more detailed application logic within the DSL. Utilizing a JSON Schema offers several advantages: it simplifies the versioning of the DSL and allows for the validation of DSL instances against the schema. These validation abilities in turn provide syntax highlighting and integrated documentation within the developers' editors for increased usability and ease of adoption.

4.2 Requirements

This section will present the requirements identified through our requirements engineering process, presented in further detail in Section 4.2.2. This entails both the user stories, focused on creating a high-level view of the solutions provided by our artifact, along with our functional and non-functional requirements. The requirements are presented together with a short description aimed to provide further context and reasoning behind the requirement.

4.2.1 User Stories

We present our set of user stories, which was obtained from our initial interviews, following the process described in Sect. . User stories are features written from the perspective of a user, in our case a developer [30].

- **UC1:** As a developer, I want to be able to create and modify ML pipelines for multiple platforms without requiring platform-specific knowledge.
- **UC2:** As a developer, I seek to avoid writing equivalent, platform-specific code for multiple platforms when deploying ML models.
- **UC3:** As a developer, I want a configuration file in a format I recognize, like JSON, to quickly change ML model parameters for rapid experimentation to enhance efficiency.
- **UC4:** As a developer, I aim to dynamically adjust ML model configurations using the DSL at runtime, thus avoiding releasing new application or library versions for changes to the configuration.
- **UC5:** As a developer, I wish to use pre-built templates for common ML tasks, enabling me to concentrate on developing new and unique features for improving model performance.
- **UC6:** As a developer, I need a framework to easier identify potential failures in the ML pipeline, reducing manual debugging efforts.

4.2.2 Functional Requirements

We now present functional and non-functional requirements, both of which were derived from the user stories, following the process described in Sect. . The functional requirements specify the functions of the system, the features it is going to have, and how it handles data [31].

Pipeline Specification (DSL)

The ML serving pipeline refers to the set of processing steps required for an ML serving model. Each step in the process is a pipeline step, that performs a specific operation or transformation to data. As specified in **FR1.1**, this would include the pre- and postprocessing steps required before and after using the ML models. The preprocessing steps Wiretronic uses include cropping an image, rotating an image, changing image format, normalizing pixels, and initializing buffers for storing image data. The post-processing steps include tensor conversion, and extracting tensor data into other formats.

- **FR1.1:** The DSL should be able to specify which pre- and postprocessing steps are required for an ML model in a given context.
- **FR1.2:** The DSL should be able to be validated against a JSON Schema to ensure its correctness.

Given the need for a clear and flexible way to define these pipelines, we have chosen to use JSON Schemas for our DSL. JSON Schemas provides a structured yet lightweight approach to defining the syntax and validation rules for our DSL, ensuring compatibility and ease of use across different platforms.

Platform-Specific Model Interpretation (DSL + Architecture)

When specifying the steps in the DSL, the library should allow for model interpretation directly in Swift and Java. It ensures the application can be run across different platforms, in this case iOS and Android, by abstracting away the complexities of writing platform-specific code, while also allowing for changes to the ML serving pipeline on the fly.

- **FR2.1:** The DSL should enable model interpretation in Swift and Java, initiating an ML serving pipeline from existing native functionality based on the steps defined in an instance of the DSL.

Support Pre-Existing and Custom Operations (DSL)

A tool like this needs to be able to maintain the freedom of implementing specific operations if needed. Our tool already provides the existing operations mentioned in Section 4.2.2, however, these are still pre-defined operations Wiretronic uses for their ML models. When working with ML models the preprocessing steps can significantly impact the predictions of the models, hence making it an iterative process using different operations that could need these custom operations [32].

- **FR3.1:** The DSL should enable the developers to use local functions instead of those pre-defined in the DSL.

Support Dynamic Changes of the Pipeline (Architecture)

One of the advantages of implementing a DSL and library solution is that it enables dynamic changes during runtime. By having the ML serving pipeline set up dynamically through a configuration JSON file, we can change the model serving parameters

without Wiretronic having to release new versions of their library. Since all functionality already exists in the library, we can dynamically load new model parameters when changes happen to the configuration file, or initialize a new configuration file.

- **FR4.1:** Being able to switch between several configurations while the application is running, enabling A/B testing of pipelines.

4.2.3 Non-Functional Requirements

Non-functional requirements, or quality requirements, specify how well the system performs its functions. It is very important to address these alongside the functional requirements, as they play a crucial role in what we want to achieve with the requirements as stated in Section 4.2 [31].

Usability

Usability refers to how friendly the system is to users [33]. The artifact aims to ease the workflow of the developers, hence it needs to be intuitive and have a low learning curve.

- **NFR1.1:** The system should be easy to learn, allowing developers to use it with minimal training required.

Maintainability

Maintainability here refers to the ability to improve and understand software [33]. We collaborated with a particular industry partner, we deemed it essential to make the artifact easy to build further upon by the company after the completion of the project. By writing documentation about our solution, the company's developers should easily be able to understand our library and DSL to make changes or add new features.

- **NFR2.1:** The system should be easy to update, with clear documentation and guides.
- **NFR2.2:** It should facilitate the addition of new ML serving pipeline features without having to make substantial modifications to the existing code.

Performance

Performance defines how fast a software system or component responds to actions [33]. Performance may be a concern for some when using model interpretation [15]. Through our research and implementation, we aim to prove that using model interpretation should not negatively impact the application startup time when initiating the ML serving pipeline through the library and DSL.

- **NFR3.1:** The system should not add more than 50ms to the application startup time when initiating an ML serving pipeline from an instance of the DSL.
- **NFR3.2:** The system should not cause performance overheads when running an application containing an ML serving pipeline dynamically set up by the library.

Compatibility

Compatibility refers to a system that exists and interacts with another system in the same environment [33]. As the system is in a cross-platform environment it is important to not have any limitations due to different operating systems or IDEs.

- **NFR4.1:** The system should work across multiple platforms (MacOS, Windows, and Linux).
- **NFR4.2:** The system should work in Flutter codebases.

4.3 Design and Implementation

4.3.1 Current Approach

Figure 1 is a code snippet from the existing library at Wiretronic, it displays how the preprocessing is written in Java for one of their models. The method performs cropping, rotation, and normalization of an image, with the parameters for image size being instance variables in the Java class. When implementing a new ML model or making changes to an existing ML serving pipeline, the developers will also have to write this code for Swift to support iOS devices. As will be presented in this section, our DSL abstracts away the platform-specific details and provides the developer with a single interface to specify the ML serving pipeline.

```
IValue preprocess(BitmapWrapper image) {
    BitmapWrapper squareImage = image.squareCrop(offset);
    BitmapWrapper postProcessed = squareImage.transform(inferenceShape.getWidth(), inferenceShape.getHeight(), 90);

    final FloatBuffer floatBuffer = Tensor
        .allocateFloatBuffer(3 * postProcessed.bitmap.getWidth() * postProcessed.bitmap.getHeight());

    final int pixelsCount = postProcessed.bitmap.getWidth() * postProcessed.bitmap.getHeight();
    postProcessed.bitmap.getPixels(buf, 0, postProcessed.bitmap.getWidth(), 0, 0, postProcessed.bitmap.getWidth(),
        postProcessed.bitmap.getHeight());

    // Torch uses CHW image format
    final int offset_g = pixelsCount;
    final int offset_b = 2 * pixelsCount;

    for (int i = 0; i < pixelsCount; i++) {
        final int c = buf[i];

        float r = ((c >> 16) & 0xff) / 255.0f;
        float g = ((c >> 8) & 0xff) / 255.0f;
        float b = ((c) & 0xff) / 255.0f;

        // imagenet normalization
        r = (r - 0.485f) / 0.229f;
        g = (g - 0.456f) / 0.224f;
        b = (b - 0.406f) / 0.225f;

        floatBuffer.put(i, r);
        floatBuffer.put(i + offset_g, g);
        floatBuffer.put(i + offset_b, b);
    }

    IValue castedInputData = IValue.from(Tensor.fromBlob(floatBuffer,
        new long[]{1, 3, inferenceShape.getWidth(), inferenceShape.getHeight()}));
    return castedInputData;
}
```

Fig. 1 The preprocessing method in Java that Wiretronic uses for one of their models.

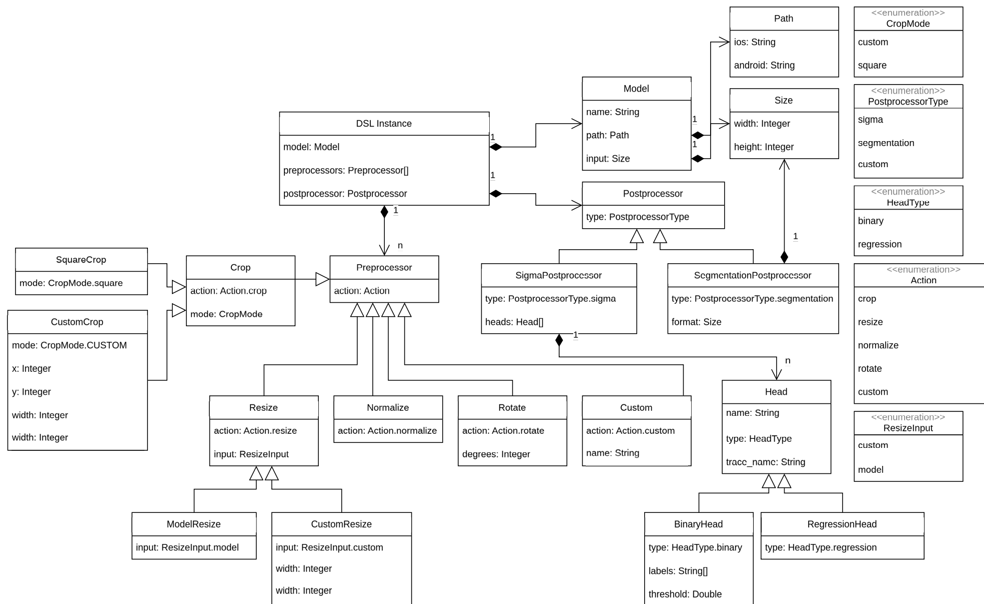

Fig. 2 Abstract syntax of the DSL.

4.3.2 Proposed Approach

In this section, we propose an alternative approach to manage the ML serving pipelines in cross-platform mobile environments, decoupling this configuration from the underlying platform. This proposal is the result of the previously outlined requirements definition and the work done to inform that. It consists of two separate but connected parts: the DSL aimed to aid developers in specifying the ML serving pipelines in a single, familiar format, and the Flutter library which supports the DSL and generates the pipelines at runtime.

Domain-Specific Language

The DSL provides definitions for three different aspects of the pipeline: the model metadata, preprocessing, and postprocessing. Figure 2 displays the abstract syntax of the language through a metamodel, showing the main concepts of the domain and their relationships. We now present the concepts from our meta-model and illustrate their specification in our concrete syntax by example.

Using the DSL, a developer can provide metadata about the model, consisting of its name, the specific path of where to fetch the model from on iOS and Android respectively, and the required input size of the model, which any image fed to the pipeline can be resized to. How this metadata can be defined is displayed in Listing 1.

Preprocessing is divided into separate steps, called preprocessors. Each preprocessor supports one specific action and can receive arguments from the developer as necessary. The DSL provides built-in support for four preprocessors: cropping, resizing,

```
"model": {
    "name": "Multihead",
    "path": {
        "android": "multihead.pt",
        "ios": "multihead.mlmodel"
    },
    "input": {
        "width": 380,
        "height": 380
    }
}
```

Listing 1 An example of how the DSL allows for specifying metadata about the model.

```
"preprocessors": [
    {
        "action": "crop",
        "mode": "square"
    },
    {
        "action": "resize",
        "input": "custom",
        "height": 380,
        "width": 380
    },
    {
        "action": "normalize"
    }
]
```

Listing 2 An example preprocessing configuration using the DSL.

rotating, and normalizing an image. These steps are commonly used when preprocessing images for ML tasks, as the image received from e.g. the camera can be of different dimensions and orientation depending on the device configuration.

The order of preprocessing steps is important. If, for example, an image received from the camera is 2000x2000 pixels after cropping, but the model requires an image with normalized colors of size 300x300, it would be a waste of time and computing power to apply the normalization before resizing the image, as it would require iterating through over 40 times as many pixels. Since the JSON specification does not guarantee a maintained order of object entries, the preprocessors have to be defined in an array of objects and not an object with a key for each preprocessor [34]. To accommodate this, each preprocessor is defined as an object with a key called *action* specifying the name of the step. The additional argument entries that are required for the preprocessor are then inferred by the schema through the value of the *action* key. The built-in preprocessors are defined below, and an example preprocessor configuration is displayed in Listing 2.

```
1  "postprocessor": {
2      "type": "segmentation",
3      "format": {
4          "height": 320,
5          "width": 320
6      }
7  }
```

Listing 3 Example of the postprocessing in Wiretronic's Segmentation model using our DSL.

```
1  "postprocessor": {
2      "type": "multihead",
3      "heads": [
4          {
5              "name": "is_visible",
6              "type": "binary",
7              "threshold": 0.3
8          },
9          {
10             "name": "centerpoint_x",
11             "type": "regression"
12         },
13         {
14             "name": "centerpoint_y",
15             "type": "regression"
16         }
```

Listing 4 Example of the postprocessing in Wiretronic's multi-headed model using our DSL, showcases 4 out of 11 output heads.

- **crop:** Allows the developer to specify a *mode*. If *mode* is *square*, it will perform a square crop in the center of the image. If mode is *custom*, the DSL requires the additional arguments x, y, *width*, and *height*, specified as integers.
- **resize:** Resizes the input image. The developer can choose the *input* for the measurements, if it is *custom* the image will be resized according to the arguments specified by the developer for *width* and *height*. If it is *model*, the function will use the size specified in the model metadata.
- **rotate:** Rotates an image by the number of degrees specified in the argument *degrees*.
- **normalize:** Takes no additional arguments. Normalizes the image.

While we found the preprocessing steps to be generalizable and had a large overlap in usage across models, the postprocessing was close to the opposite. Here, instead of implementing support for specific functions that can be used for many different models, we had to implement model-centric solutions.

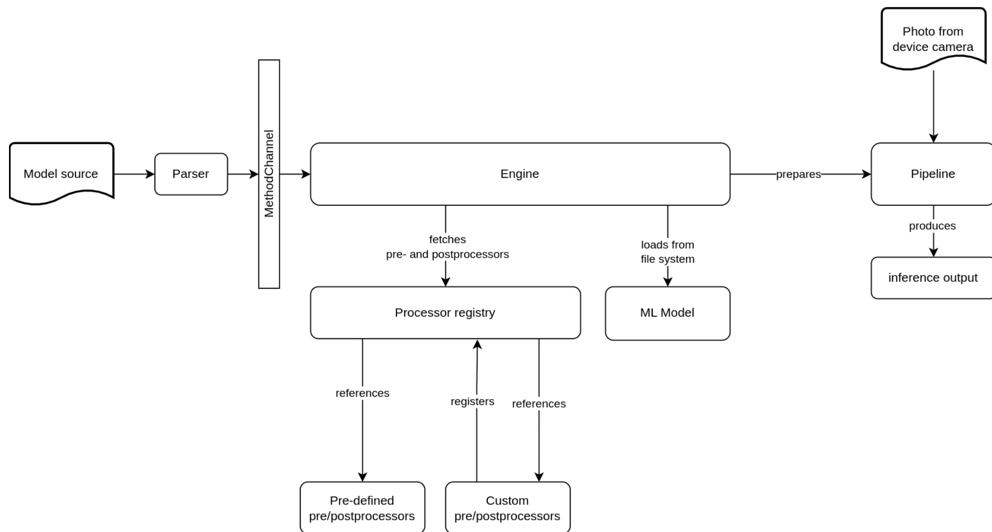

Fig. 3 Illustration of how the model engine prepares an ML serving pipeline from a DSL instance.

Listing 4 displays the postprocessing of Wiretronic's multi-headed model. This figure illustrates why model output can pose a challenge when defining postprocessing of these outputs using our DSL. This model outputs 11 "heads", specific to this model. Comparing this to Listing 3, we see how different the two models' outputs and postprocessing can be. During our research, we implemented functionality for these models as proof-of-concepts, displaying that the DSL can be utilized for both simple and advanced postprocessing tasks. However, if Wiretronic were to implement a completely new model, they would need to implement this in the DSL.

While extending the DSL can be suitable when introducing a new model with completely new postprocessing, there may be one-off situations where a model requires some custom operations in either the pre- or postprocessing stages. To accommodate this, we implemented a pre- and postprocessor registry, which allows developers to introduce custom functionality without the DSL being an obstacle. Contrary to Wiretronic's current approach, where everything ML-related is handled in a library, our DSL and library would allow defining custom functionality directly in the application where it's required. If developers then encounter the same situation in more applications, they can decide to introduce the custom step into the DSL and library permanently. The main difference between a custom implementation and an existing one is that it would require a re-release of the application since it involves writing platform-specific code that needs to be bundled with the application.

4.3.3 Model Engine

As required when opting for a model interpretation-based approach, a model engine was implemented to handle the model-to-code transformation. The workflow for using this model engine is shown in Figure 3. When starting the application, the developer

can initialize the model engine in Flutter by providing a path to the correct DSL instance. This DSL instance is loaded and parsed, creating a nested dictionary referred to as the model instance. Performing the parsing in Flutter helps avoid discrepancies in parsing or file system access between platforms. After this, the model instance is fed through a MethodChannel into the platform-specific model engines. The model engine uses the model instance to fetch the correct pre- and post-processing steps for the ML model from the processor registry. Additionally, it also uses the path provided in the model instance to load the correct ML model from the file system. Upon fetching the pre- and postprocessing steps and loading the ML model, the ML serving pipeline is ready and can receive images from the device's camera. Since the model interpretation happens at startup, any performance overheads incurred will be present on application startup and not when performing inference.

4.3.4 DSL Development Tools

```typescript
import { Type } from "@sinclair/typebox";

const street = Type.Object({
  name: Type.String(),
  number: Type.Number({ minimum: 0 }),
});

const apartment = Type.Object({
  floor: Type.Integer({ minimum: 0 }),
  street,
});

const house = Type.Object({
  street,
});

const schema = Type.Object({
  housing: Type.Union([apartment, house]),
});
```

```json
{
  "type": "object",
  "properties": {
    "housing": {
      "anyOf": [
        {
          "type": "object",
          "properties": {
            "floor": { "minimum": 0, "type": "integer" },
            "street": {
              "type": "object",
              "properties": {
                "name": { "type": "string" },
                "number": { "minimum": 0, "type": "number" }
              },
              "required": ["name", "number"]
            }
          },
          "required": ["floor", "street"]
        },
        {
          "type": "object",
          "properties": {
            "street": {
              "type": "object",
              "properties": {
                "name": { "type": "string" },
                "number": { "minimum": 0, "type": "number" }
              },
              "required": ["name", "number"]
            }
          },
          "required": ["street"]
        }
      ]
    }
  },
  "required": ["housing"]
}
```

Fig. 4 Comparison of an example JSON schema as defined using Typebox (left) and the actual schema outputted by Typebox (right).

We used the TypeScript tool TypeBox to develop the JSON Schema and abstract syntax that defines our DSL. TypeBox significantly reduces the amount of code having to be written compared to defining a JSON Schema manually. Additionally, it improved developer ergonomics by providing functions for set theory, allowing us to easily define complex conditional types. After the JSON Schema had been defined using TypeBox, we ran a TypeScript script that outputs the rendered JSON Schema to a JSON file. Figure 4 displays how TypeBox allows for separation and significantly reduced code when defining a JSON Schema, using a mock example.

5 Evaluation

This section covers our evaluation of the developed artifact. Here, we first conducted an internal evaluation of the developed artifact, comparing the result with the visions presented by Wiretronic and the set of requirements we developed as a result of our initial exploration. Secondly, we evaluated the artifact together with Wiretronic, performing a controlled experiment with two groups of engineers. In doing this evaluation, we covered both aspects of verification and validation, ensuring not only that the artifact has been built correctly, but also that it solves the correct problem.

5.1 Controlled Experiment Set-up

To perform the evaluation, we performed a controlled experiment. The purpose behind this was two-fold: first, to identify the specific impact of our artifact, and second, to maintain increased control over the experiment helped ensure similar conditions for each trial, minimizing the impact of outside factors. The experiment was carried out using a Latin square design [35], in which two groups each performed two tasks. One group used our artifact to solve the first task and not for the second task, and vice versa, as is displayed in Table 2. The Latin square design was chosen for its ability to control for two potential sources of variability: the order of tasks and the individual differences among participants. This design ensures that each participant experiences each condition (using the artifact and not using the artifact) in a different order, which helps to minimize learning effects and other biases that could influence the results.

Because of the small available sample size, we utilized stratified sampling [36]. The engineers were categorized into two groups for the Latin square design (explained above): experienced and inexperienced, with both experienced engineers having four years of experience and inexperienced zero, not having worked in the environment at all. In total, we formed the two experiment groups with an equal number of experienced and inexperienced engineers, as visible in Table 3.

All sessions were performed in a 60-minute time slot in which all participants had the same time to perform the tasks. Furthermore, the two groups received identical documentation for our artifact. The provided documentation consisted of a concise description of the DSL elements with their attributes. The documentation as well as the experimental tasks are available as part of our online appendix [7].

With this experiment, we aimed to identify whether the introduction of our artifact improves the workflow of the specific process it is designed to improve, to give answers to **RQ3**. Opting for a contrived setting allowed us to identify the impact of

	Group 1	**Group 2**
Task 1	not using artifact	using artifact
Task 2	using artifact	not using artifact

Table 2 Experiment setup.

Group 1	**Engineer A****	Experienced
	Engineer B	Inexperienced
Group 2	**Engineer C***	Experienced
	Engineer D	Inexperienced

Table 3 The engineers from Wiretronic that participated in the experiment, with their respective experience levels. *Interviewee A, **Interviewee B.

the artifact, albeit at the cost of generalizability and realism [37]. To provide further nuance and compensate for the drawbacks of a controlled experiment, we conducted semi-constructed interviews with the participants to gain qualitative insights.

Metrics

In RQ3 we wanted to answer to what extent our artifact improves the developer experience in aspects such as maintenance, feature development, time-saving, and resource planning. We used the experiment to obtain quantitative data and combined this with the interviews for qualitative data. The quantifiable metrics we observed through the experiment were the following:

- Time per completed task. Measured in minutes, extracted from commit timestamps.
- Lines of code written to complete each task. Measured in lines inserted and lines deleted for each commit.
- Correctness, a binary metric of whether the task was performed correctly or incorrectly. Measured by manual static analysis of the solution, and occurrence of runtime errors after the experiment.

The post-experiment interviews helped us obtain qualitative data about more subjective metrics, identifying how usable, intuitive, and useful the artifact can be for the engineers' daily workflow. The questions asked in these interviews are available in the appendix.

Additionally, we performed hypothesis testing on our metrics, specifically time and correctness to get a more comprehensive view of our results. We expected the data to not be normally distributed due to a small sample size, natural variations in human performance, and the variability in the experience levels of the engineers. For the development time, we utilized the Mann-Whitney U Test [38]. This test is suitable because it is non-parametric and does not assume a normal distribution, making it appropriate for small sample sizes [39]. For the correctness metric, we used McNemar test in its exact variant [40]. It is designed for paired, binary data, in our case the cases *correct* and *incorrect*, and is ideal for small sample sizes [41].

The hypotheses for the Mann-Whitney U test:

- Null Hypothesis (H0): There is no statistically significant difference in the development time between the old and new approaches.
- Alternative Hypothesis (H1): There is a statistically significant difference in the development time between the old and new approaches.

The hypotheses for Fisher's Exact Test:

- Null Hypothesis (H0): There is no statistically significant difference in correctness between the old and new approaches.
- Alternative Hypothesis (H1): There is a statistically significant difference in correctness between the old and new approaches.

Tasks

As stated, we designed two example tasks to evaluate the artifact. Task 1 had three subtasks and Task 2 had two subtasks. These were designed with the pain points of Wiretronic in mind, identifying how effective the artifact can be in maintenance for both the pre- and postprocessing parts of an ML serving pipeline. Therefore, Task 1 is completely related to preprocessing and Task 2 is completely related to postprocessing.

Task 1 - Assessing preprocessing: Given an existing model with accompanying pre- and postprocessing methods implemented, the engineers will perform the following subtasks:

- Change the path from which the model is loaded.
- Modify the size of the input data, that the image will be resized to from 300 by 300 to a new specified dimension, 380 by 380.
- Enable normalization for the input image.

Task 2 - Assessing postprocessing: The model that has the least trivial postprocessing is a multi-headed model used for several computer vision tasks. Being multi-headed, it can both provide e.g. whether an item is visible in the frame, and produce a bounding box for locating the item.

- Adjust the threshold of the binary classification head named *is_visible* to 0.5.
- Implement interpolation for the binary classification called *size*. Set the size to 300 if below the threshold, otherwise set it to 500.

5.2 Results

This section will go through the findings from our different evaluations of our artifact. This involves examining whether it fulfills the requirements set out at the beginning of the study along with the experiment and accompanying interviews from the third cycle.

5.2.1 Experiment Results

The results from the experiment conducted as part of our evaluation are presented here. They are presented group-wise, presenting the results from Group 1 and Group 2 for each metric. Each metric is reported per subtask.

	New approach			Old approach	
	1.1	1.2	1.3	2.1	2.2
Engineer A	1	1	1	3	4
Engineer B	2	1	2	9	21

Table 4 The time (in minutes) it took for engineers A & B to complete the subtasks in the first task using the new approach, and the subtasks in the second task using the old approach.

	Old approach			New approach	
	1.1	1.2	1.3	2.1	2.2
Engineer C	2	5	4	1	2
Engineer D	7	5	15	2	6

Table 5 The time (in minutes) it took for engineers C & D to complete the subtasks in the first task using the new approach, and the subtasks in the second task using the old approach.

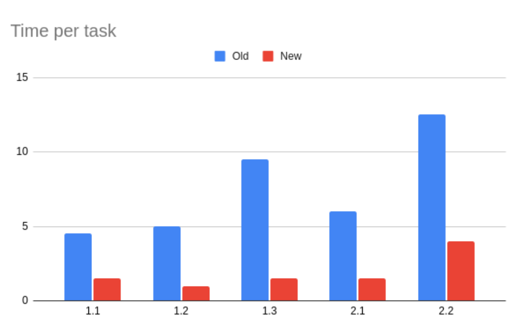

Fig. 5 The mean time per task (in minutes) for the old and new approach, respectively.

Development Time

Overall, the artifact generated a substantial improvement in development time for all subtasks. As displayed in Table 4 and 5, this was true for both the experienced engineers (A, C) and the inexperienced engineers (B, D). Figure 5 displays the mean time for all participants, categorized by task and approach used. When comparing an inexperienced engineer not using the artifact and one using the artifact, the average improvement in development time was 344%. If comparing experienced to inexperienced engineers before introducing the artifact, the experienced engineers on average performed 141% better than the inexperienced engineers.

Lines of Code

As displayed in Table 6 and 7, there were no significant differences in the absolute number of lines of code required to complete the tasks. Only in task 2.2 was there a major difference but this is attributed to normalization being a built-in function in the DSL, thus only requiring it to be enabled instead of having to perform the normalization manually. This experiment only included development on Android, however, and

		New approach			Old approach	
		1.1	1.2	1.3	2.1	2.2
Engineer A	Insertions	2	2	3	1	3
	Deletions	2	2	0	1	3
Engineer B	Insertions	2	2	4	1	6
	Deletions	2	2	0	1	4

Table 6 The lines of code written by engineers A & B to complete the subtasks in the first task using the new approach, and the subtasks in the second task using the old approach.

		Old approach			New approach	
		1.1	1.2	1.3	2.1	2.2
Engineer C	Insertions	2	1	4	1	1
	Deletions	2	1	0	1	0
Engineer D	Insertions	1	1	3	2	1
	Deletions	1	1	0	2	0

Table 7 The lines of code written by engineers C & D to complete the subtasks in the first task using the old approach, and the subtasks in the second task using the new approach.

	New approach			Old approach	
	1.1	1.2	1.3	2.1	2.2
Engineer A	correct	correct	correct	correct	incorrect
Engineer B	correct	correct	incorrect	correct	correct

Table 8 The correctness for engineers A & B when completing the subtasks in the first task using the new approach, and the subtasks in the second task using the old approach.

some of the tasks would require performing equivalent operations also on iOS, thus increasing the required lines of code when not using the DSL.

Correctness

When measuring correctness we manually tested each commit to catch any runtime failures, and statically analyzed the commits, ensuring that the commits using our artifact did not include any unnecessary code not required for the task description. As can be observed in Table 8 and 9, six subtasks implemented using the old approach were considered correct, accounting for 60%. Meanwhile, for the new approach, eight solutions were deemed correct, representing 80%.

5.2.2 Hypothesis Testing

We performed hypothesis testing on the development time and correctness metrics, as explained in Section 5.1, Hypothesis testing yields a p-value between 0 and 1, which is compared to a predefined threshold α. If p is smaller than α, the null hypothesis can be rejected, that is, the difference between two samples (in our case, the two treatments of using our solution or not) is deemed statistically significant. Following the widely used default in scientific studies [42], we used the threshold value $\alpha = 0.05$.

	Old approach			New approach	
	1.1	1.2	1.3	2.1	2.2
Engineer C	correct	correct	wrong	correct	correct
Engineer D	wrong	correct	wrong	wrong	correct

Table 9 The correctness for engineers C & D when completing the subtasks in the first task using the old approach, and the subtasks in the second task using the new approach.

For our development time comparison, a Mann-Whitney U test was utilized. We obtained a p-value of 0.002 (Mann-Whitney U statistic: 90.0), which is clearly below the significance threshold. Therefore, the results of the Mann-Whitney U test indicate a statistically significant improvement in time efficiency with the new approach. This finding supports discarding the null hypothesis, hence the new approach would reduce the time required to complete tasks.

For the correctness comparison, a McNemar Exact test was used. We obtained a p-value of 0.5 (test statistic: 0.0). Hence, the null hypothesis cannot be rejected, and the difference between the new and old approaches cannot be deemed statistically significant. This result is likely due to the small sample size. In conclusion, while, unlike for the case of development time, we do not observe a statistical significant benefit of our approach with regard to correctness, we do observe that our approach does not compromise the ability of developers to produce correct solutions.

5.2.3 Requirements

Here, we evaluate the artifact with respect to the requirements defined in the first cycle. This is split into functional and non-functional requirements, and evaluated both using metrics from testing the software and subjective opinions presented by engineers during the evaluative interviews held after the experiment.

Functional Requirements

Pipeline Specification: The DSL does enable developers to specify which pre- and postprocessing steps are required for an ML model. The DSL does validate against a JSON Schema when using an IDE, both in terms of what is required for an ML model in general, and autocomplete with all pre-existing operations.

Platform-Specific Model Interpretation: The DSL does enable model interpretation in Swift and Java using the model engine illustrated in Figure 3, and further explained in Section 4.3.3.

Support Pre-existing and Custom Operations: Since the DSL was implemented based on the results of our interviews and repository studies, we were able to identify and implement support for the most commonly used operations, both in pre- and postprocessing. We complemented this with the previously mentioned pre- and postprocessor registry, which allows developers to include custom functionality, thus fulfilling the requirement of supporting both pre-existing and custom operations.

Support Dynamic Swapping of Configuration: The ML serving pipeline is set up through the runtime parsing of a JSON file. Thus, developers can write code that supports changing which JSON file is loaded, and the library would instantiate a

new pipeline. This is possible thanks to the model interpretation approach, performing the model-to-code transformation at runtime.

5.2.4 Non-functional Requirements

Usability and learnability: The goal was to make the DSL easy to learn, allowing developers to use it with a minimal training required. During the second round of interviews, the participants were asked to rate the DSL in terms of the properties of intuitiveness, learnability, and usability on a Likert scale from 1 to 5. Intuitiveness was scored with an average of 4.75, learnability was scored with average of 4.75, and usability was scored with an average of 5. These answers indicate accomplishing our goal. Additionally, one of the inexperienced participants stated the tool provides a lower barrier of entry for contributing to the code: "*I would not dare to work in this environment otherwise, using the new method makes me feel more secure*" - Engineer D. However, we did get feedback on the documentation being slightly confusing, with both Engineer C & D stating that we should improve the documentation and that the large amount of text in a single place made it difficult to get an overview. Based on this feedback, we made improvements to the documentation after the interviews.

In line with the high average score for learnability, in their qualitative feedback, participants expressed that they found the DSL to be highly learnable, with several noting how quickly they adapted to its use. Engineer D mentioned, "*I got a feeling for how the processing steps should be formatted quite quickly, only having to look at the readme for the relevant parameters when implementing a new step.*" This sentiment was echoed by Engineer B: "*I thought the questions were harder to grasp than the DSL itself.*" Engineer C highlighted the immediate understanding of the DSL, stating, "*In this case: immediately,*" while Engineer A described the learning process as "*Very fast, I would say.*" These comments underscore the DSL's intuitive design and the effectiveness of the provided documentation in facilitating rapid learning and adoption.

To conclude, given the diverse background of our participants and the short training time to use the DSL for the provided task, we believe that our results shed a promising light on the usability and learnability of the DSL, assuming a developer who is familiar with the domain of pre- and postprocessing data in the context of machine learning, and with JSON-based formats.

Maintainability: The main feature of the new approach is enabling easier updates of ML serving pipelines: "I think it was much better compared to without, there are so many files I don't recognize and difficult navigating the file structure" - Engineer B. The DSL enables the developers to modify the models through only one configuration file, not having to make substantial changes to the existing code.

Performance: We conducted a test measuring how long the application takes from startup to readiness. This test was performed on a typical instance of our DSL selected as a representative example for practical usage contexts of our DSL, based on communication with our industry partner. Specifically, the DSL instance included three preprocessing steps of square crop, resizing and normalization, parameter specifications for a multi-headed model, and nine built-in post-processing instructions for the multi-headed model. We started the application ten times using each approach. From the measurements, we found that our approach increases the startup time by an

	New approach	Old approach
	1314	1381
	1323	1356
	1393	1278
	1427	1348
	1404	1340
	1392	1350
	1410	1383
	1387	1372
	1402	1374
	1371	1399
average	**1382.3**	**1358.1**
median	**1392.5**	**1364**

Table 10 The startup time (in milliseconds) of the application for the new and old approach respectively.

average of 24ms. This fulfills requirement **NFR3.1**, stating that our approach should not add more than 50ms to the startup time of an application consuming the library. The test was conducted on a single computer and OS, therefore the results might differ. The full results of the trial runs are displayed in Table 10.

Compatibility: We manually tested the new approach across Windows, Linux, and MacOS. As long as the system had installed all the necessary software like Android Studio and Flutter, there were no issues in either system during build or runtime.

6 Discussion

This chapter discusses different ways to support several ideas provided by the interviewees during evaluation in both the first cycle and the second cycle.

6.1 Research Question 1

> **RQ1:** How can we design a domain-specific language (DSL) for an ML model with its required inputs, outputs, and pre- and post-processing stages?

It is important to distinguish that our research is directed at making a DSL describing the input, output, preprocessing, and postprocessing around ML models, not the models themselves.

Through our iterative process working on the project, how an ML model can be described through a DSL depends on how generalizable you want it to be. During our research into the domain, we have found that it is quite easy to describe what happens before the data is fed into the ML model, however, the difficult part is describing what happens after. Finding a balance here was one of the more challenging tasks of the study. In the end, the choice to implement the DSL using a JSON Schema with an accompanying library allowed us to provide both a simple interface to describe ML serving pipelines and a way to implement custom, one-off features without slowing down development.

Our approach allows developers to specify metadata about the model, consisting of its name, path on the device, and input shape. The preprocessing is described as a series of steps, where we implemented support for the most common actions used in Wiretronic's current ML serving pipelines in addition to the possibility of defining custom steps. Lastly, the DSL supports specifying the required postprocessing actions. While the preprocessing actions were found to be quite trivial and generalizable, the postprocessing steps are usually different between each model, opting for having to implement custom functionality to handle the model outputs. Here, the functionality to easily be able to define custom postprocessing actions is necessary.

Summary

We represent an ML model's input, output, preprocessing, and postprocessing steps a language in a DSL, using an underlying meta-model for capturing the abstract syntax, and utilizing JSON Schemas to offer concrete syntax tailored to industrial requirements.

6.2 Research Question 2

RQ2: How can we best implement and utilize the DSL in a concrete setting, specifically in the development of cross-platform mobile applications?

One aim with the DSL was to create a unified interface for not only multiple platforms but also for engineers of different backgrounds. This meant that we did not want to make it overly related to the underlying platforms, since this could cause confusion or unfamiliarity for ML-focused engineers. Furthermore, we did not want to make the DSL too restricting, offering experienced engineers the possibility to combine the DSL with custom, platform-specific functionality.

While the DSL is aimed at cross-platform mobile development, we did not want to make it tied to the technique currently used at Wiretronic, for example as an internal DSL written in Dart (for Flutter). This connects to the previously mentioned point of creating a unified interface across platforms and experience levels, but it also allows for porting or extending the DSL to additional platforms. We developed the DSL and accompanying library so that if Wiretronic decides to shift its cross-platform development to another technique, the DSL would not require any modifications.

To accommodate the initial requests made by engineers to not require a complete re-release of the application upon changes to the ML serving pipeline, we implemented the DSL using a model interpretation approach, instead of using code generation. Any application consuming our accompanying library could fetch a remote file written in our DSL and dynamically set up the ML serving pipeline, without having to re-release the application.

Summary

The DSL and accompanying library were implemented to support different underlying techniques, engineers of different backgrounds, and making changes to the ML serving pipeline without re-releasing the application.

6.3 Research Question 3

RQ3: To what extent does the introduction of a DSL and an accompanying library improve the developer experience in the aspects of maintenance, feature development, time-saving, and resource planning?

From our controlled experiment and two rounds of interviews, it has been made clear that a DSL designed in a familiar format can aid in lowering the barrier of entry in this area of development. This may be the largest improvement when comparing the previously used approach, as new engineers can contribute and experiment in development. Through both objective and subjective metrics, our evaluation showed that the engineers worked faster and more confidently while using our approach, partly thanks to the ML-related functionality being isolated into a single file and format. In addition, the results in Section 5.2.1 show us improvement in all metrics using our new approach. The average improvement in development time was 344%, which is also backed by the hypothesis testing performed in Section 5.2.2. The correctness improved by 20% in the controlled experiment, but we could not prove statistical significance. The engineers did however state in the interviews following the experiment they still felt more secure using the new approach. If the DSL can help more engineers contribute to this area of development it can help in all the aspects stated in this research question. More engineers will be able to perform maintenance tasks and develop new features, further helping Wiretronic deliver features faster and easing their planning.

Summary

The DSL does lower the complexity of edge-deployed ML at Wiretronic. The DSL and accompanying library make the entry into the field quicker, while also enabling the engineers to do the work faster and more confidently. Hence, improving the developer experience in the aspects outlined in the research question.

6.4 Cross-Platform Communication

Due to the nature of cross-platform development, there are many instances of communication between the Flutter layer and the native layer through MethodChannels. During the development of our library, we ran into many instances of having to debug on both sides of the MethodChannels. This can become a very tedious and time-consuming task. With our library solving the issues of layer communication, we can effectively ease the need for debugging for the developers.

In the future, we may see a shift away from using channels and data serialization for inter-layer communication. React Native explores this in their new architecture, which is under development at the time of writing. Here, the native code is written in C++ and the cross-platform layer (in this case, JavaScript) can hold references to C++ objects and vice-versa, calling functions directly on these objects [13].

6.5 Threats to Validity

We discuss threats to validity in the three main directions of internal, external, and construct validity.

Internal Validity

Internal validity is of concern when we examine causal relations [43]. We aimed to ensure internal validity in our study by using a controlled setting, with the intent of eliminating any confounding factors. The elicitation of our requirements and scope was defined through only two interviews and casual conversations, each conducted with the one mainly responsible for the Android part, and the other responsible for the iOS part.

We evaluated our DSL and library with the developers at Wiretronic as the problem and research area were brought to light by them. This is a small group, and this may cause problems with internal validity. If creating random groups for the experiment, there is a chance that uneven levels of previous experiences in the area become a confounding factor. As a mitigation to this issue, we used stratified sampling and a cross-over design, in which participants from both groups are exposed to both treatments, thereby turning each participant into their own control. As part of the evaluation we also used manual static testing for the correctness, which may introduce human error or bias. To mitigate this we also ran the code to see if the solutions would introduce any runtime errors.

The two experienced participants in the experiment were also the interviewees in the initial interviews, which might introduce bias or influence their performance. This overlap could affect the way they approached the tasks. We are aware of this as a potential threat but tried to minimize its impact through clear communication.

Another factor that may affect internal validity is the possibility that participants may not be honest in their feedback. One of us has previously been employed at Wiretronic, so the developers might provide positive feedback to help us in our work. To mitigate this issue, we will clearly communicate that the respondents should give answers and prefer tasks as they naturally would. Additionally, having been colleagues in the past, we know the employees well enough that they will be comfortable giving honest critiques.

During the experiment, one participant performed worse than the others, which can be considered an outlier. We ran the statistical analysis again without the data from this engineer, but we still observed statistical significance. Mitigating the impact on our results.

External Validity

External validity is to what extent it is possible to generalize the findings [43]. Our DSL was designed and implemented only the single ecosystem of Wiretronic, and our evaluation involved a significant subsect of their engineers as participants. This population comprises of proficient software engineers with several years of experience. Thus, the homogeneity and small size of our experiment may not be generalizable. In defense of external validity, while our study was tailored to the needs of Wiretronic, the underlying principles and methodologies we employed are not inherently limited to this specific context We outline a set of assumptions for other companies and application contexts where our findings could be relevant: First, the company develops applications for several mobile platforms, such as iOS and Android. Second, the company needs to deploy pre-trained machine learning models on edge devices, particularly smart phones. Third, the company requires the ability to update ML serving pipelines dynamically at runtime without re-releasing the application. Fourth, the company seeks to improve collaboration and workflow efficiency between ML engineers and application developers. Fifth, the company is open to using model-driven engineering techniques, including a DSL for specifying ML serving pipelines. Companies meeting these assumptions might directly benefit from the findings of our study.

Furthermore, as our scope limits us to deploying ML models on edge devices, this study may not be applicable to all sorts of models or devices. Additionally, integration and compatibility with existing ML frameworks or external platforms may pose a challenge as the DSL is tailored to the needs of Wiretronic.

Finally, our evaluation does not address any standard benchmarks for ML, which are not available for our addressed problem of edge deployment. Importantly, our scope is separate from model development, where standard benchmarks such as MNIST are widely used. These benchmarks do not come with a pipeline of deployment steps (pre- and postprocessing) that would benefit from execution with platform-specific libraries, the focus of our approach.

Construct Validity

Construct validity reflects if the measurements really represent what they are meant to do [43]. During the controlled experiment we saw the low amount of lines of code needed for our new DSL approach. It can be argued that our experiment is not comprehensive enough, but the tasks were designed with this specifically in mind. One of the main pain points of Wiretronic was the need to write unnecessary amounts of code to make these small tweaks. Another point is the completeness of the DSL, the experiment does not capture all functionalities of the DSL.

A threat to our study of performance, in the context of validating the performance requirement in 4.3.3, is that we might neglect other important dimensions of performance, such as the scalability when ML models, datasets and pipelines grow in size. Importantly, our approach is not directly involved with the processing of ML models or their underlying datasets. The models processed by our approach are DSL instances that describe ML deployment pipelines. For the actual processing of ML models, we employ available platform-specific libraries. The automated use of such libraries, which otherwise would have to be set up manually by developers, is facilitated by our model

interpretation engine, explained in Section 4.3.2. As the performance behavior for ML models is inherited from the underlying libraries, a performance evaluation on larger ML models or datasets would not evaluate our approach, but the used libraries.

As discussed in Section 5.2.4, our approach does lead to a performance penalty related to pipeline preparation (specifically, loading and processing the DSL instance). We report on benchmarking results measuring the impact of this penalty, which was as small as 24 milliseconds for typical instances of our DSL. By comparison, the overall time taken for loading the ML models and libraries, which is unavoidable regardless of whether our approach is used or not, was significantly greater, dwarfing the performance penalty of our approach. Scalability to larger DSL instances (that is, pipelines with significantly more of pre- and postprocessing steps) was not a relevant requirement of our industry partner, as the number of pre- and postprocessing steps is generally small in practical usage contexts.

7 Related work

We now discuss related work in the directions of machine learning in cross-platform Mobile Environments, MDE in edge devices, and domain-specific languages in the deployment and development of machine learning models.

7.1 ML in Cross-Platform Mobile Environments

In the deployment of machine learning (ML) models within cross-platform mobile environments, it is often advantageous to utilize platform-optimized ML frameworks. An example of such a framework is Core ML for iOS [5]. The advantages presented by such an approach increase as the ML model requires interaction with other hardware functionalities, such as the camera of the device. In a scenario where continuous inference on a camera stream is required, significant processing time can be saved by performing the entire computation flow on the native layer, as it omits the data serialization introduced by inter-layer communication [13]. Yet, the platform-optimized logic for each target platform needs to be implemented for each application anew, a disadantage we aim to mitigate with our model-driven approach, in which the knowledge about the capabilities of the target platforms is encoded in the model interpretation engine.

7.2 MDE in Edge Devices

As hardware improves and new areas of applicability arise, the demand to deploy ML models on edge devices increases. However, integrating ML models into edge device environments still comes with many limitations in terms of computational resources, power constraints, and network communication [44]. Furthermore, there is a significant heterogeneity in edge devices, spanning from low-memory microcontrollers to high-end smartphones. Working in this domain can therefore require familiarity with several techniques and operating systems. Within the specific domain of mobile development, Vaupel et al. [45] discuss how model-driven techniques can be used to create flexible, cross-platform mobile applications, stating that models should be "As abstract as possible and as concrete as needed." [45, 46]. By opting for model-driven techniques

and using higher abstraction levels we can create separate native builds from a single source, similarly to techniques mentioned in Section 2.2.

7.3 Domain-Specific Languages in the Deployment and Development of ML Models

DSLs can play a significant role in the deployment of ML models, especially on edge devices where computation power and memory are limited. A DSL can help ensure type and function compatibility, which is an integral part for models used for tasks such as image recognition and text processing, as well as providing the ability to efficiently manage tasks such as inputs and outputs. Zhao et al. [47] introduce a system that exemplifies the use of a DSL in such a context. However, their focus is to enable the service-oriented composition and deployment of data analytics services in a heterogeneous edge environment, where different services can be deployed to different backends. They do not support the specification of pre- and processing logic and its execution using the custom hardware capabilities of different target platforms, as is the focus of our approach. Similarly, in an IoT context, Moin et al. [48, 49] propose the MLQuadrat approach for modeling a system architecture that can incorporate machine learning and data science aspects and generate code from it. A primary focus of their work is on model development, as their approach, unlike ours, supports model training on IoT devices. While they provide a mode for using a black-box ML model, they do not focus on our addressed task of generating code for the same deployment pipeline to several, heterogeneous devices, which is our focus. Their approach is holistic, as it assumes modeling structure and behavior of the entire software architecture, whereas we provide a lightweight, targeted MDE solution for a specific task in a potentially larger system (e.g., an app with significant UI and other non-ML components).

Beyond the development and deployment, MDE has also been used to support the management of orthogonal ML aspects, such as asset management and dataset management. Traditional version control systems (VCS) can struggle to handle complex assets such as ML models and datasets. In the paper by Idowu et al. [50], they address these asset management challenges by introducing the Experiment Management Meta-Model (EMMM). A meta-model to characterize ML asset structures as concepts and their relationships observed in state-of-the-art tools, and conceptual VCS structures that can hold both ML and traditional assets. Focusing on machine learning experimentation and model development as well, d'Aloisio et al. present approach for the quality-driven development of machine learning software through extended feature models that capture aspects of the training pipeline and can be used to systematically identified configurations to enhance the given quality attributes. With a focus on supporting the construction of new deep learning frameworks, Atouni [52] el al. have published an artifact and reference model that supports build tasks and data management in this context. Giner-Miguelez et al. [53] presents DescribeML, a tool for utilizing a DSL to describe datasets. This tool aims to enable a more data-centric approach in ML, to handle issues like undesired model behaviors resulting from biased predictions.

7.4 ML Experiment Management Tools and MLOps

As ML-based software has become widespread, there is a trend from the ad hoc development of machine learning projects to more systematic workflows that exploit automation, based on dedicated tools. Two relevant terms in this context are *ML experiment management*, which focuses on the systematic management of assets during the experimentation and model training phases, and *MLOps*, which additionally considers deployment and monitoring aspects. Idowu et al. [25, 54] studied 30 available tools in this technical sphere. As two selected examples, we consider MLFlow and Comet.ML. Both tools have dedicated support for experiment management, such as providing a tracking API for logging parameters and results during experiments, Towards deployment, MLFlow has dedicated support for Cloud deployment, with a focus on the containerization of models together with their dependencies, whereas Comet.ML primarily focuses on model monitoring facilities with dashboards and visualizations. However, to our knowledge, no existing tools in this technical sphere include dedicated features for bridging technical heterogeneity of deployment to end devices with custom, platform-specific code, which is the focus of our approach.

8 Conclusion

and Future Work In this paper, we present the development of a DSL specifically for defining an ML serving pipeline in a cross-platform environment. We aimed to simplify the deployment and maintenance of ML models on edge devices, focusing on the unique challenges present in this environment. These challenges included software deployment to external devices and working in cross-platform environments with several code bases. The first cycle was about defining the scope through initial interviews with Wiretronic engineers and obtaining an overview of the domain through researching relevant literature. Combining Wiretronic's needs and thoughts with what technologies would be the optimal approach for them, ultimately comes down to the lightweight and easy-to-learn JSON Schema approach. Allowing the developers to in one single configuration file change quickly what before would need a redeployment of their library. The second cycle used the information gathered in the first cycle to design and implement the artifact. Using the guidance from the research, interviews, and engineers to ensure usefulness for the company and compatibility with their current systems. Lastly, the third cycle conducts the controlled experiment followed by second interviews to evaluate the artifact, both in terms of metrics and subjective reactions from the engineers. Our research confirms that a DSL can reduce the complexity typically involved when writing equivalent functionality for several platforms. By abstracting platform-specific code, the DSL enables developers to define ML models and their required inputs, outputs, and pre- and postprocessing stages, simplifying the aspects mentioned in our evaluation. In conclusion, the development of our artifact represents a step toward simplifying the deployment of ML models especially on smartphones in a cross-platform environment. It bridges the gap between ML engineers and other developers, enabling common understanding and lowering the barrier of entry to cross-platform development at Wiretronic.

The work done during this study can be expanded on, both within the domains covered and across other areas of software engineering. First, there are numerous possibilities for adding more features to the DSL, increasing its functionality and flexibility. For instance, a scenario where Wiretronic needs to execute different pre- and postprocessing steps depending on the deployment platform. Additionally, the current requirement of having a separate configuration file for each model could be streamlined by extending the DSL to integrate all model configurations into a single, more compact file. While there are many possibilities, it is important to note this could also lead to increased complexity. Second, to explore further applicability of the DSL within the same domain, it can be beneficial to implement a simple way to specify if pipeline steps should be executed on only some platforms. Furthermore, to continue the work within the same or adjacent domains, a more generalizable solution can be explored. This study focused only on the needs of a single company, primarily developing ML models for image-based tasks. Due to the small size of Wiretronic's AI division, it would be valuable to study the applicability of similar artifacts in a larger organization. This could both help in refining and generalizing the DSL and help combat the validity threats related to the small sample size of this study. Expanding the focus to adjacent domains, it can be beneficial to study both the potential of using a lightweight DSL to describe ML serving pipelines in environments other than smartphones and to explore its applicability for scenarios other than image-based ML. Third, although the developed artifact was tailored to the needs of Wiretronic, future research can be conducted into the impact of employing lightweight DSLs in other settings, for organization-specific scenarios. While the development of a more traditional DSL can be expensive in both development and adoption time, opting for a lightweight and familiar approach can lower both development and adoption time, while simultaneously lowering the barrier of entry for contributing to the DSL. In addition, while our current implementation is based on Flutter, due to Wiretronic's requirements, the underlying principles and methodologies we employed are not inherently tied to this framework. The core concepts of our model-driven approach, such as the use of a DSL for specifying ML serving pipelines and the model interpretation engine, can be adapted to other cross-platform frameworks like React Native or Xamari. Finally, our preliminary evaluation demonstrates a need for standard benchmark for our considered problem of ML deployment on edge devices.

References

[1] Chollet, F.: Deep Learning with Python, 2nd edn. Manning Publications, Shelter Island, New York (2021). Chap. 1.2.7

[2] wirevision. https://apps.apple.com/se/app/wirevision/id1543684933

[3] Husqvarna Gear Identifier. https://apps.apple.com/se/app/husqvarna-gear-identifier/id6464394076

[4] Flutter. https://flutter.dev/

[5] Core ML. Apple Inc. Accessed: 2024-01-22 (2023). https://web.archive.org/web/20231126225328/https://developer.apple.com/machine-learning/core-ml/

[6] Custom Models — ML Kit. Google. Accessed: 2024-01-22 (2023). https://web.archive.org/web/20231208182614/https://developers.google.com/ml-kit/custom-models

[7] The authors: Online artifact for 'Cross-Platform Edge Deployment of Machine Learning Models: A Model-Driven Approach' (2024). https://drive.google.com/drive/folders/1y_3t7hfGKS7yujkOkGmSzya7PJ8RZHKZ

[8] Lai, L., Suda, N.: Rethinking Machine Learning Development and Deployment for Edge Devices (2018)

[9] Bayerl, S.P., Frassetto, T., Jauernig, P., Riedhammer, K., Sadeghi, A.-R., Schneider, T., Stapf, E., Weinert, C.: Offline Model Guard: Secure and Private ML on Mobile Devices. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 460–465 (2020)

[10] Song, H., Dautov, R., Ferry, N., Solberg, A., Fleurey, F.: Model-based fleet deployment of edge computing applications. In: Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2020, pp. 132–142 (2020)

[11] Zhang, Q., Li, X., Che, X., Zhou, A., Xu, M., Wang, S., Ma, Y., Liu, X.: A Comprehensive Benchmark of Deep Learning Libraries on Mobile Devices. In: Proceedings of the ACM Web Conference 2022 (WWW '22), p. 10. ACM, New York, NY, USA (2022)

[12] Stack Overflow Developer Survey 2023. Stack Overflow. Accessed: 2024-01-22 (2023). https://web.archive.org/web/20240121213425/https://survey.stackoverflow.co/2023/#most-popular-technologies-misc-tech-prof

[13] React Native: Why a New Architecture. Accessed: 2024-02-22 (2023). https://web.archive.org/web/20231206110642/https://reactnative.dev/docs/the-new-architecture/why

[14] Flutter: Writing custom platform-specific code. Accessed: 2024-02-22. https://web.archive.org/web/20240214020240/https://docs.flutter.dev/platform-integration/platform-channels

[15] Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Springer, Cham, Switzerland (2017). https://link.springer.com/10.1007/978-3-031-02549-5

[16] Mernik, M., Cwi, J.H., Sloane, A.M.: When and How to Develop Domain-Specific Languages (2005)

[17] Xtext. Accessed: 24.01.24. https://eclipse.dev/Xtext/

[18] Iung, A., Carbonell, J., Marchezan, L., Rodrigues, E., Bernardino, M., Basso, F.P., Medeiros, B.: Systematic mapping study on domain-specific language development tools. Empirical Software Engineering **25**, 4205–4249 (2020)

[19] Chavarriaga, E., Jurado, F., Rodríguez, F.D.: An approach to build JSON-based domain specific languages solutions for web applications. Journal of Computer Languages **75**, 101203 (2023)

[20] Habib, A., Shinnar, A., Hirzel, M., Pradel, M.: Finding data compatibility bugs with json subschema checking, pp. 620–632 (2021). Association for Computing Machinery, Inc

[21] LinkedIn: What are the most common data conversion formats and standards in your industry? Accessed: 2024-02-19 (2023). https://www.linkedin.com/advice/0/what-most-common-data-conversion-formats-standards

[22] The Top 10 Data Interchange or Data Exchange Format Used Today. Accessed: 2024-02-19. https://aster.cloud/2023/05/11/the-top-10-data-interchange-or-data-exchange-format-used-today/

[23] JSON Schema: Getting Started Step by Step. Accessed: 2024-02-19 (2023). https://json-schema.org/learn/getting-started-step-by-step

[24] Attouche, L., Baazizi, M.-A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: Validation of Modern JSON Schema: Formalization and Complexity (2024)

[25] Idowu, S., Osman, O., Strüber, D., Berger, T.: Machine learning experiment management tools: a mixed-methods empirical study. Empirical Software Engineering **29**(4), 1–35 (2024)

[26] Knauss, E.: Constructive master's thesis work in industry: Guidelines for applying design science research. In: 2021 IEEE/ACM 43rd International Conference on Eoftware Engineering: Software Engineering Education and Training (ICSE-SEET), pp. 110–121 (2021)

[27] Matzler, K., Hinterhuber, H.: The kano model: How to delight your customers, pp. 313–327 (1996). https://www.researchgate.net/publication/240462191

[28] Cohn, M.: User Stories Applied: For Agile Software Development. Addison-Wesley Professional, Boston, MA, USA (2004)

[29] Martin, R.C.: Design Principles and Design Patterns, pp. 12–14 (2000)

[30] Interaction Design Foundation - IxDF: What are User Stories? https://www.interaction-design.org/literature/topics/user-stories. Accessed: 02.02.24 (2016)

[31] Lauesen, S.: Software Requirements-Styles and Techniques, (2002)

[32] Huang, J., Li, Y.F., Xie, M.: An empirical analysis of data preprocessing for machine learning-based software cost estimation. Information and Software Technology **67**, 108–127 (2015)

[33] Non-Functional Requirements: Examples, Types, and How to Approach Them. https://www.altexsoft.com/blog/non-functional-requirements/. Accessed: 2024-02-26 (2023)

[34] The JSON Data Interchange Syntax. Accessed: 2024-04-14 (2017). https://web.archive.org/web/20240407090452/https://ecma-international.org/publications-and-standards/standards/ecma-404/

[35] Gao, L.: Latin squares in experimental design. Michigan State University (2005)

[36] Baltes, S., Ralph, P.: Sampling in software engineering research: A critical review and guidelines. Empirical Software Engineering **27**(4), 94 (2022)

[37] Stol, K.J., Fitzgerald, B.: The ABC of software engineering research. ACM Transactions on Software Engineering and Methodology **27** (2018)

[38] SciPy Developers: scipy.stats.mannwhitneyu. Accessed: 2024-05-20. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html

[39] Sijtsma, K., Emons, W.: Nonparametric statistical methods. In: International Encyclopedia of Education, pp. 347–353. Elsevier, Amsterdam, Netherlands (2010)

[40] statsmodels Developers: statsmodels.stats.contingency_tables.mcnemar. Accessed: 2024-11-19. https://www.statsmodels.org/dev/generated/statsmodels.stats.contingency_tables.mcnemar.html

[41] Jonsson, R.: Exact properties of mcnemar's test in small samples (1993)

[42] McShane, B.B., Gal, D., Gelman, A., Robert, C., Tackett, J.L.: Abandon statistical significance. The American Statistician **73**(sup1), 235–245 (2019)

[43] Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering (2012). https://link.springer.com/book/10.1007/978-3-642-29044-2

[44] Moin, A., Challenger, M., Badii, A., Gunnemann, S.: Supporting AI engineering on the IoT edge through model-driven TinyML. In: Proceedings - 2022 IEEE 46th Annual Computers, Software, and Applications Conference, COMPSAC 2022, pp. 884–893 (2022)

[45] Vaupel, S., Taentzer, G., Gerlach, R., Guckert, M.: Model-driven development of

mobile applications for Android and iOS supporting role-based app variability. Software & Systems Modeling **17**, 35–63 (2018)

[46] Vaupel, S., Strüber, D., Rieger, F., Taentzer, G.: Agile bottom-up development of domain-specific IDEs for model-driven development. In: FlexMDE'15: Workshop on Flexible Model Driven Engineering, Part of MODELS 2015, pp. 12–21 (2015)

[47] Zhao, J., Tiplea, T., Mortier, R., Crowcroft, J., Wang, L.: Data analytics service composition and deployment on edge devices. In: Big-DAMA 2018 - Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks, Part of SIGCOMM 2018, pp. 27–32 (2018)

[48] Moin, A., Challenger, M., Badii, A., Günnemann, S.: A model-driven approach to machine learning and software modeling for the iot: Generating full source code for smart internet of things (iot) services and cyber-physical systems (cps). Software and Systems Modeling **21**(3), 987–1014 (2022)

[49] Kirchhof, J.C., Kusmenko, E., Ritz, J., Rumpe, B., Moin, A., Badii, A., Günnemann, S., Challenger, M.: Mde for machine learning-enabled software systems: a case study and comparison of montianna & ml-quadrat. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 380–387 (2022)

[50] Idowu, S., Strüber, D., Berger, T.: EMMM: A unified meta-model for tracking machine learning experiments. In: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 48–55 (2022)

[51] d'Aloisio, G., Di Marco, A., Stilo, G.: Democratizing quality-based machine learning development through extended feature models. In: International Conference on Fundamental Approaches to Software Engineering, pp. 88–110 (2023). Springer Nature Switzerland Cham

[52] Atouani, A., Kirchhof, J.C., Kusmenko, E., Rumpe, B.: Artifact and reference models for generative machine learning frameworks and build systems. In: Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 55–68 (2021)

[53] Giner-Miguelez, J., Gómez, A., Cabot, J.: DescribeML: a tool for describing machine learning datasets. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 22–26 (2022)

[54] Idowu, S., Strüber, D., Berger, T.: Asset management in machine learning: State-of-research and state-of-practice. ACM Computing Surveys **55**(7), 1–35 (2022)

Appendix 1 - Interview Guide for Initial Interviews

- Can you explain what code is equivalent but different in implementation across platforms, and give examples from your experience where you had to write or manage it?
- How often do you have to edit this code?
- How much work is it to make adjustments to this code?
- Is the file structure consistent across other projects you work on?
  - Do you have a standardized file structure or guideline to use across projects?
- Is the code in its current form reusable across other repositories, or is it context-specific?
- What IDE do you use for this work?
  - Do you use different IDEs for the different languages (Java/Swift)?
- How do you do testing on native mobile code?
- Would it speed up your workflow if you had a tool to generate platform-specific code for you by having to specify it only once in a file, similar to a configuration file?
  - If so, how would you imagine the tool being implemented?

Appendix 2 - Interview Guide for Experiment Interviews

- How extensive is your previous experience in deploying ML models in a cross-platform mobile environment on a scale from 1 (no experience at all) to 5 (very experienced)?
  - How many years have you spent doing that work?
- What was your general impression of using the DSL to implement and make changes to an ML serving pipeline?
- After using the DSL in the experiment, how would you rate it for the following properties (from 1 to 5):
  - Intuitiveness - how intuitive and natural was the DSL to use?
  - Learnability - how fast could you get up to speed using the DSL?
  - Usability - how usable was the DSL for the task at hand?
- Based on your own experience in implementing ML models: How would you rate the usefulness of the DSL and the accompanying library on a scale from 1 (not useful at all) to 5 (very useful) in your day-to-day development workflow, and why?
- Can you think of any scenarios where the DSL in its current form would be particularly useful?
  - If so, are there any features you would want the DSL to have?
- Can you think of any scenarios where the DSL in its current form would not be useful?

– If so, are there any existing features that are unnecessary?