4DT903 Group Report

Model-Driven Development of Notebook-to-Project Transformation System

Samuel Berg, Jesper Wingren & Emil Ulvagården

January 2026

GitHub Repository

Contents

1 Introduction

This report presents the collaborative work of our group on developing a model-driven engineering (MDE) solution for transforming Jupyter notebooks into structured project artifacts. The project addresses the challenge of converting exploratory data science notebooks into maintainable, production-ready code structures.

1.1 Background and Problem Description

Jupyter notebooks are widely used in data science for exploratory analysis, prototyping, and documentation. However, transitioning from notebooks to production-ready projects presents several challenges including code organization, modularity, testing, and maintainability.

Jupyter notebooks can in general be messy and lack structure if not properly created and maintained. Therefore this project aims to take those messy notebooks and structure them into a python project which is both easier to maintain and use in production.

1.2 Project Objectives

The main objectives of this project are:

- Design metamodels for representing notebook structures and project architectures
- Implement text-to-model (T2M) transformation form notebook files to notebook models
- Implement model-to-model (M2M) transformations from notebook models to project structure models
- Develop model-to-text (M2T) transformations to generate project artifacts
- Usage of some sort of AI to classify code blocks into different files
- Create a comprehensive pipeline demonstrating the full system flow from notebooks to python
- Show a proof of concept that it this kind of pipeline works and could be useful fo use in production

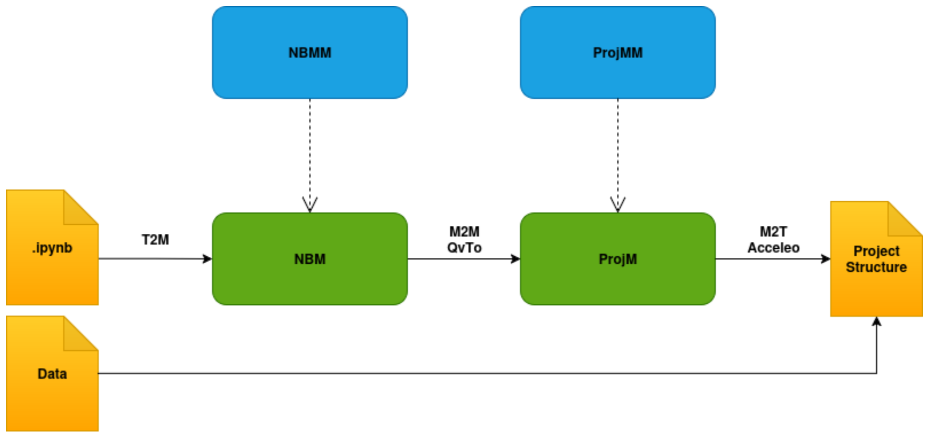

Figure 1: Application Pipeline

2 Application Overview

The applications objective is to turn a jupyter notebook into a python project. It begins by taking input files.
Input:

- .ipynb file (source code)
- Data
- Models

With the inputs the pipeline which consists of 2 models, 2 MetaModels and 3 transformations turns the notebook project into a python project which is further explained in section 3. The application uses docker to copy over data files from the input to output. It begins by turning the source code to objects inside our notebook model using a text-to-model transformation. Separating imports, code and comments into different objects. This notebook model is then transformed using model-to-model to a project model which comes from the project MM. This project model is then transformed using model-to-text into a python project containing folders and files. The output of the project pipeline is:
Output:

- main.py
- Folder structure
- requirements.txt
- Docker file

3 Architecture

3.1 Notebook Metamodel

The Notebook Metamodel serves as the foundation for the abstract representation of input data. It is designed to capture the different parts of a Jupyter Notebook.

3.2 Project Metamodel

In contrast to the source, the Project Structure Metamodel defines the target state of a python project.

3.3 Text-to-Model Transformation

The first stage is transforming the notebook code into a model object.

3.4 Model-to-Model Transformation

The transition from a notebook model to a project model is handled by the Model-to-Model (M2M) transformation using QvTo. This component is responsible for the mapping of flat notebook elements into the project structure.

3.5 Model-to-Text Transformation

The final stage of the pipeline is the Model-to-Text (M2T) transformation using Acceleo, which serializes the project model into concrete file system artifacts. Using template-driven generation, this layer produces Python source files. Beyond source code, the model-to-text transformation generates the necessary infrastructure for a project, including dependency information like `requirements.txt`, installation scripts such as `setup.py`.

4 MDSE Problems and Solutions

We encountered challenges ranged from tool-specific instabilities to architectural decisions regarding where logic should reside within the transformation chain.

4.1 The Integration and Pipeline Challenge

Our initial development strategy involved building and testing the Text-to-Model (T2M), Model-to-Model (M2M), and Model-to-Text (M2T) transformations as isolated units. While these units functioned correctly in a standalone environment, integrating them into a continuous, automated pipeline proved difficult. The primary challenge was ensuring that the output of one transformation was perfectly compatible with the input requirements of the next stage. With guidance from our supervisor, we resolved this by formalizing the execution order, ensuring that the project-specific model was fully instantiated before the code generation templates were invoked.

4.2 Tooling Instability

A significant technical hurdle was the instability of the Eclipse Metamodel Graphical Editor. Frequent crashes during the visual design of our Ecore metamodels. To mitigate this and maintain project momentum, we bypassed the GUI and worked directly with the XMI. While this made the visual representation of our architecture less intuitive, it provided a more stable and precise environment for defining the constraints and attributes of our Notebook and Project metamodels.

4.3 Architectural Mapping: M2M vs. M2T Responsibility

Later in the work, we encountered a conceptual problem regarding the "Separation of Concerns" within our transformations. We initially generated static infrastructure files—such as the `Dockerfile` and `Server.py` within the Model-to-Model layer. This resulted in an over-complicated M2M transformation. This violated the core of MDSE where the M2M layer should focus on structural mapping, while the M2T layer should handle syntax and file generation. By refactoring the system and moving these responsibilities to the Acceleo-based M2T templates, we simplified the QVTo logic and made the overall system more maintainable and easier to debug.

5 Discussion

5.1 Strengths

The primary strength of this system lies in the separation of concerns achieved through the use of metamodels. This modularity results in reusable transformation logic, for instance, the Notebook metamodel and its corresponding T2M parser could be utilized in different projects using notebooks. Furthermore, the use of mappings and ecore models makes changing and managing the code easier as well as utilizing the strengths of MDSE.

5.2 Limitations

Despite its successes in testing, the current implementation has certain limitations, particularly regarding the support for different notebook patterns. Furthermore, the scope of the project was limited to the most common notebooks containing standard python environments.

5.3 Challenges Encountered

Throughout the development process, the most significant challenge was connecting the notebook model to our project model using QvTo as it proved to be difficult. We also encountered problems finding a way to classify the different code blocks to separate into different files with later deciding that it would be future work. On a technical level, the integration and setup of the Eclipse Modeling Framework and its associated transformation plugins proved to be quite hard, highlighting the steep entry barrier often associated with professional MDE tooling.

5.4 Lessons Learned

The project provided valuable insights into the importance of an iterative design process, particularly regarding metamodels. We learned that metamodels should not be considered finished until several cycles have been completed. The development also highlighted the value of testing transformations early in the lifecycle and catching a logic error in a smaller script is far more efficient than debugging the final generated source code.

5.5 Improvements

One improvement that could be made is to try classifying the code better and creating several python files such as training.py and preprocessing.py which would make the python project even better and more organized. This would although imply on even more work to make the python project run.

5.6 Extensions

One possible extension is that for a larger project a static analysis could also be integrated to perform automated refactoring, such as identifying duplicate code across cells and consolidating them into a single utility module. From an operational perspective, integrating the transformation pipeline into standard CI/CD workflows or developing a dedicated GUI for configuration would make the system more accessible to data scientists.