4DT903 Individual Report

Reflections on Model-Driven Software Engineering

Samuel Fredric Berg (sb224sc)

January 2026

Contents

1 Core Components and Responsibilities

Model-Driven Software Engineering (MDSE) systems are built upon three primary pillars that work together to transform abstract models into concrete implementations. The first pillar consists of **models**, which are high-level abstractions of software that focus on what the system does rather than how it is implemented. They represent specific system instances. In our notebook-to-project transformation system, the notebook model captures the structure of Jupyter notebooks, containing code cells, markdown cells and metadata as concrete instances conforming to the notebook metamodel.

The second pillar comprises **metamodels**, which define the language of the system by specifying rules, constraints and structural elements. They act as the "schema" for models, defining the vocabulary and grammar of the domain. Our project uses two metamodels: the Notebook Metamodel, which defines elements like CodeCell, MarkdownCell and Import statements and the Project Structure Metamodel, which defines elements like PythonFile, Folder and dependencies. These metamodels establish the structural foundation upon which all transformations operate.

The third pillar encompasses **transformations**, which handle the conversion between different levels of abstraction and form the operational backbone of MDSE. Text-to-Model (T2M) transformations parse source artifacts, such as .ipynb files, into structured models conforming to a metamodel. In our implementation, this transformation reads JSON-formatted notebook files and instantiates model objects. Model-to-Model (M2M) transformations convert one model into another, potentially using different metamodels. Our QVTo-based M2M transformation maps notebook model elements into project structure elements, organizing code cells into appropriate Python modules. Finally, Model-to-Text (M2T) transformations generate final artifacts like source code or documentation from a model. Our Acceleo-based M2T transformation produces Python files, requirements.txt and Dockerfiles from the project structure model.

2 Core Benefits of MDSE

The implementation of MDSE offers several technical and industrial advantages, demonstrated through our notebook transformation project. One of the most significant benefits is increased productivity through automation of code generation for repetitive or high-volume tasks, which saves significant time. Our system automatically generates project structure, boilerplate code and configuration files from notebooks, eliminating hours of manual restructuring work that would otherwise be required.

MDSE also provides improved quality because code follows a standardized pipeline, ensuring consistency across the project. Every generated Python file adheres to the same organizational principles, reducing variability and potential errors that commonly arise in manual development. This standardization extends to platform independence, where models are abstract enough to be deployed across different platforms as needed. The notebook metamodel could be reused in other contexts, such as generating R projects, Julia projects, or web-based notebook viewers, without requiring fundamental redesign.

Maintainability represents another crucial advantage, as system updates are easier because rebuilding the system from models ensures the pipeline remains consistent. Changes to output structure only require modifying the M2T templates, not manually updating every generated file. This approach significantly reduces the risk of introducing inconsistencies during maintenance. Additionally, automated processes eliminate the risk of

manual errors during repetitive coding tasks, such as when import statements are automatically collected and consolidated into requirements.txt, preventing dependency management errors. The clear separation between metamodels (structure), transformations (logic) and templates (syntax) makes the system easier to understand, test and maintain compared to monolithic code generation scripts.

3 Challenges and Limitations

Despite its strengths, MDSE presents several challenges that we encountered in our project. The development time required for creating robust metamodels and transformations is substantial and demands specialized expertise. Defining the Notebook and Project metamodels required multiple iterations as we discovered edge cases and refined our understanding of the domain. Each iteration revealed new requirements and constraints that necessitated metamodel modifications, which in turn required updating all dependent transformations.

The complexity of MDSE tools presents another significant hurdle. Sophisticated tools like QvTo and Acceleo have steep learning curves and without deep knowledge, developers may use them inefficiently. The QVTo mapping syntax and Acceleo template language required significant learning investment before we could implement effective transformations. The documentation for these tools, while comprehensive, assumes a level of modeling expertise that takes time to develop.

Pipeline debugging becomes increasingly difficult as the system grows. When the final output was incorrect, we had to trace back through M2T templates, M2M transformations and T2M parsing to locate the root cause. This multi-layered debugging process is more complex than debugging traditional imperative code, where errors typically manifest close to their source. Furthermore, there is a genuine risk that the complexity of building the MDSE infrastructure exceeds the complexity of the task it is meant to solve. For small-scale projects or one-off transformations, writing direct Python scripts might be more efficient than setting up an entire MDSE pipeline.

We also encountered significant tooling instability issues with the Eclipse Modeling Framework tools. The graphical Ecore editor crashed frequently, forcing us to edit XMI files directly, which reduced the usability of visual metamodeling features. This instability disrupted our workflow and made the development process more frustrating than it should have been. Additionally, connecting the separate transformation stages (T2M, M2M, M2T) into a continuous automated pipeline proved challenging. Ensuring that each stage's output was compatible with the next stage's input required careful interface design and extensive testing.

4 Problem Solving and Design Approach

Approaching a problem with MDSE involves shifting focus toward patterns and abstraction, as demonstrated in our notebook transformation project. The process begins with pattern analysis, where we analyze the domain to find recurring concepts and structures. In our project, we identified that notebooks consistently contain code cells, markdown cells, import statements and dependencies—these became the core concepts in our Notebook Metamodel. Similarly, Python projects follow predictable patterns with modules, packages, requirements files and configuration files. This pattern recognition phase is

crucial because it determines the scope and structure of the metamodels that will guide the entire transformation pipeline.

Once patterns are identified, the next step involves metamodel definition, where we create a Domain Specific Language (DSL) that captures these core concepts using Ecore. Our Notebook Metamodel defines classes like Notebook, CodeCell, MarkdownCell and Import, each with appropriate attributes and relationships. The Project Structure Metamodel defines PythonFile, Folder and Dependency classes that represent the target structure. The metamodel serves as a contract between different parts of the system, ensuring that all transformations operate on well-defined, consistent data structures.

With metamodels in place, we develop transformation rules to map data into model objects, utilizing tool-specific features like QVTo mappings to keep logic modular and testable. For example, our M2M transformation includes mappings like `notebook2project` that orchestrates the overall transformation and `codeCell2PythonFile` that handles individual code cell transformations. Each mapping encapsulates a specific transformation concern, making the logic easier to understand and maintain.

The implementation follows a sequential approach where we build and test transformations separately (T2M, M2M, M2T) before integrating them into a full automated pipeline. We first validated that T2M correctly parsed notebooks, then ensured M2M produced valid project models and finally verified M2T generated syntactically correct Python code. Only after each stage worked independently did we connect them. This incremental approach reduces debugging complexity and allows for faster identification of issues. Throughout this process, iterative refinement is essential—metamodels should not be considered final until several iterations have been completed. We discovered through testing that our initial metamodels were too rigid and had to be enhanced to handle edge cases like notebooks with no imports or cells with mixed code types.

5 Adaptation of Software Engineering Practices

Standard engineering practices must be modified to suit an MDSE environment, as we learned through our project. Version control in MDSE requires careful management to track changes within models and metamodels, not just traditional source code. We had to add .ecore, .genmodel, .qvto and .mtl files to version control, while excluding generated artifacts and Eclipse workspace files using .gitignore. Binary model files (.aird) posed challenges as they cannot be easily diff'd or merged, requiring more careful coordination when multiple developers work on metamodels. This is a significant departure from traditional software development where text-based source code files are easily managed with standard version control workflows.

Testing and validation practices also undergo a fundamental shift in MDSE environments. Rather than line-by-line code reviews, testing focuses on model validation and transformation testing. We needed to ensure models conformed to metamodel constraints, validate transformation logic independently of generated code and verify that generated code met functional requirements. Unit testing QVTo transformations and Acceleo templates required different approaches than testing traditional imperative code, as the logic is often declarative rather than procedural.

CI/CD integration presents unique challenges in MDSE projects, as automated pipelines must handle transformations in a strict sequential order. Our pipeline required

Eclipse modeling tools to be available in the CI environment, which meant either installing Eclipse in containers or using headless Eclipse applications. The pipeline must execute T2M, then M2M, then M2T in order, with each stage depending on the previous stage's output. This rigid sequencing is more complex than typical CI/CD pipelines that can often parallelize independent build and test tasks.

Documentation practices also shift significantly in MDSE projects. Rather than explaining code implementation details, documentation must focus on metamodel design decisions, transformation logic and the relationships between models. We documented not just what the code does, but why we chose certain metamodel structures, what design patterns influenced our transformations and how the different components interact. Understanding the "why" behind metamodel design is crucial for future maintenance, as changes to metamodels can have cascading effects throughout the entire transformation pipeline.

Finally, code reviews in MDSE projects must cover metamodel design, transformation correctness and template quality—not just generated code. Reviewers need to understand Ecore metamodeling, QVTo semantics and Acceleo template syntax, which requires specialized knowledge beyond traditional programming language expertise. This places higher demands on team members and makes the onboarding process more challenging for developers new to MDSE.

6 Specific MDSE Problems and Solutions

Throughout our notebook-to-project transformation development, we encountered several significant challenges that required MDSE-specific solutions:

6.1 The Integration and Pipeline Challenge

Our initial development strategy involved building and testing the T2M, M2M and M2T transformations as isolated units. While these units functioned correctly in standalone testing, 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. For example, the T2M transformation had to produce model instances that exactly matched what the M2M transformation expected as input.

Solution: We formalized the execution order and created explicit interface specifications between stages. We also added validation steps between transformations to catch incompatibilities early. With guidance from our supervisor, we ensured that the project model was fully instantiated before the M2T templates were invoked, preventing null reference errors and incomplete code generation.

6.2 Architectural Mapping: M2M vs. M2T Responsibility

We initially generated static infrastructure files—such as the Dockerfile and requirements.txt—within the M2M layer. This resulted in an over-complicated M2M transformation that mixed structural mapping with syntactic generation concerns. This violated the separation of concerns principle central to MDSE, where the M2M layer should focus on structural transformations between metamodels, while the M2T layer should handle text generation and syntax.

Solution: We refactored the system by moving all text generation responsibilities to the Acceleo-based M2T templates. The M2M transformation now only creates and populates project model elements, while M2T handles all file generation, including formatting, syntax and boilerplate code. This simplified the QVTo logic significantly and made the system more maintainable and easier to debug.

6.3 Handling Notebook Diversity

Jupyter notebooks vary widely in structure, coding style and organization. Some notebooks have clear section markers, while others are unstructured. Some have all imports at the top, while others scatter imports throughout the code. This diversity made it challenging to create a single metamodel and transformation pipeline that could handle all cases effectively.

Solution: We focused on the most common notebook patterns and created a metamodel flexible enough to capture various structures while still providing useful organization in the output. We also implemented normalization logic in the T2M stage to standardize certain aspects (like collecting all imports) before model-to-model transformation. This preprocessing approach reduced the complexity of downstream transformations.

7 Points Section

- Samuel Berg (sb224sc): 4
- Emil Ulvagården (eu222dq): 3
- Jesper Wingren (jw223rn): 3