GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

Doctoral Thesis

Supporting Domain-Independent Model Management through Automated REST APIs Generation

PhD Program in Computer Science: XXXVI cycle

Supervisor:

Prof. Ludovico Iovino
ludovico.iovino@gssi.it

Author:

Adiel Tuyishime
adiel.tuyishime@gssi.it

Co-Supervisor:

Prof. Javier Luis Cánovas Izquierdo
jcanovasi@uoc.edu

March 2025

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Abstract

The increasing complexity of modern software systems has made effective model management a crucial challenge in software engineering. Model-Driven Engineering (MDE), which places models at the core of the development process, has emerged as a promising approach to managing this complexity. The MDE paradigm is a software development method that aims to improve productivity and software quality by using models as primary artifacts in all aspects of the software engineering process. In this approach, models serve as abstract representations of a system, enabling data manipulation, property validation, and ultimately transformed into application artifacts such as code, documentation, and tests. While MDE pretended to be the silver bullet for software engineering, its adoption in distributed, web-based environments has revealed significant limitations, particularly in scalability when managing large and complex models within industrial contexts, ensuring seamless interoperability across diverse modeling tools, and simplifying model querying and manipulation, which often requires domain-specific knowledge that poses challenges for non-expert users.

Traditional model management tools often falter as models grow larger and more complex. These tools are generally ill-suited to handle the dynamic and rapidly evolving nature of modern systems, particularly in distributed, web-based environments, where managing and handling large-scale models efficiently is essential. Additionally, existing model-based tools often operate in isolation, relying on proprietary languages, and repositories, which hinder seamless model exchange and collaborative development. The lack of interoperable solutions leads to model silos, making models inaccessible to other model-based tools and platforms that rely on them. Another key challenge in model management is model querying and manipulation, which often require domain-specific knowledge that may not be accessible to non-expert users. Many existing solutions demand familiarity with custom query languages and intricate model structures, making it difficult for users without deep MDE expertise to efficiently access and manipulate model elements. A more user-friendly and intuitive approach is needed to lower the barrier to model interaction and enhance accessibility.

Current web-based model management approaches typically rely on static generative solutions to generate REST APIs. These solutions require predefined models and manual intervention, leading to inefficiencies and delays when adapting to evolving system requirements. Such limitations render them unsuitable for dynamic distributed environments, where models are not known in advance and REST APIs must be provisioned on-the-fly to support adaptive model management.

This research addresses these challenges by developing a middleware solution that facilitates scalable, interoperable, and efficient model management in distributed, web-based environments. The middleware serves as a bridge between model repositories and modeling tools, enabling efficient model exchange, manipulation, and querying in an environment independent of any specific technology. This solution enables dynamic REST APIs provisioning for model management, ensures scalability for handling large and complex models, and enhances interoperability across diverse model-based tools and platforms. Specifically, the contributions of this work are threefold. (i) Dynamic REST API provisioning, by designing and implementing a middleware solution capable of generating REST APIs on-the-fly for any given domain model, while providing a more user-friendly approach to model querying and manipulation. (ii) Scalability and efficient model handling to efficiently query, and manipulate large and complex models. (iii) Enhancing interoperability by enabling seamless model exchange across heterogeneous model-based tools and platforms.

This work makes a significant contribution to advancing MDE practices by addressing critical limitations in existing model management solutions, enabling more efficient, adaptable, and scalable model management in distributed, web-based environments.

Acknowledgements

First and foremost, I express my deepest gratitude to the Almighty God for His unwavering protection and guidance throughout this journey.

I am profoundly thankful to my advisors, Ludovico Iovino and Javier Luis Cánovas Izquierdo, for their invaluable guidance, patience, and insight. Their mentorship has been instrumental in shaping my growth as a researcher. This work would not have been possible without their constant support and belief in my abilities.

My heartfelt thanks go to Francesco Basciani and Martina De Sanctis for their steadfast support throughout my research journey. Your collaboration and encouragement have greatly enriched my experience.

I am also deeply grateful to Ilenia Taddei for her unwavering support during administrative processes at the GSSI. Your kindness and willingness to help in times of need have made a significant difference.

A special note of appreciation to Aimable Havugiyaremye for your support, and inspiration to embrace new horizons and explore the world. Your belief in me has been a source of strength.

To my beloved parents, thank you for instilling in me the values, resilience, and education that have guided me to this milestone. Your sacrifices and unconditional love are the foundation of all my achievements. To my siblings, your unwavering love and moral support have been a beacon of comfort and motivation.

To the family of Elie and Jeannette, thank you for your hospitality, kindness, and moral support. Your presence in my life has been a blessing and a source of encouragement.
To the family of Myriam Mukamageza, your kindness and generosity have meant so much to me.

Finally, to my dear friends, whom I have had the privilege of meeting throughout my academic journey, thank you for your friendship, encouragement, and countless memorable moments.

This journey has been shaped by each of you, and I will forever carry your kindness, support, and love in my heart.

Contents

List of Figures

Chapter 1

Introduction

This chapter provides an overview of the research presented in this dissertation. Section 1.1 explores the motivation driving this research. Section 1.2 defines the problem and outlines the key research challenges tackled in this study. Section 1.3 articulates the main objectives of the thesis. In Section 1.4, the research questions guiding this work are presented, along with a high-level overview of the proposed solution. Section 1.5 highlights the major contributions of this research. Section 1.6 provides an overview of the research publications that contributed to the development of the thesis. Section 1.7 references the tools and demonstrations developed as part of this research endeavour. Finally, Section 1.8 concludes the chapter with an outline of the thesis structure.

1.1 Motivation

In today's rapidly advancing technological landscape, software systems have become the backbone of modern society, transforming the way we live, work, communicate, and interact with the world around us. These systems drive innovation across a broad spectrum of industries, from healthcare and automotive to aerospace and finance, etc. Whether it is managing everyday activities like online shopping and banking or overseeing critical, safety-sensitive operations, software has emerged as an indispensable force, accelerating the digital transformation of businesses and governments alike.

As the reliance on software systems continues to increase, so too does their size and complexity grow exponentially. Organizations, governments, and individuals demand sophisticated and scalable solutions that can adapt to constantly evolving demands. Developing, maintaining, and scaling interconnected systems is a critical challenge, particularly as software becomes larger, more intricate, and continuously evolving. Traditional development practices, which have served the industry well in the past, now

struggle to keep pace with the increasing scale and complexity of modern software. This results in inefficiencies, bottlenecks, and ultimately, delays in delivering high-quality solutions.

One of the most effective strategies for managing this growing complexity is abstraction [6]. Abstraction simplifies the development process by focusing on high-level structures while omitting unnecessary details, allowing developers to manage increasingly complex systems. At the forefront of this abstraction is the concept of models as representations of the essential components of a system. Models not only facilitate better design and analysis but also enable automation, transforming the way software is built, maintained, and evolved. The shift toward Model-Driven Engineering (MDE) has emerged as a powerful solution to these challenges. In MDE, models are primary artifacts that drive the development process [7]. These models can be automatically transformed into functioning systems through code generation and model transformation techniques, dramatically reducing manual work and boosting productivity. By automating key portions of the software development lifecycle, MDE enhances scalability, maintainability, and overall efficiency, making it a vital approach in modern software engineering.

As MDE places models at the center of software development process, the need for efficient model management has become paramount [1]. Effective model management encompasses a range of tasks, including storage, querying, transformation, validation, and maintenance, throughout the entire model lifecycle. However, managing models effectively has become increasingly complex, particularly as they grow larger, more interconnected, and more integral to the development workflow. These challenges are particularly pronounced in distributed, web-based environments, where models are stored across multiple repositories and must be efficiently accessed by various users and model-based tools. Without effective model management, this decentralization can lead to inefficiencies and inconsistencies, ultimately disrupting the software development process.

These challenges underscore the critical need for innovative model management solutions that can address the scalability, interoperability, and dynamic requirements of modern systems, in distributed web-based environments. Existing approaches often fall short, as they are typically static, lack adaptability, or struggle with seamless integration across heterogeneous platforms. This thesis aims to bridge these gaps by developing a middleware solution that tackles key model management challenges in distributed, web-based environments. The proposed solution aims to provide a scalable, dynamic, and interoperable solution to efficiently manage models in web-based environments.

1.2 Problem Statement

MDE has emerged as a transformative approach to managing the growing complexity of software systems. By placing models at the core of the development process, it raises the level of abstraction, enabling developers to design, analyze, and implement systems by automating and streamlining various stages of the software development lifecycle. However, its effectiveness is constrained by several critical challenges, particularly in distributed, web-based environments.

One significant challenge is interoperability. Existing model-based tools and modeling frameworks often operate in isolation, each relying on proprietary languages, and repository structures [1, 8, 9]. This fragmentation creates barriers to seamless model exchange, hindering collaboration and integration across diverse environments. Existing solutions are often tightly coupled to specific tools or platforms, leading to vendor lock-in and limiting their applicability in heterogeneous settings. Achieving interoperability remains a central issue, especially in distributed environments where models are spread across multiple repositories and must be accessed and manipulated in a consistent, unified manner.

Another critical issue is scalability. As software systems and their corresponding models grow in size and complexity, traditional model management practices struggle to maintain performance and efficiency [10–13]. Distributed, web-based environments amplify this challenge, requiring systems capable of handling large-scale, and evolving models while ensuring high availability and responsiveness. Current tools often fall short in supporting the dynamic nature and scalability requirements of such environments, resulting in inefficiencies in model storage, querying, and transformation processes.

Beyond scalability, another fundamental challenge is the efficient management of models in distributed, web-based environments. Although Web technologies facilitate remote model creation and manipulation, existing model repositories often lack the flexibility, scalability, and adaptability needed for effective model management [14]. The nature of web-based environments further exacerbates these challenges, requiring solutions that can adapt to evolving models, efficiently manage large-scale models, and support real-time collaboration. This necessitates effective storage, retrieval, and manipulation of models, as well as seamless model exchange across diverse modeling tools and platforms. Another key challenge in model management is model querying and manipulation, which often require domain-specific knowledge that may not be accessible to non-expert users. Many existing query languages demand specialized skills, making it difficult for users without deep MDE expertise to efficiently retrieve and manipulate model elements.

Current solutions offering complete RESTful services for model management primarily rely on static generative solutions [15], which require predefined models in order to

generate the REST APIs and manual intervention whenever models evolve. This static approach is inadequate for the dynamic nature of distributed web-based environments, where models are not known in advance and the REST APIs must be provisioned on-the-fly. The lack of dynamic REST API provisioning further hinders adaptability to rapidly changing requirements, limiting the full potential of modern MDE practices.

These challenges underscore the need for a solution that addresses interoperability, scalability, and dynamic provisioning of REST APIs to support model management in distributed web-based environments. Such a solution should facilitate seamless model exchange and manipulation across diverse platforms, ensure efficient handling of large and complex models, and enable the dynamic generation of REST APIs for intuitive model interaction without requiring domain-specific expertise. Addressing these issues is crucial for advancing MDE practices and enabling more adaptive, scalable, and user-friendly model management solutions.

1.3 Objectives

The primary objective of this thesis is to propose and develop a middleware solution that enhances model management in distributed, web-based environments. This solution should address the key challenges identified in the problem statement, specifically focusing on the dynamic provisioning of REST APIs, scalability, and interoperability. The specific objectives of the research are as follows:

1. **Design and implement a middleware for model management through dynamic provisioning of REST APIs.**

   The primary objective is to design and implement a middleware solution capable of dynamically generating REST APIs for any given domain model. This includes provisioning REST APIs on-the-fly for model management without requiring manual intervention or predefined models. The solution should facilitate model management in a flexible and intuitive way, eliminating the need for domain-specific expertise.

2. **Address scalability issues in model management.**

   As software systems grow in size and complexity, scalability becomes a critical challenge in model management. To address this, the goal is to develop scalable solutions that enable the proposed middleware to efficiently handle increasingly large models. This involves optimizing model storage, querying, validation, and transformation processes to ensure high performance and responsiveness.

③ **Enable interoperability across diverse model-based tools and platforms.**

This research aims to enhance interoperability among diverse modeling tools and platforms in distributed environments. The goal is to develop a solution that facilitates seamless model exchange, manipulation, and integration while preventing vendor lock-in and ensuring uniform and reliable model management across different platforms.

1.4 Research Questions

To address the challenges outlined in the problem statement and advance the state of model management in distributed, web-based environments, this research focuses on answering the following key questions:

RQ1: *How can we design and implement a solution that enables dynamic, on-the-fly model management in distributed environments?*

This question focuses on developing a solution for dynamic, on-the-fly model management for any given domain model. It highlights the need for automated model provisioning and management without requiring manual reconfiguration or prior knowledge of the model structure. The goal is to enable seamless model accessibility, querying, and manipulation.

RQ2: *How can scalability be supported in model management to efficiently handle complex and interrelated models?*

This question addresses the scalability challenges associated with managing increasingly large and complex models. The research investigates strategies to optimize model retrieval and query processing, emphasizing efficient model manipulation and advanced query-handling mechanisms.

RQ3: *How can we achieve seamless interoperability to facilitate the consistent exchange and manipulation of models across heterogeneous model-based tools and platforms in distributed environments?*

This question investigates solutions to overcome the fragmentation caused by proprietary tools and different technologies, ensuring that models can be shared and managed seamlessly across diverse model-based tools and platforms. The focus is on eliminating vendor lock-in and enabling collaborative model management in distributed, web-based environments.

These research questions collectively aim to bridge the gaps in existing MDE practices, providing innovative solutions for model management in distributed, web-based

environments. The answers to these questions will contribute to the development of a middleware solution and tools that advance interoperability, scalability, and REST APIs generation in MDE.

1.5 Research Contributions

This thesis presents several key contributions to the field of MDE, especially model management in distributed, web-based environments. The contributions focus on addressing the critical challenges of dynamic, on-the-fly model management, scalability, and interoperability. The primary contributions of this research are as follows:

(1) **Middleware solution for dynamic REST APIs provisioning.**

A central contribution of this thesis is the development of a middleware solution capable of dynamically provisioning REST APIs for any given domain model, even in the absence of predefined models. The middleware is designed to automatically generate API endpoints that correspond to the structure of the given domain model. This approach facilitates seamless model interaction, enabling users to efficiently query, manipulate, and manage models. This work extends the capabilities of existing MDE tools by enabling model management for any domain model in web-based environments.

(2) **Scalable model management solution.**

Scalability is a critical concern in managing large, complex models in distributed environments. This thesis contributes to the development of scalable model management solutions by proposing efficient techniques for model querying and manipulation. The middleware solution introduced in this work is designed to accommodate increasing model size and complexity while maintaining high performance as the system scales. This contribution supports the growing need for scalable MDE solutions, enabling effective model management in large-scale distributed systems.

(3) **Enhancing interoperability across diverse model-based platforms and tools.**

This research advances interoperability in MDE by proposing a solution that enables seamless model exchange and manipulation across diverse model-based tools and platforms. The middleware facilitates model interaction and exchange across heterogeneous tools and platforms, overcoming model silos imposed by proprietary tools and diverse technologies.

④ **A platform agnostic solution for model management**

Another key contribution is the development of a platform-agnostic solution that generates REST APIs for model management. Designed for both domain experts and non-technical users, the solution simplifies common model manipulation operations (i.e., Create, Read, Update, Delete). By abstracting technical complexities, it enables users to efficiently manage models in a web-based environment, promoting broader adoption of MDE practices. This contribution bridges the gap between technical expertise and model management, making MDE more accessible to a wider audience.

Through these contributions, this thesis aims to advance model management by providing solutions that address key challenges faced by MDE practitioners in web-based environments.

1.6 Research Publications

The research presented in this thesis has led to the development of several scholarly publications that contribute to the advancement of MDE, particularly in the areas of model management, scalability, and interoperability.

The following is a list of research publications that support this dissertation.

[IEEE ICWS 2024]: ***Adiel Tuyishime****; Francesco Basciani; Ludovico Iovino; Javier Luis Cánovas Izquierdo:* Dynamic Provisioning of REST APIs for Model Management (WIP Paper). In: IEEE International Conference on Web Services (ICWS 2024) [16].

This paper presents a middleware-based approach for dynamically provisioning RESTful services for model management. The proposed solution enables on-the-fly API generation, allowing seamless interaction with models in a web-based environment.

[MODELS 2024]: ***Adiel Tuyishime****; Francesco Basciani; Javier Luis Cánovas Izquierdo; Ludovico Iovino:* Enhancing Model Management with Automated REST API Generation (Tool Demo Paper). In: ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS 2024) [17].

This paper presents a tool that automatically generates RESTful services for model management, enabling interaction with any

given domain model in a web-based environment. The approach supports the development of web-based modeling tools, providing modeling-as-a-service while ensuring a stateless and scalable service. By exposing models through REST APIs, our solution allows users to seamlessly query and manipulate models on the web with minimal effort making it a more user-friendly and efficient approach for distributed model management. ensures a stateless and scalable service.

[MODELS 2023]: ***Adiel Tuyishime**; Francesco Basciani; Ludovico Iovino; Javier Luis Cánovas Izquierdo; Jordi Cabot; Alfonso Pierantonio:* Bridging Workflow Automation Tools and EMF Modeling Ecosystems (Workshop Paper). In: ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS 2023) [18].

This paper presents a bridging mechanism that integrates workflow automation tools with the Eclipse Modeling Framework (EMF) ecosystem. The proposed solution enhances interoperability, enabling seamless model exchange by allowing workflow automation tools to interact with EMF-based models.

1.6.1 Other Publications

In addition to the primary research activities presented in this dissertation, various other research projects and collaborations were undertaken during the PhD program. These additional activities explored the application of MDE across a broader range of domains, including Workflow Automation Tools (WATs), the Internet of Things (IoT), and software quality. This broader engagement provided a comprehensive perspective on the versatility and potential of MDE in diverse contexts.

The following papers were produced during these activities.

[SEAA 2024]: ***Adiel Tuyishime**; Francesco Basciani; Amleto Di Salle; Javier Luis Canovas Izquierdo; Ludovico Iovino:* Streamlining Workflow Automation with a Model-based Assistant. In: 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2024) [19].

MeSS 2022: ***Adiel Tuyishime**; Javier Luis Cánovas Izquierdo; Maria Teresa Rossi; Martina De Sanctis:* Modeling Linked Open Data. In: International workshop on MDE for Smart IoT Systems (MeSS'22) (2022) [20].

ICSA 2023: *Samira Silva; **Adiel Tuyishime**; Tiziano Santilli; Ludovico Iovino; Patrizio Pelliccione:* Quality Metrics in Software Architecture. In: IEEE 20th International Conference on Software Architecture (ICSA 2023) [21].

1.7 Tools and Demo

Table 1.1 provides references to the repositories containing all the developed artifacts, as well as links to video demonstrations.

Tools and Demos		
Tools and Resources	**Description**	**Link**
Middleware	Git repo: Implementation of the approach. Provisioning of REST APIs for model management	https://github.com/tuadiel6/DynamicEMF-REST-API/blob/main/Scenario.md
RESTful services for model management	A video demonstration tool and its applications	https://www.youtube.com/watch?v=HM98oJly7x8
Interoperability between EMF and WATs	Git repo: Bridging Workflow Automation Tools (WATs) and EMF modeling ecosystem	https://github.com/gssi/emf_workflow_project/tree/main
Interoperability between EMF and WATs	Video demo: Bridging Workflow Automation Tools (WATs) and EMF modeling ecosystem	https://www.youtube.com/watch?v=C0NdMI4h0Eg&t=1s
Quality metrics	Git repo: Quality metrics in software architecture	https://github.com/tiziasan/QAandMetricsForArch
Quality metrics framework	Video demo: MDE framework for maintaining a catalog of quality metrics in software architecture	https://www.youtube.com/watch?v=JRMdxSpEjEg

Table 1.1: List of resources associated with the developed artifacts

1.8 Thesis Outline

This thesis is organized into eight chapters, structured as follows:

- Chapter 2 provides the foundational concepts and terminologies relevant to this dissertation. It introduces key principles of MDE, model management, REST APIs, and the role of low-code development within the MDE ecosystem.
- Chapter 3 reviews the current state of the art in model management. We examine existing model repositories, web-based modeling approaches, scalability, and interoperability challenges in MDE. The chapter identifies the limitations of current solutions and derives the research challenges that drive the work presented in this thesis.

- Chapter 4 presents the core contribution of this thesis, which is a middleware solution for the dynamic provisioning of REST APIs to support model management. It details the architecture, design principles, and implementation details of the proposed approach.
- Chapter 5 demonstrates the effectiveness of the proposed middleware solution through a scenario-based case study. This chapter showcases how the middleware facilitates the automated generation of REST APIs and supports the manipulation of models via a web interface.
- Chapter 6 focuses on scalability, presenting a comprehensive performance analysis of the proposed approach introduced in Chapter 4. It highlights the ability of our approach to efficiently handle models of varying sizes and complexities, showcasing its scalability capabilities.
- Chapter 7 explores how the middleware solution enables interoperability between MDE ecosystems and workflow automation tools (WATs). It covers integration strategies, use case, and experimental validation, demonstrating how the proposed solution facilitates seamless model exchange between MDE ecosystems and WATs.
- Finally, Chapter 8 summarizes the key contributions of this dissertation, highlighting how the proposed middleware approach addresses the identified research questions. It also outlines potential future directions to advance model management.

Chapter 2

Background

This chapter introduces the key concepts and technologies that form the foundation of this thesis. Section 2.1 provides an overview of key concepts in Model-Driven Engineering (MDE). Section 2.2 discusses Domain-Specific Languages(DSLs), while Section 2.3 covers the Eclipse Modeling Framework(EMF). In Section 2.4 essential tasks in model management, and the role of model repositories are explored. Section 2.5 explores the foundational principles of REST APIs. Finally, Section 2.6 examines Low-Code Development and its relationship with MDE.

2.1 Model-Driven Engineering (MDE)

MDE is built upon the concept of abstraction, an approach long used in computing to simplify programming by focusing on high-level design rather than low-level details. Early programming languages like Assembly and FORTRAN employed abstraction to simplify the process of writing code by allowing developers to work at a higher level of abstraction compared to direct machine code [7].

In software development, abstraction is used to hide the complex aspect of reality, in which the representation of this reality is expressed using models. These models represent the essential features of a system while omitting unnecessary details, making the problem domain easier to understand for both technical and non-technical stakeholders [22]. The abstraction-based approach is central to MDE, a paradigm where models are considered as "first-class citizens" throughout the entire software development lifecycle [1]. These models are subsequently refined to generate specific views of the system, create platform-dependent software artifacts, and contribute to a final refinement step that produces (part of) the application code. MDE has demonstrated its effectiveness as

a powerful systems engineering approach across various architectural domains. Numerous studies have highlighted its advantages, particularly in productivity and maintainability when compared to traditional development methods [23]. Consequently, MDE is being increasingly embraced as a valuable software development methodology by leading companies, including Sony Ericsson and Thales [24, 25].

The MDE methodology is based on three core concepts: *metamodels*, *models*, and *model transformations*. (i) A metamodel defines the set of concepts, relationships, and semantic rules regulating how models can be denoted in a particular language definition. (ii) A model is an instance that conforms to a given metamodel, adhering to all the rules and structure specified by that metamodel. (iii) A model transformation is a modeling operation that automatically generates one or more output models from one or more input models, based on a transformation specification. Model transformations are defined at the metamodel level and are executed by transformation engines on models that conform to these metamodels. To address domain-specific requirements, MDE commonly uses Domain-Specific Modeling Languages (DSMLs), along with transformation engines and generators [26]. DSMLs are tailored to specific domains, allowing for more precise modeling, while transformation engines and generators are used to analyze manipulate, and generate various aspects of models.

Models and metamodels are organized across multiple levels, often referred to as the metamodeling stack, which are interconnected by the conformance relationship. In the MDE approach, this stack typically consists of three levels, with the topmost level being the meta-metamodel, a self-reflective metamodel that defines the concepts and rules governing all metamodels. The MetaObject Facility (MOF) [27], proposed by the OMG, serves as the standard architecture for meta-modeling. MOF is built around a set of modeling standards, including OCL [28], which specifies constraints on MOF-based models, and XML Metadata Interchange (XMI) [29], which enables the storage and interchange of MOF-based models in XML format. In the following we detail these levels.

2.1.1 Multi-level Metamodeling

A metamodel serves as the formal specification governing how models are constructed and how they relate to one another, offering multiple layers of abstraction to ensure consistency. Multi-level metamodeling architecture [30] is commonly used to describe the relationships between models, metamodels, and meta-metamodels. This architecture defines an instance-of (or conformance) relationship, which exists between a model and its metamodel, as well as between a metamodel and its meta-metamodel. Essentially,

the elements within a (meta)model are instances of, and conform to, the metaclasses defined in its respective (meta)metamodel. Essentially, this layered approach organizes the abstraction levels of modeling, ensuring consistency and coherence across all levels. The *M0 layer* consists of real-world objects or instances, the *M1 layer* contains models representing those instances, the *M2 layer* holds the metamodel, which defines the structure of the models at M1, and the *M3 layer* is the meta-metamodel, a framework that defines the foundational constructs for creating metamodels. Each layer conforms to the one above it, ensuring consistency and structure across different levels of abstraction in model-driven processes. Given that, we say that a model conforms to a metamodel if it is well-formed regarding the constraints and types in its metamodel.

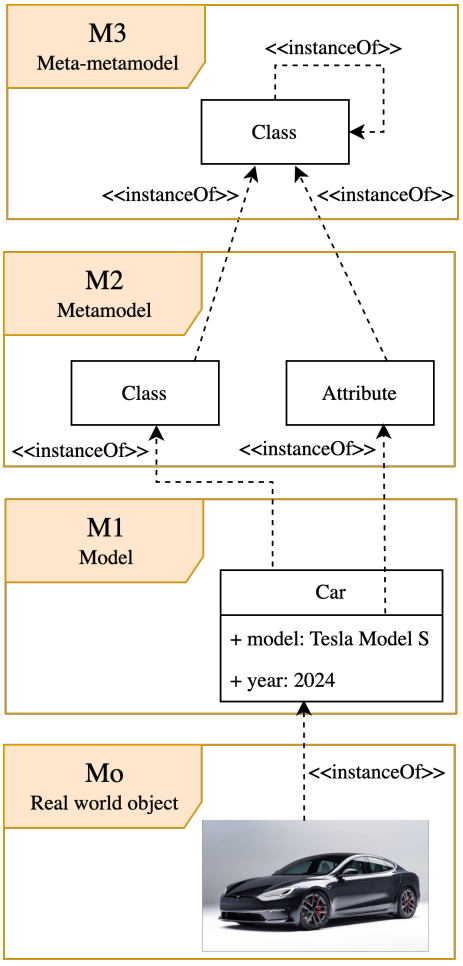

FIGURE 2.1: Abstraction Layers with Example (Inspired by [1])

Figure 2.1 illustrates the hierarchy of model abstraction with a *car* as an example. At the M0 layer, the real-world object, such as a Tesla Model S, represents the tangible instance. At the M1 layer, this object is abstracted into a model with attributes like *model* name and *year*. The M1 model conforms to the M2 metamodel, which defines the structure and rules for creating car models, such as specifying classes (e.g., Car) and their properties (e.g., model, year). Finally, at the M3 layer, the meta-metamodel provides

the foundational concepts (e.g., Class, Attribute) which are used to define metamodels at the M2 layer.

2.2 Domain-Specific Languages

A Domain-Specific Language (DSL) is a language that is tailored to address the needs of a specific domain, unlike a General-Purpose Language (GPL). The Unified Modeling Language (UML) is a widely adopted general-purpose modeling language that has become the de facto standard for modeling software systems [31–33]. While UML has demonstrated its efficiency in software modeling, several studies have highlighted its shortcomings [31–36]. To reduce the complexity of software development, the use of DSLs, which are more closely aligned with specific problem domains, have shown to be more effective than general-purpose programming languages [37].

Typically, DSLs are created using metamodeling, where a DSL is defined by a metamodel, and its instances are models. By capturing the design patterns and terminologies common to a domain, DSLs simplify the process of creating models tailored to that domain. A model, in this context, represents an abstraction of a system, developed with a specific goal in mind [38]. DSLs eliminate unnecessary general notations, focusing instead on domain-specific concepts, which significantly reduces the amount of code developers need to write compared to GPLs [39]. Applications built with DSLs are typically more concise, easier to maintain, and faster to develop [40], thanks to their high level of abstraction. Additionally, they often support code generation, allowing developers to translate models into executable code in any programming language of their choice.

A DSL is composed of three primary elements: abstract syntax, concrete syntax, and semantics. The abstract syntax describes the set of language concepts and their relationships, along with the rules to combine them. The concrete syntax defines the DSL's notation, which can be textual, graphical, or a combination of both. The semantics describe the behavior of the DSL and are often implemented through a translator (compiler) or an interpreter. In the context of MDE, the metamodel representing the abstract syntax serves as the foundation of a DSL, with its notation and semantics built around it [41].

Several techniques have been proposed for the implementation of both textual DSLs [42, 43] and graphical DSLs [44, 45]. Textual DSLs use text-based syntax and are often similar to programming languages, allowing users to express domain concepts through

code-like structures. Graphical DSLs, on the other hand, provide visual representations where users interact with diagrams and models to define domain concepts.

2.3 Eclipse Modeling Framework (EMF)

Modeling frameworks are platforms designed for the development of technical solutions to create, manipulate, and store (meta) models. They typically offer a low-level programming interface for interacting with models, as well as high-level graphical tools that simplify the creation of (meta) model definitions, constraint specifications, and creation of model elements. Additionally, modern modeling frameworks often include a code generator that produces software artifacts based on a given model, which can then be easily integrated into client applications.

Over the past decade, EMF [46] has become the standard framework for building Domain-Specific Languages (DSLs) and modeling tools within the Eclipse ecosystem. The widespread adoption of EMF is reflected in the large number of EMF-based tools available on the Eclipse marketplace [47], which come from both industry and academia.

EMF [46] is a foundational open-source toolset for MDE, widely recognized as the primary standard for modeling within the Java ecosystem. EMF was Initially developed by IBM in 2002 and began as an internal solution to meet IBM's need for defining and managing domain models. It quickly grew in functionality and scope, eventually becoming an open-source project under the Eclipse Foundation [48]. EMF is a framework that facilitates the definition and instantiation of metamodels, and forms the backbone of the Eclipse modeling ecosystem, supporting a rich set of tools and features for domain-specific language (DSL) engineering and model management.

At the core of EMF is **Ecore**, a metamodeling language that allows users to define domain-specific models. Ecore allows developers to specify the structure, data types, relationships, and constraints of models in an abstract, reusable format. Once defined in Ecore, models can be instantiated and used as first-class citizens. Ecore models serve as a foundation for code generation and other model-driven operations, making EMF a powerful tool for developers in the MDE world. Figure 2.2 illustrates a high-level overview of the Ecore metamodeling language. Metamodel elements are defined using EClasses, which include EReferences to other metamodel elements and EAttributes to specify attributes that instances of that class can contain. Model elements are derived from the EModelElement class. They can contain EAttributes to store details about the element and can have EReferences to connect with other model elements.

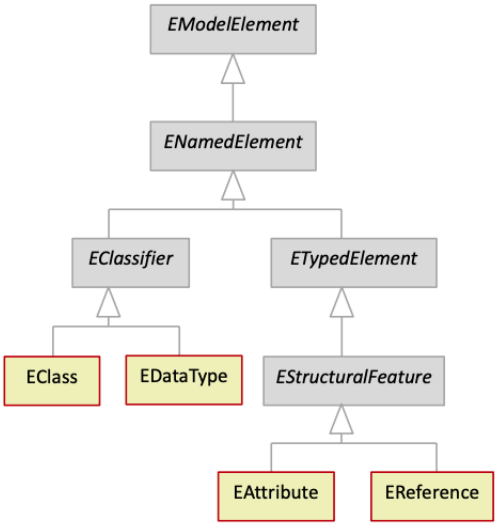

FIGURE 2.2: Simplified Diagram of the Ecore Metamodeling Language [2]

By default, models in EMF are stored in a standard XML-based format known as XML Metadata Interchange (XMI), an OMG-standardized format designed to enhance tool interoperability [29]. The EMF promotes extensibility and reuse, making it a versatile foundation for developing DSLs, executing model transformations, and generating code. EMF is often regarded as the defacto standard in the MDE community due to its adaptability, the extensive ecosystem of plugins, and its ability to support a broad spectrum of model-driven development activities. This combination of features makes EMF an essential asset in MDE, equipping developers with the capabilities needed to manage complex models effectively throughout the software development life cycle.

2.3.1 Eclipse Epsilon

Epsilon is a platform for developing task-specific model management languages. Eclipse Epsilon [49, 50] is a framework of interoperable, Java-based languages and tools designed to streamline model-based software engineering tasks. As a mature and well-established family of model management languages and tools, it provides support to a wide range of model management tasks such as transformation, merging, comparison, and validation, among others. What sets apart Epsilon in its architecture is the fact that all of its languages are built on a core language, offering the benefit of an extensible framework.

At the core of Epsilon is the Epsilon Object Language (EOL) [51], a scripting language with model-querying capabilities of the Object Constraint Language (OCL) [28]. While EOL can be used as a standalone language for programmatic model management, its purpose is to be re-used by task-specific languages. This is achieved through grammar-inheritance and the extensible architecture of the EOL execution engine.

The Epsilon framework offers a suite of specialized languages to support a range of model management tasks.

(i) The *Epsilon Transformation Language (ETL)* [52], a rule-based model transformation language consists of transform-rules that can translate elements of the source models into elements in the target models. (ii) The *Epsilon Comparison Language (ECL)* [53] is a rule-based language tailored to the task of model comparison. It consists of match-rules that evaluate and compare pairs of elements from the input models. (iii) The *Epsilon Merging Language (EML)* [54] merging models from the same or different metamodels and technologies. It extends ECL match rules and ETL transform rules by introducing merge rules, allowing developers to define structured and comprehensive merging specifications. (iv) *The Epsilon Validation Language (EVL)* [55] is designed for validating models against custom constraints and rules. It allows developers to define constraints that check model integrity and provides support for generating user-friendly messages and automated fixes when violations are detected. (v) *The Epsilon Generation Language (EGL)* [56] is tailored for generating textual artifacts, such as code or documentation, from models. It enables developers to create templates that dynamically produce output based on model elements, facilitating customizable and automated code generation. (vi) *Flock* [57] is a model migration language designed to update models as their metamodels evolve. It provides rules to specify how elements in a source model should be migrated to a new version, ensuring consistency and preserving key data during the migration process.

These languages are underpinned by a model connectivity layer called Epsilon Model Connectivity (EMC) [58], which shields them from the specifics of individual modeling technologies. EMC enables a uniform way of interacting with models, allowing operations such as transforming an EMF model into Simulink, or cross-validating an XML document and a UML model, regardless of their underlying technologies.

2.4 Model Management

In MDE, models serve as central artifacts representing various aspects of a system, ranging from high-level designs to detailed implementations. They play a pivotal role in capturing the structure, behavior, and requirements of software systems throughout the development lifecycle. Given the centrality of models in MDE, effective model management is critical. Model management encompasses a set of tasks and techniques dedicated to creating, transforming, analyzing, and manipulating models to achieve specific objectives in software development [1]. These tasks are typically defined at the metamodel level of abstraction and are applied directly to the models [59].

Efficient model management is crucial for several key reasons:

1. *Complexity handling*: As systems become more complex, so do their corresponding models. Model management techniques provide structured approaches for analyzing, transforming, and maintaining models, ensuring that even complex systems remain manageable, consistent, and easier to understand.

2. *Consistency and integrity*: Models often undergo numerous changes and transformations throughout the software development lifecycle. Model management techniques help maintain consistency and integrity across these changes, ensuring that models accurately reflect the intended system architecture and behavior.

3. *Automation and efficiency*: Model management supports the automation of repetitive tasks, such as code generation, model migration, etc. By automating these processes, it enhances productivity throughout the development lifecycle, reduces manual effort, and minimizes the likelihood of human error.

4. *Model evolution and adaptation*: Software systems evolve over time, and their models must adapt correspondingly. Model management includes operations for refactoring and migrating models, enabling existing models to accommodate new requirements or technological advancements without starting from scratch.

5. *Analysis and validation*: Model management techniques facilitate the analysis and validation of models, helping to identify potential issues early in the development process. This ensures that models meet the specified requirements and constraints, leading to the creation of higher-quality software.

2.4.1 Model Management Tasks

Having explored the foundational concepts of modeling and the role of models as key artifacts, we now examine how these models are used throughout the development lifecycle. Model management tasks encompass a variety of tasks that programmatically handle models to process, evolve, analyze, and leverage them effectively. These tasks include:

(i) **Model Transformation**: This process involves transforming one model into another of common or diverse metamodels. It plays a crucial role in automating tasks such as code generation, model refinement, and enabling interoperability between tools. Transformations are defined by sets of rules or mappings, typically specified at the metamodel level. Model transformations can be classified as: *Model-to-Model (M2M)* which transforms a source model into a target model (e.g., UML

to BPMN), or *Model-to-Text (M2T)* which generates textual artifacts like code or documentation from models.

Model transformations can be automated using scripts written in specialized transformation languages, which fall into three main categories.

Declarative Model Transformation Languages: These languages focus on describing what needs to be achieved rather than specifying how to achieve it. They focus on a high-level abstraction of the problem domain, making the code more concise and easier to understand. An example of a declarative language is QVT (Query/View/Transformation) [60].

Imperative Model Transformation Languages: In these languages, the programmer defines the exact sequence of steps or operations required to complete a task. The emphasis is on providing detailed algorithms and control flows that the computer should execute to perform a task. Examples include ETL (Epsilon Transformation Language) [52].

Hybrid Model Transformation Languages: These languages combine the strengths of both declarative and imperative approaches. They offer high-level constructs for abstraction while maintaining the flexibility of imperative programming. ATL (Atlas Transformation Language) [61] is an example of a hybrid language.

(ii) **Model Validation**: This task ensures that a model complies with its metamodel and adheres to the specified rules, constraints, and semantics. The goal is to ensure that the model is correct, consistent, and complete for its intended use. Metamodeling languages usually focus on supporting basic modeling constraints, like cardinality constraints, which define the relationships between different modeling elements. While metamodeling languages typically support basic constraints, such as cardinality, they may not fully capture the complexities needed for a comprehensive modeling language definition. Model validation languages address this gap by providing additional constraints. Examples include the OCL [28], which defines well-formedness rules that models must satisfy, enabling more precise and comprehensive specifications beyond the metamodel's capabilities. Similarly, the Epsilon Validation Language (EVL) [62], offers advanced validation features for models.

(iii) **Model Querying**: This task involves retrieving specific information or subsets of a model based on predefined criteria. It is commonly used during model transformations, defining constraints, or extracting relevant information. While general-purpose programming languages allow developers to navigate models programmatically, several domain-specific declarative languages, such as OCL [28], EMF

Query [63], and EMF-IncQuery [64], are designed to enhance model querying.

(iv) **Model Evolution and Refactoring**: These tasks focus on adapting models to meet changing requirements while maintaining consistency and quality. Evolution involves modifying models to reflect new or updated features, ensuring they remain relevant throughout the lifecycle [65]. Refactoring aims to improve the structure of the model while preserving its internal quality characteristics [66].

(v) **Model Comparison and Merging**: These tasks involve analyzing and combining multiple models. Model comparison identifies differences and similarities between models, aiding in tracking changes or ensuring consistency [67]. Model merging combines multiple models into one, resolving conflicts where models diverge [1].

To support these tasks, various tools and languages have been developed. Table 2.1 provides an overview of the most common tasks in model management, along with prominent languages and tools that support these activities. It is important to note that this table presents only a selection of notable languages and tools, representing a subset of the broader landscape supporting these tasks.

Model Management Task	**Description**	**Languages & Tools**
Model Transformation	Transform one model into another, either within the same or a different metamodel.	ATL (Atlas Transformation Language) [68], QVT (Query/View/-Transformation) [60], ETL (Epsilon Transformation Language) [52], Acceleo [69]
Model Validation	Ensures that a model complies with its metamodel constraints and rules.	OCL (Object Constraint Language) [28], EVL (Epsilon Validation Language) [55]
Model Querying	Retrieves specific information from a model.	OCL [28], EOL (Epsilon Object Language) [51], EMF Query [63]
Model Comparison	Identifies differences between two or more models.	EMF Compare [70], Epsilon Comparison Language (ECL) [53], SiDiff [71]
Model Co-Evolution	The process of adapting models to align with changes in their associated metamodels or related artifacts.	EMF Compare [70], Epsilon Flock [72]
Model Versioning	Tracks and manages different model versions.	EMF Store [73], CDO [74], Epsilon Merging Language [75]

TABLE 2.1: Common model management tasks, their supporting languages and tools

2.4.2 Model Persistence

Model persistence is a fundamental aspect of MDE processes: input models are retrieved from an existing source and loaded into memory, where they can be navigated and modified through model queries and transformations. These models are then stored in a specific format, enabling later access by client applications or shared among modelers.

Over the past few decades, the increasing adoption of MDE techniques in industry [23, 24] has highlighted the pressing need for persistence solutions capable of addressing scalability challenges in storing, querying, and transforming large, complex models. Initially, modeling frameworks were developed to handle simple modeling tasks and typically relied on XML serialization for model storage, a solution that proved insufficient as the complexity and scale of models grew.

Model persistence can be categorized into two main types: *file-based* and *repository-based*. File-based persistence stores models as individual files, while repository-based persistence involves a remote storage solution accessible by users and tools. Repositories typically rely on databases and offer additional features such as transactions and versioning to enhance model management.

File-based persistence

File-based persistence refers to storing models as standalone files, typically on local or shared file systems. This approach is straightforward and versatile, allowing models to be saved in various formats such as XML, JSON, or proprietary model-specific formats. File-based persistence makes it easy to share, version, and transfer models across different tools and environments, offering developers a flexible solution for model storage.

A widely adopted format for file-based persistence is the XML Metadata Interchange (XMI) [29]. Since the introduction of the XMI standard, file-based XML serialization has become the preferred format for storing and sharing models and metamodels. This preference aligns with early modeling frameworks, such as EMF, which were designed to handle human-produced models of relatively modest size. In these frameworks, models are serialized and stored as XML files, supporting a full-load approach where the entire model must be parsed into memory to be loaded or saved. This method is adequate for small models but becomes impractical when handling large, complex models due to memory constraints.

The limitations of XML-based serialization become evident as MDE practices gain traction in industry [24], and generative approaches such as Model-Driven Reverse Engineering (MDRE) [76] lead to the creation of large, automatically generated models. XML-based serialization has two major drawbacks.

1. **Compactness vs. Readability**: XML sacrifices compactness for human readability, which reduces the efficiency of I/O operations.

2. **Memory Limitations**: To navigate or manipulate models, XML files must be fully loaded into memory. This increases memory consumption and restricts scalability, making it difficult to load large models or enable partial loading of model elements.

Furthermore, XMI-based persistence lacks advanced features such as transactions and collaborative editing, and the use of large monolithic files complicates integration with version control systems [77]. These limitations highlight the need for more dynamic persistence solutions that can manage memory efficiently, support partial loading and incremental saving of models, and facilitate better collaboration. This necessity has led to the development of repository-based persistence solutions that address these challenges by leveraging database technologies for scalable and efficient model management [78].

Repository-based persistence

Repository-based persistence offers a robust solution for addressing the limitations of file-based persistence in Model-Driven Engineering (MDE). Instead of storing models as standalone files, repository-based approaches leverage centralized databases—either relational (RDBMS) or NoSQL—to efficiently store, query, and manipulate large models. This method facilitates concurrent access, collaboration, and advanced features such as transactions and versioning, making it suitable for modern, scalable, and distributed development environments [79, 79].

In repository-based persistence, models are serialized into an on-disk representation via model-to-database mappings. A key feature of this approach is lazy loading, which reduces memory consumption by loading only the model elements that are accessed, rather than the entire model. This enables efficient handling of large and complex models that would otherwise be impractical to manage in memory.

Several mature and widely adopted repositories support repository-based persistence.

The Connected Data Objects (CDO) model repository [74] was one of the first solutions designed to handle large models using a client-server architecture. A CDO application connects to a CDO server through a specialized interface, offering a dedicated implementation of the EMF API for seamless model manipulation. CDO employs a lazy-loading mechanism to efficiently manage memory consumption by loading only the required parts of a model. It supports essential features such as transactions, access control policies, and concurrent editing, enabling a collaborative modeling environment. The default implementation uses a relational database connector to serialize models into SQL-compatible databases. Although CDO's modular architecture allows extensions to

other storage solutions, in practice, only relational database connectors are consistently maintained and supported.

Morsa [79] was the first model repository designed to leverage the scalability benefits of NoSQL document databases for storing and accessing large models efficiently. Like CDO, Morsa employs a lazy-loading mechanism to reduce memory usage and supports incremental updates. The framework is built on MongoDB, using the document hierarchy features of the database to represent model elements and their relationships. Morsa integrates with standard EMF mechanisms for creating and accessing models transparently.

EMFStore [80] is a model repository designed for collaborative modeling, using a client-server infrastructure to store and manage models. It provides a default XMI-based implementation for handling standard EMF models and also includes a MongoDB connector for managing larger models. EMFStore focuses on collaborative features, offering git-like version control with support for model versioning, branching, and history tracking.

2.5 REST APIs

As discussed in the previous sections on model persistence and repository-based solutions, modern Model-Driven Engineering practices increasingly rely on scalable and interoperable solutions to manage models effectively. To achieve seamless communication between different systems, tools, and devices in distributed environments, Web Application Programming Interfaces (APIs) have emerged as a critical enabling technology [81, 82]. APIs facilitate integration by allowing different applications to communicate and interact while abstracting the underlying complexity. This capability aligns with the need for dynamic, scalable, and interoperable model management solutions.

A Web API operates within a client-server architecture, where the client sends a request, and the server processes it and returns a response. This interaction requires a standardized method of communication between client and server. Modern web APIs predominantly follow the REpresentational State Transfer (REST) architectural style [83] resulting in what are known as RESTful web APIs. These APIs are designed to be simple, scalable, and stateless, making them particularly suitable for internet-scale distributed systems.

REST, introduced by Roy Fielding in 2000 [83], outlines a set of principles and constraints for building scalable web services. RESTful APIs use URIs (Uniform Resource Identifiers) to identify resources and leverage standard HTTP methods such as GET,

POST, PUT, and DELETE to perform operations on these resources [84]. This approach provides several advantages, including simplicity, flexibility, and broad compatibility with various platforms and devices.

In the context of MDE, RESTful APIs support the dynamic of model management services, such as model retrieval, transformation, validation, and querying. By adopting RESTful web APIs, MDE platforms can achieve improved scalability and interoperability, enabling distributed model repositories to expose their functionalities as-web-services. This ensures that models can be efficiently managed, manipulated, and shared across heterogeneous environments, further promoting the adoption of MDE practices in modern software development.

2.5.1 HTTP Methods

REST APIs support the development of web applications by enabling CRUD (Create, Retrieve, Update, Delete) operations on resources. Following REST principles, each type of operation uses a specific HTTP method, ensuring clear, predictable interactions with the server. The table below provides an overview of the common HTTP methods used in RESTful APIs [84].

HTTP Method	Description
GET	Retrieves data from the server. Commonly used for reading or fetching resource data.
POST	Sends data to the server to create a new resource.
PUT	Update an existing resource on the server.
PATCH	Partially update an existing resource by modifying only specific fields.
DELETE	Remove a specified resource from the server.
OPTIONS	Returns the supported HTTP methods for a specific resource or server.
HEAD	Similar to GET but only retrieves the headers, not the body, of a resource.
CONNECT	Establish a tunnel to the server identified by the target resource.
TRACE	Perform a message loop-back test along the path to the target resource.

Table 2.2: HTTP Methods in REST APIs as defined by the RFC 9110 standard [84]

In a RESTful API, data is transmitted over the web via HTTP (Hypertext Transfer Protocol), and URIs are used to identify resources. Data is typically exchanged in either XML (Extensible Markup Language) or JSON (JavaScript Object Notation) formats.

While XML allows developers to define their own tags, JSON has become the preferred format due to its simplicity, ease of use with JavaScript, and human-readability [85].

A REST API call generally consists of four key parameters: *Endpoint*, *Method*, *Header*, and *Body*.

- The *Endpoint* refers to the URI that routes the client's request to the appropriate server resource.
- The *Method* defines the action to be performed, typically one of the HTTP verbs: GET, POST, PUT, or DELETE, which correspond to retrieving, creating, updating, and deleting resources.
- The *Header* contains metadata about the request, such as the format of the data (e.g., JSON or XML).
- The *Body*, also known as the message, holds the data the client wants to send to the server.

Figure 2.3 illustrates the structure of a REST API request, showing key components such as the Endpoint, HTTP Method, Headers, and Body. In this example, a client sends a JSON-formatted review to the server using the POST method, including details like the review title, description, and rating, while using headers to specify content type and authorization credentials.

FIGURE 2.3: An example of REST API call [3]

The flexibility and simplicity of REST, along with its wide compatibility with modern web technologies, have made it the standard for building scalable, efficient Web APIs.

REST enables interoperability between a client and server through a well-defined architectural style. These fundamental characteristics of REST set the foundation for its widespread adoption in web services, which are further defined by six key design principles that ensure its scalability and efficiency [83].

Below, we outline the six key design principles, also referred to as architectural constraints, that define a RESTful API.

1. *Client-Server*: This principle defines the separation between the client and the server. The client sends a request, the server processes it, performs the necessary actions, and returns a response. This separation allows the client and server to evolve independently.

2. *Stateless*: REST requires that each communication between the client and server is stateless. This means every request from the client must contain all the information needed for the server to process it, without relying on any stored context on the server. Session state, if needed, is maintained entirely on the client side.

3. *Cache*: The caching principle allows servers to define whether responses are cacheable. If a response is designated as cacheable, the client can store and reuse it for future equivalent requests, reducing unnecessary server load and improving performance.

4. *Uniform Interface*: This constraint ensures a consistent, standardized way of interacting with resources. All clients and servers must use a uniform interface, defined by a limited set of well-understood methods, to manipulate resources.

5. *Layered System*: REST allows a layered architecture, meaning a client can communicate with an intermediate server, which in turn forwards the request to the appropriate server. The client doesn't need to be aware of the underlying infrastructure, promoting scalability and flexibility.

6. *Code-on-Demand (optional)*: While optional, this constraint allows servers to enhance a client's functionality at runtime by sending executable code, such as scripts or applets, that can be executed on the client side.

2.5.2 Documenting REST APIs

REST APIs are commonly described using languages such as the OpenAPI Specification (OAS) [86], originally created as a part of the Swagger tool suite [87], or the REST API Modeling Language (RAML) [88]. These specifications offer a structured way to define a REST API, enabling both humans and machines to discover and understand a service's

capabilities without needing its source code or additional documentation. For instance, once an API is defined in an OAS document, this specification can be leveraged to automatically generate documentation, client and server code, or even foundational test cases.

The creation and formalization of OAS were driven by the OpenAPI Initiative (OAI) [89], a consortium established by leading API industry players. The primary goal of OAI is to maintain a neutral, portable, and open definition for describing REST APIs. A prominent example of the success of this initiative is Swagger[1], a widely adopted framework for designing, building, and documenting REST APIs. Swagger allows developers to define endpoints, request parameters, and response structures using the OpenAPI Specification (OAS) format, which serves as a machine-readable contract for client-server interactions. One of the key strengths of Swagger is its extensive toolset, which supports the generation of interactive documentation. This interactive documentation allows users to test API endpoints directly within the documentation interface, enhancing usability and reducing the learning curve. Additionally, Swagger provides features for auto-generating client libraries and server stubs in multiple programming languages, minimizing manual coding effort and accelerating integration.

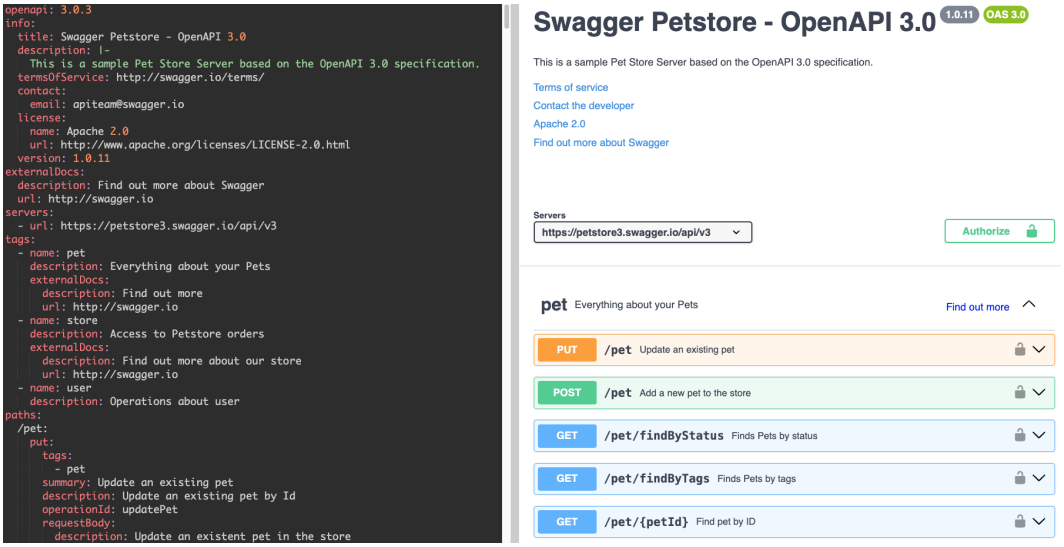

FIGURE 2.4: An example of API documentation [4]

Figure 2.4 illustrates a sample API documentation for the *Swagger Petstore* API, which follows the OpenAPI 3.0 specification. On the left, an excerpt from the OpenAPI specification provides metadata such as the title, description, license, and version information. This specification also defines the API structure, including tags for grouping endpoints and paths that outline individual operations. On the right, Swagger's interactive documentation displays a list of available endpoints for the *Pet* resource, featuring methods

[1] https://swagger.io/

like PUT, POST, GET, and DELETE, each corresponding to actions such as updating, adding, retrieving, or deleting pet records.

REST APIs play a crucial role in achieving interoperability within MDE, enabling model-based tools and repositories to communicate seamlessly with external systems and services. By providing a standardized interface for accessing, retrieving, and manipulating model data, REST APIs facilitate interactions among diverse, independent applications that require model repository access. This standardization makes model repositories more accessible and flexible, allowing exchange of models across model-based tools and platforms. Furthermore, REST APIs support synchronization across distributed systems, which is essential for maintaining consistency in collaborative and concurrent modeling environments. They enable repositories to act as centralized sources of model data that external applications can dynamically access and update.

2.6 Low-Code Development and MDE

The rapid shifts in market conditions and the continually evolving IT landscape have created an urgent need for fast and cost-effective solutions to meet rising software demands. Compounding this challenge, recruiting developers has become increasingly difficult and expensive [90]. In response, low-code development platforms (LCDPs) have emerged, designed to reduce reliance on traditional hand-coding practices [91].

The term "low-code" was first introduced in a 2014 Forrester Research report [92], which formally defined and explored these platforms as tools that minimize the need for extensive hand-coding in application development. In recent years, LCDPs have gained considerable traction, both in the market and within academia. These platforms are engineered to accelerate software delivery by minimizing coding efforts, empowering users with little or no programming experience to develop business applications [92]. Low-code development enables both technical and non-technical users to quickly create applications through visual interfaces and pre-built components. By leveraging drag-and-drop functionality, these platforms significantly simplify the development process, reducing the complexity typically associated with traditional coding. Another important feature provided by LCDP is interoperability support with a broad spectrum of external services and data sources [93].

Many LCDPs include connectors that use REST APIs, which are widely recognized for facilitating data exchange across systems on the web. These APIs act as a bridge between low-code applications and external systems, enabling data access, manipulation,

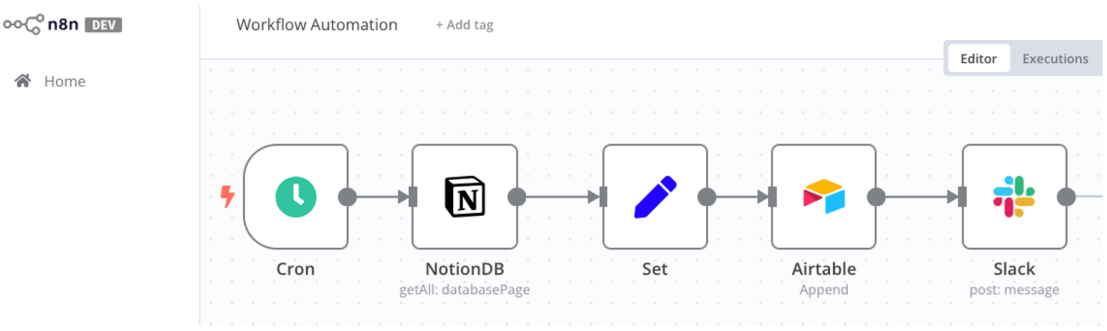

FIGURE 2.5: Business logic specification with N8N [5]

and synchronization without the need for complex backend development. This flexibility allows users to automate workflows and incorporate various enterprise services directly into their applications, reducing both the time and expertise required to build powerful, integrated solutions. By supporting REST API-driven integration, LCDPs empower both technical and non-technical users to create applications that are fully embedded within their organization's digital ecosystem, supporting a unified, adaptable environment that can quickly respond to evolving business needs.

Figure 2.5 showcases an example of LCDP called n8n [5] which integrates external services by using drag-and-drop facilities. This workflow demonstrates how n8n automates the exchange of data between external services such as Notion [94], Airtable [95], and Slack [96] by leveraging REST APIs. In this example, the NotionDB node retrieves data, which is then processed by Airtable to generate a report. Finally, the report is sent to the team via Slack. This highlights how LCDP enables seamless integration and automation of enterprise services, simplifying the creation of powerful workflows with minimal technical effort.

MDE is often compared to the low-code movement, as both share the common goal of improving software development by raising the level of abstraction and leveraging automation [93, 97]. However, they differ in their methodologies, target users, and areas of strength. Low-code platforms emphasize rapid development through visual interfaces and minimal hand-coding, making application creation accessible to citizen developers and those with limited technical expertise. This approach excels in delivering speed, iterative prototyping, and cloud-based integration, enabling quick feedback loops and deployment. However, low-code platforms face limitations when it comes to developing complex, large-scale, or highly customized enterprise systems due to constraints in flexibility, scalability, and potential vendor lock-in.

On the other hand, MDE relies on rigorous, formalized models as primary development artifacts, using DSLs to achieve precise, intentional representations of systems. This approach supports automated code generation, model verification, and transformation,

making it ideal for developing complex, enterprise-grade applications that require high levels of consistency, maintainability, and scalability. While MDE provides powerful tools for ensuring quality and correctness, it comes with a steep learning curve, complex tooling requirements, and slower initial prototyping compared to low-code platforms.

Despite these differences, low-code and MDE can complement each other effectively. Each discipline brings unique strengths to this shared objective, offering substantial opportunities for cross-pollination [93]. MDE can infuse low-code platforms with the precision and rigor required for complex systems, while low-code can bring accessibility and speed to MDE practices. This convergence paves the way for more efficient, scalable, and user-friendly development environments that leverage the best of both worlds.

2.7 Conclusion

In this chapter, we introduced the foundational concepts and technologies that underpin this dissertation. We began with an overview of the key principles in Model-Driven Engineering (MDE), establishing the central role of models in modern software development. We explored the core components of MDE techniques, including models, metamodels, and model transformations. This was followed by a discussion on model management, focusing on essential tasks and the significance of model repositories in facilitating efficient storage, retrieval, and collaborative modeling. Additionally, we examined the principles of REST APIs and their crucial role in enabling interoperability and scalability for model management in distributed web environments. Finally, we compared low-code movement with MDE, highlighting their unique strengths and the potential for cross-pollination between these two approaches.

These foundational concepts and technologies set the stage for addressing the challenges and solutions explored in this dissertation. The insights gained here provide the necessary background to understand the solution proposed in subsequent chapters, which aims to advance model management in distributed, web-based environments through dynamic RESTful services, scalability, and interoperability.

Chapter 3

State of the Art

In MDE, models are essential artifacts for representing and automating the development of complex software systems. However, as MDE adoption grows, new challenges emerge in managing and interacting with models efficiently. This is especially true in distributed web-based environments, where models must be easily accessible and efficiently queried and exchanged across model-based tools.

In distributed web-based environments, existing approaches often fall short in addressing the scalability, flexibility, and interoperability required while managing and interacting with models effectively. Recent advances in web technologies have prompted the development of web-based modeling solutions that aim to support these requirements, yet the landscape remains fragmented. Each solution typically introduces its own repository structure, and query language, making it challenging for users who require a domain-agnostic, interoperable solution that can unify model management without imposing restrictive dependencies. This fragmentation underscores the need for more generic solutions that can handle a range of model management tasks, including storing, querying, and manipulating models, while being flexible enough to work across diverse modeling platforms. Furthermore, such solutions must bridge the gaps between different platforms that rely on models, enabling seamless model reuse, improving accessibility, and fostering collaboration across different modeling environments.

This chapter explores the state of the art in model management, examining existing solutions and approaches in the field. The rest of the chapter is structured as follows. Section 3.1 explores repository-based solutions, while Section 3.2 investigates scalability-focused approaches. Section 3.3 examines web-based solutions, and Section 3.4 explores interoperability-centric approaches. Section 3.5, summarizes the limitations of current model management solutions and discusses the research gaps motivating this dissertation. Section 3.6 presents key research challenges in model management, followed

by Section 3.7 which defines the research scope of this thesis, and finally, Section 3.8 concludes the chapter.

3.1 Repository-Based Model Management

Model repositories serve as essential infrastructure for applying MDE in practical settings [1]. Over the years, significant research has focused on enabling efficient model persistence in repositories, ranging from traditional file-based solutions like XMI serialization [98] to database-driven approaches, including relational and non-relational (NoSQL) databases [2].

By default, models in EMF are stored in XML Metadata Interchange (XMI), where models are represented in single XMI files. Although XMI is designed as a standard for model interchange, it does not scale well for large models [2]. To address this limitation, relational database-based solutions such as Connected Data Objects (CDO) [74] and Teneo-Hibernate [99], and many others have been developed.These solutions leverage an Ecore metamodel to derive both a relational schema and an object-oriented API, abstracting the underlying database and allowing developers to work with models conforming to the Ecore metamodel at a high level of abstraction. Furthermore, these solutions optimize performance by supporting features like partial and on-demand loading of model elements, which reduces the memory overhead of loading entire models. Despite these advancements, relational databases struggle with the inherently interconnected nature of models. Executing complex queries often requires costly table joins, resulting in performance bottlenecks, particularly for large and highly interrelated models.

To overcome the limitations of relational databases, NoSQL solutions have gained traction for model persistence in MDE. NoSQL databases like OrientDB [100], Neo4j [101], MongoDB [102], HBase [103], and Morsa [79] offer enhanced scalability, flexibility, and performance, especially when dealing with complex, interconnected data structures. These solutions are particularly well-suited for scenarios where relational databases fall short, as they eliminate the constraints of the traditional relational model. However, NoSQL databases introduce their own challenges. For instance, the lack of a standardized query language (such as SQL) often requires custom query solutions, increasing the development burden [104]. Additionally, the diversity of NoSQL platforms such as MongoDB for document storage, Neo4j for graph databases, and HBase for columnar storage, complicates the creation of a unified approach for accessing and managing models across repositories. The distinct technologies, query languages, and data models employed by these platforms hinder interoperability and accessibility, making seamless integration across repositories a significant challenge.

In a modeling context, modelers require tools and techniques to efficiently traverse, query, and reuse modeling artifacts stored in repositories. Ideally, such tools should enable platform-agnostic interaction, allowing users to retrieve models or filter specific elements within models without concern for underlying technologies or requiring deep technical expertise. However, many existing solutions are tightly coupled to specific technologies and demand significant expertise, creating steep learning curves for non-expert users [8, 105, 106]. These platform dependency limits interoperability and restrict broader adoption.

Efforts to address model querying and discovery have led the development of approaches and tools like MDEForge-Search [14], MAR [107], MORSE [108], Moogle [109], and IncQuery [110]. These tools support the discovery and reuse of modeling artifacts in repositories. For instance, Basciani et al. [111] propose a tagging-based approach to automatically explore model repositories, simplifying artifact discovery. However, it lacks advanced search mechanisms to support more complex queries. Similarly, MDEForge-Search [14] facilitates the search and reuse of modeling artifacts stored in cloud-based repositories, further advancing artifact discovery in MDE. Moogle [109], a model search engine, uses metamodeling information to construct rich search indexes, enabling more sophisticated queries compared to traditional text-based searches. While these tools provide valuable capabilities, they often lack efficient, generic, and technology-independent mechanisms for model management and retrieval. Additionally, their reliance on specific technologies reinforces platform dependency and limits interoperability across diverse modeling ecosystems.

These limitations underscore the need for a middleware solution that bridges the gap between model repositories and external applications. Such a solution should enable platform-independent model management by offering efficient techniques for accessing, querying, and manipulating models, regardless of the underlying repository technology. A robust middleware would simplify querying, enhance model accessibility, and facilitate seamless interaction with diverse model repositories, paving the way for scalable, interoperable, and intuitive model management in distributed environments.

3.2 Scalability

In the last decades, the increasing adoption of MDE techniques in industry [23, 24] has highlighted the need for persistence solutions capable of addressing scalability challenges

in storing, querying, and transforming large and complex models. Although MDE techniques have demonstrated promising results when applied in industrial processes, studies indicate that scalability limitations in current solutions remain a significant barrier to their broader adoption [65]. This issue is particularly pronounced in generative approaches, which demand efficient techniques for managing very large models typically created within single-user contexts.

To address this issue, numerous studies have proposed solutions focusing on model storage, querying, and artifact discovery, all aimed at enhancing performance and efficiency in managing large-scale models.

Efficient storage mechanisms are fundamental for handling large models while ensuring fast access and model manipulation. Several approaches have been developed to address these needs. For instance, CDO (Connected Data Objects) [74] offers a distributed model repository that supports shared access to models in collaborative environments. As one of the earliest and most widely used solutions integrated into the Eclipse Modeling Framework, it remains a popular choice. Despite its widespread use, studies highlight performance limitations when dealing with very large models [79, 112], underlining the need for more robust storage mechanisms.

NoSQL-based approaches have emerged to address the limitations of relational databases in handling large models. One such solution is Morsa [79], which leverages the scalability features of NoSQL document databases to efficiently store and access large models. Similar to CDO, Morsa employs a lazy-loading mechanism to optimize memory usage and supports incremental updates for efficient model management. Built on MongoDB, the framework utilizes the document hierarchy capabilities of the data store to represent model elements and their associations. Morsa models can be seamlessly created and accessed transparently using standard EMF mechanisms. However, to fully exploit the performance benefits of the underlying data store, model queries must be expressed using a dedicated query language, MorsaQL [11]. Another solution is Mongo EMF [113], which facilitates storing EMF models in MongoDB. While Mongo EMF offers a standard API similar to other persistence solutions, its storage mechanism behaves differently in certain cases, such as handling collections of objects or managing bi-directional cross-document containment references. Consequently, integrating Mongo EMF into an existing system typically requires modifications to the codebase.

The Hawk framework [77] provides a model indexing solution using graph databases and offers an efficient model query API. Hawk enables users to define custom indexes to accelerate queries and improve access to elements and attributes. Although it functions as a NoSQL persistence layer, Hawk is not optimized for EMF-based queries and relies

on the Epsilon Object Language (EOL) [51] for navigating and manipulating models efficiently.

EMFStore [80] is a client-server model repository designed for collaborative modeling. It offers a default XMI implementation for handling standard EMF models and includes a MongoDB connector for managing larger models. EMFStore focuses on features such as model versioning, branching, and history tracking, adopting a git-like approach. However, scalability is not its primary concern, and its data-store configurations are not optimized for very large models.

Despite advancements in persistence frameworks that improve the handling of large models within MDE toolchains, these solutions are often tailored to specific data-store implementations. Integrating them frequently requires modifying the codebase to accommodate their advanced APIs. Additionally, many frameworks necessitate the use of specialized query languages to fully leverage the underlying database capabilities.

Several scalable querying mechanisms have been proposed. For example, IncQuery [110] facilitates scalable model queries by leveraging incremental graph search techniques and deploying the framework on cloud infrastructures. It also incorporates a domain-specific language (DSL), the Viatra Query DSL, to define queries. However, its domain-specific language presents a steep learning curve, which can hinder its usability [114]. MorsaQL [11], a query language designed for large models stored in NoSQL-based repositories, enables reflective artifact lookup through the Morsa repository [108]. However, the lack of indexing and reliance on Java-based queries limit its scalability and user-friendliness.

Despite significant advancements in model storage, querying, and artifact discovery, challenges remain in achieving seamless scalability for extremely large and complex models. Key issues include balancing performance with resource efficiency, particularly when managing extremely large and complex models. Additionally, most model query languages have a steep learning curve, requiring advanced modeling and programming expertise, which limits accessibility for non-expert users seeking to query and manipulate models.

3.3 Web-Based Modeling Approaches

The transition from traditional desktop modeling environments to web-based modeling has transformed model management in distributed and dynamic settings. This shift has

enabled collaborative, scalable, and flexible modeling practices, addressing the needs of modern distributed applications.

One notable advancement in this area is EMF-REST [15], a model-based approach that automates the generation of RESTful APIs for EMF models. EMF-REST simplifies model management by exposing endpoints that support CRUD (Create, Read, Update, Delete) operations, enabling users to interact with model instances via standard HTTP methods. While EMF-REST offers significant advantages for web-based model management, it also presents limitations, particularly its reliance on a predefined model structure for REST API generation. Since this approach requires a predefined model to generate APIs, it is less suited for dynamic, distributed modeling environments where models may not be known in advance and APIs must be provisioned on-the-fly. Additionally, if the underlying metamodel evolves, the REST API generation process must be re-executed to reflect the changes, posing challenges in environments where models frequently evolve. This lack of flexibility makes EMF-REST less suitable for dynamic platforms that require seamless adaptation and on-the-fly API provisioning.

Other model-driven approaches have explored API representation and integration. For instance, approaches like those in [115, 116], define APIs using models that represent web API definitions, enhancing the visualization and comprehension of API operations. Some of these approaches employ API metamodels to facilitate the transformation between APIs and their definitions, supporting integration with standards like OpenAPI [115]. A notable example is the API2MoL engine [117], which acts as a bridge between APIs and MDE. API2MoL allows the generation of models from APIs, simplifying the management of multiple APIs. While these approaches improve API documentation and management, they do not address the critical need for dynamic, on-the-fly generation of RESTful APIs for model management.

Another notable advancement is Sirius Web [118], which extends the Eclipse Sirius framework [119] into a web-based environment, allowing users to create and interact with graphical models directly through a browser. Initially, Sirius was developed to create customized modeling workbenches for desktop applications, primarily using Eclipse-based tools. However, with the growing need for cloud-based, accessible modeling tools, Sirius Web was created to provide similar functionalities over the web. It allows users to design and interact with graphical models directly through a browser, which enhances the accessibility of model-driven engineering in cloud settings and supports collaboration among distributed teams. Sirius Web is built on the EMF, allowing it to manage EMF-based models and offering a familiar environment for users already embedded in the Eclipse ecosystem. Through its RESTful APIs, Sirius Web integrates modeling capabilities into larger web applications, making it easier to develop interactive applications

that can programmatically manage and visualize models. Despite its advantages, Sirius Web has limitations. The development of a Web-based editor with Sirius requires a significant implementation effort which may lead to a steep learning curve. It lacks automated, on-the-fly API generation, meaning that adjustments to model structures or the addition of new models often require significant reconfiguration. This rigidity can be a drawback in dynamic environments where models frequently evolve. RESTful model management platform could ease this, leveraging familiar technologies like JSON, making models accessible even to those without specialized modeling expertise, and offering hybrid capabilities for querying across the entire model repository, rather than being confined to individual models. Furthermore, Sirius Web's emphasis on graphical representations can restrict its flexibility, as it primarily caters to visual interaction, which may not fit all model management needs.

Several research efforts [120–122] have explored model-driven techniques for generating RESTful web services and code from models. For example, MDD4REST [122] applies Domain-Driven Design (DDD) principles to create structured RESTful services through model transformations. While effective for predefined models, MDD4REST is constrained by its reliance on static model-to-code transformations, which limit adaptability in agile environments with evolving model requirements.

Other efforts extend the utility of models and model transformations for designing and semi-automatically generating Web applications, focusing on data, navigation, and presentation models [123–127]. While some approaches extend this support to web services, however, generating fully functional RESTful APIs from arbitrary domain models remains underdeveloped [128–130]. These approaches typically require the designer to outline the API using a specialized domain-specific language (DSL), leading to only a partial API generation and limited flexibility. Unlike these existing approaches, there is a need of a solution designed to generate a full RESTful API directly from any given domain model, that simplifies interactions with models, bypassing the complex technical details commonly associated with utilizing MDE models.

Other works include approaches that extend the utility of models and model transformations for designing and semi-automatically producing web applications, focusing on data, navigation, and presentation models [123–127]. However, creating RESTful APIs remains significantly underdeveloped [128–130]. These methods typically require the designer to outline the API using a specialized domain-specific language (DSL), leading to only a partial API generation. Unlike these existing approaches, our method is designed to generate a full RESTful API directly from any data model, creating a middleware that simplifies interactions with the foundational models, bypassing the complex technical details commonly associated with utilizing MDE models.

These limitations underscore a critical gap in the need for a flexible, domain-agnostic solution capable of dynamically provisioning RESTful services for any given model. Such a solution would facilitate efficient and adaptable model management in web-based environments.

3.4 Interoperability in MDE

Interoperability refers to "the ability of two or more systems to exchange information and to use the information that has been exchanged" [131]. Interoperability has long been a central challenge in MDE, particularly in distributed and heterogeneous environments where models are created and manipulated by different tools, vendors, or platforms. The diversity of modeling languages, repositories, and tools introduces technical silos that often operate in isolation, making it difficult to exchange and integrate models effectively. This lack of seamless interaction poses significant obstacles for users seeking to leverage models across varied domains and technical ecosystems.

A key challenge in achieving interoperability in MDE is the reliance on proprietary tools, languages, and different technologies, which hinder cross-platform integration. As Bézivin [132] highlights, while models act as unifying artifacts in software engineering, the fragmentation across technical spaces often undermines this potential. Each tool or system typically uses its own syntax for storing information and relies on proprietary semantics to represent and manipulate that information [1]. This diversity in syntaxes and semantics often conflicts with the expectations of other tools, complicating model exchange and creating barriers to seamless integration. As a result, the lack of standardization limits both interoperability and the broader adoption of MDE practices.

The issue of interoperability has been widely discussed in the literature [133, 134]. Early efforts focused on connecting tools through their APIs (e.g., [135]) or interfaces (e.g., [136]), often using design patterns like the facade pattern. However, these API-level approaches proved insufficient for achieving true data interoperability due to their low-level perspective. With the emergence of MDE, new approaches have shifted the focus to a higher level of abstraction, recognizing the benefits of addressing interoperability at the (meta)model level [137]. In this model-based paradigm, interoperability is defined by explicitly aligning the internal metamodels of tools, which then guide the exchange of information between them. Despite this advancement, many solutions remain ad hoc in nature [138–143]. Notable exceptions such as [144] which proposes a bus-based approach for tool interoperability, offering predefined data interchange and conversion services.

Several approaches have been proposed to address interoperability in MDE. The Object Management Group's (OMG) [29] is a notable example, emphasizing the use of standardized modeling languages such as UML [145], MOF[27], and XMI for enabling cross-platform model exchange. Tool interoperability has long been a research focus, driven by the vision of being able to plug together tools to achieve interoperability between them. An example of such efforts is the Bridging Eclipse and Microsoft Modeling Tools initiative [146], which aimed to facilitate interaction between distinct modeling ecosystems. Similarly, the Epsilon framework [52] supports cross-platform interoperability by integrating its suite of languages for tasks like model transformation, validation, querying, etc.

Despite advancements in MDE, achieving seamless interoperability between tools remains a significant challenge due to the reliance on proprietary technologies, languages, and data structures, which hinder cross-platform integration. As models become increasingly central to system development, the need for solutions that enable effortless model access and exchange across diverse platforms has become increasingly essential. These solutions should provide domain-agnostic interfaces, ensuring consistent model exchange and manipulation in distributed, web-based environments.

3.5 Comparative evaluation of model management solutions

This section provides a summary comparing existing model management solutions, evaluating them across several key dimensions essential for scalability, interoperability, and effective model management in web-based environments. The following criteria serve as the foundation for this comparative evaluation.

(i) **Storage**: Indicates whether the approach provides a structured mechanism for persisting models, ensuring that models are stored in a way that supports efficient access, retrieval, and manipulation.

(ii) **Scalability**: Assesses the ability of the approach to manage increasingly large and complex models while maintaining performance and efficiency.

(iii) **On-demand loading**: Evaluates whether the approach supports partial retrieval of model elements rather than requiring full upfront loading of the entire model.

(iv) **Non-expert user**: Examines whether non-expert users can efficiently retrieve and manipulate model elements without requiring extensive domain knowledge or technical expertise.

(v) **Web-based**: Determines whether the approach enables remote access, manipulation, and querying of models via a web interface.

(vi) **Tool interoperability**: Evaluates the ability of the approach to integrate seamlessly with various modeling tools and repositories, facilitating cross-platform model exchange.

(vii) **Modeling-as-a-Service (MaaS)**: Assesses whether the approach provides cloud-based, on-demand model management services, allowing users to create, manipulate, query, and share models without requiring local installations.

(viii) **Technological independence**: Identifies whether the approach supports a generic discovery mechanism, enabling users to retrieve and manage artifacts irrespective of the underlying modeling technology.

(ix) **REST API provisioning**: Analyzes whether the approach provides RESTful APIs on-the-fly for model management.

Based on these dimensions, we categorize and compare existing solutions into three main groups: (i) Model persistence and scalability solutions, (ii) Model discovery and querying approaches, and (iii) Web-based modeling approaches. The following tables provide a summary of the capabilities and limitations of each category, identifying key gaps that underscore the need for a scalable, interoperable, and dynamically adaptable model management solution.

3.5.1 Model persistence and scalability solutions

With the increasing adoption of MDE techniques in the industry, numerous model persistence solutions have been developed to address scalability challenges associated with storing, querying, and transforming large and complex models. Initially, many modeling frameworks relied on XMI-based serialization, which was suitable for small models but proved inefficient for large-scale applications. XMI requires full model loading, resulting in excessive memory consumption and performance bottlenecks, making it impractical for managing large models. To mitigate these limitations, alternative persistence solutions leveraging relational and NoSQL databases have been introduced. These solutions typically employ a lazy-loading mechanism to optimize memory consumption by loading only the necessary portions of a model. While these solutions enhance efficiency, they are

Model persistence and scalability solutions						
Feature	**ModelBus**	**CDO**	**Teneo-Hibernate**	**NeoEMF**	**Morsa**	**MDEForge**
Storage	✓	✓	✓	✓	✓	✓
Scalable architecture	✗	∼	✗	✓	✓	✓
On-demand model loading	✗	✓	✓	✓	✓	✓
Tool interoperability	✗	∼	∼	✓	✓	✓
Web-based	✓	✗	✗	✗	✗	✓
Model-as-a-Service	✗	✗	✗	✓	∼	✓
REST API provisioning	✗	✗	✗	✗	✗	✗

TABLE 3.1: Model persistence and scalability solutions

Legend: ✓: Supported ∼ : Partially Supported ✗: Not Supported

often tightly coupled to specific data-store structures (e.g., CDO's reliance on relational databases, NeoEMF's dependency on NoSQL backends, and Morsa's MongoDB-specific implementation), necessitating code modifications for integration into existing applications. Furthermore, some approaches require local installations (e.g., CDO, Teneo-Hibernate, and NeoEMF), meaning they must be deployed and configured within an on-premise environment, which can limit their adaptability in distributed settings.

MDEForge extends traditional model repositories by providing cloud-based model management services. It eliminates the need for local installations by enabling web-based access, allowing users to interact with modeling artifacts remotely. To support model discovery, MDEForge leverages Elasticsearch, a distributed, open-source, and highly scalable search and analytics engine, facilitating efficient querying of modeling artifacts. However, it does not natively provision REST APIs on-the-fly for model management. Instead, API endpoints must be manually configured or extended rather than being automatically generated based on the model structure.

Despite the advancements in model persistence solutions, none of these approaches provide built-in REST API provisioning for model management, preventing seamless interaction with any given domain model in a simple, automated manner. In web-based distributed environments, there is a critical need for a generic middleware solution that acts as an intermediary between model repositories, model-based tools, and external applications. Such a solution should enable seamless access to models regardless of the underlying repository technology, ensuring greater interoperability, flexibility, and ease of integration in dynamic modeling environments.

3.5.2 Model discovery and querying approaches

Model discovery and querying approaches							
Feature	**Moscript**	**EFinder**	**Hawk**	**EOL**	**IncQuey**	**EMF Query**	**OCL**
Model querying	✓	✓	✓	✓	✓	✓	✓
Non-expert user	✗	✗	✗	∼	✗	✗	✗
Cloud-based	✗	✗	✗	✗	✓	✗	✗
Model-as-a-service	✗	✗	✗	✗	✓	✗	✗
Technological independence	✗	✗	✗	✓	✗	✗	✓

TABLE 3.2: Model discovery and querying approaches

Legend: ✓: Supported ∼ : Partially Supported ✗: Not Supported

Several model discovery and querying mechanisms have been developed to enable efficient retrieval, filtering, and interaction with modeling artifacts across different repositories. Existing tools, such as IncQuery, Hawk, and Moscript, primarily rely on domain-specific languages (DSLs) for querying models. While these DSLs offer precise and powerful querying capabilities, they present significant challenges. First, their steep learning curve requires advanced modeling and programming skills, making them less accessible to non-expert users. For example, querying with EOL requires scripting knowledge. Though EOL is user-friendly than OCL, it remains oriented toward users with technical expertise, limiting accessibility for non-experts. Second, many model discovery tools lack efficiency, generality, and technology-agnosticism, restricting their adaptability across diverse modeling environments. Furthermore, most of these tools are tied Eclipse EMF. While EMF is a widely adopted standard in the modeling community, its platform dependency creates limitations, hindering broader adoption and restricting interoperability with non-EMF-based modeling solutions.

These limitations underscore the need for a technology-agnostic solution that simplifies model querying and manipulation, enhances accessibility for non-experts, and supports integration with external applications, and diverse modeling platforms.

3.5.3 Web-based modeling approaches

Web-based modeling approaches				
Feature	**EMF-REST**	**Sirius-Web**	**MDD4-REST**	**WebRatio**
Web-bassed	✓	✓	✓	✓
Modeling-as-service	✓	✓	✗	✗
Platform independence	✗	✗	✗	✓
Dynamic Provisioning of REST APIs	✗	✗	✗	✗

TABLE 3.3: Web-based modeling approaches

Legend: ✓: Supported ∼ : Partially Supported ✗: Not Supported

The comparison of web-based modeling approaches reveals that while several solutions provide essential RESTful interfaces for interacting with models, they exhibit significant limitations that hinder their applicability in dynamic, web-based model management.

Most existing approaches, including EMF-REST and MDD4REST, rely on static methods to generate REST APIs from models. As a result, they lack adaptability and are unsuitable for dynamic environments that require on-the-fly REST API generation for any given domain model. Furthermore, many of these solutions are tightly coupled to specific frameworks, limiting their applicability in heterogeneous modeling ecosystems. For example, Sirius Web is bound to Eclipse Sirius, and EMF-REST is tailored for EMF, restricting their flexibility across diverse modeling environments. Additionally, WebRatio is a low-code platform that utilizes models for specifying and semi-automatically generating web applications, it is primarily focused on business application development rather than comprehensive model management.

The lack of a generic solution capable of dynamically generating REST APIs for any domain model highlights a significant gap in existing web-based model management approaches.

3.6 Research Challenges (RC)

This section identifies and discusses key research challenges derived from the state of the art that must be addressed to enable effective model management in web-based environments. A primary focus of this research is the development of a middleware solution capable of dynamically provisioning REST APIs for any domain model. Such a

solution must support model repositories while addressing critical concerns of scalability and interoperability. These challenges are fundamental to the contributions of this dissertation and are pivotal for advancing MDE practices in distributed, web-based environment.

3.6.1 Dynamic REST API Provisioning for Model Management

Current model-driven approaches, such as EMF-REST [15], have made significant strides in supporting model management in distributed environments by automating RESTful API generation. However, these approaches exhibit notable limitations. They primarily rely on static, predefined models, meaning that REST APIs can only be generated after a model is explicitly defined. This static approach poses challenges in distributed modeling platforms, where models are dynamic, and may not be known in advance, necessitating on-the-fly API provisioning. Furthermore, as models evolve or their underlying metamodels change, existing solutions require manual regeneration of APIs, which introduces inefficiencies, delays, and potential inconsistencies in delivering up-to-date or new APIs. This inability to dynamically generate REST APIs as needed restricts the responsiveness and flexibility essential in rapidly changing web-based environments.

RC1:

The central research challenge lies in developing a middleware solution that can dynamically provision REST APIs at runtime for any given domain model. This middleware must also automatically adapt to the addition of new metamodels or changes in existing metamodels, ensuring that the APIs remain consistently up-to-date without requiring manual intervention.

To address this challenge, the research will focus on designing and implementing a middleware solution capable of provisioning REST APIs dynamically. The middleware must understand the structure of any given domain model and generate the corresponding API endpoints. This includes CRUD (Create, Read, Update, Delete) operations and potentially more advanced functionalities based on the model's structure and behavior. Additionally, the middleware must monitor model changes and automatically update the generated APIs to reflect modifications seamlessly.

This research challenge will be addressed in Chapter 4 and Chapter 5, where we detail the architecture and implementation of our proposed solution.

3.6.2 Scalability in Model Management

Scalability is a critical challenge in MDE, particularly as models become increasingly large, complex, and interconnected. Traditional techniques, such as file-based systems and relational databases, often struggle to meet the demands of modern applications, where models must be efficiently accessed, manipulated, and processed at scale. NoSQL-based approaches have emerged as alternatives, offering flexibility and the ability to manage large, interconnected models. However, these solutions can face significant challenges in supporting complex queries, particularly when traversing thousands of highly interrelated models. These scenarios often require advanced query mechanisms, which can significantly increase the retrieval or model manipulation time, leading to performance bottlenecks. There is a pressing need for solutions that can efficiently manage large, complex models. This includes ensuring fast model retrieval without causing performance bottlenecks and optimizing query mechanisms to handle large and complex models effectively.

RC2:

Developing a scalable model management solution that efficiently handles large, complex, and interconnected models, ensuring high performance and minimizing resource constraints.

To address scalability challenges, our approach focuses on dynamic REST API provisioning for model management, enabling seamless access and manipulation of large-scale models. At its core, the proposed solution should adopt a layered, stateless architecture, allowing REST APIs to dynamically handle models of varying size and complexity for efficient retrieval and manipulation. To further enhance scalability, optimized query mechanisms are integrated to abstract the complexity of model traversal, providing intuitive query endpoints capable of efficiently processing thousands of highly interconnected model elements.

Our proposal aligns with these objectives by introducing a scalable middleware solution for model querying and management, as will be presented and detailed in Chapter 4. The evaluation and comparative performance analysis will be further explored in Chapter 6, demonstrating how our approach effectively handle large-scale models.

3.6.3 Interoperability

Interoperability remains a longstanding challenge in Model-Driven Engineering (MDE), particularly in heterogeneous and distributed environments where models are created,

stored, and manipulated by diverse tools and platforms. Traditional MDE tools often rely on proprietary languages and technical spaces, creating significant barriers to integrating or exchanging models across different platforms. This limitation is magnified in distributed environments, where models are dispersed across multiple repositories and need to be efficiently accessed and manipulated. Current solutions tend to be tightly coupled to specific modeling tools or technologies, leading to vendor lock-in and a fragmented tooling ecosystem. This lack of flexibility undermines cross-platform collaboration and hampers efficient model management. To fully realize the potential of MDE, there is a critical need for solutions that enable seamless model exchange and interaction across heterogeneous environments.

RC3:

Developing flexible interoperability solutions, particularly for distributed web-based environments, to enable seamless model exchange and manipulation across diverse model-based tools, platforms, and repositories. These solutions should not be tied to proprietary technologies, ensuring broad compatibility and adaptability.

To address this challenge, we propose a bridging mechanism that enables seamless communication between model-based tools and platforms through a middleware solution. This approach enhances interoperability and facilitates bidirectional model exchange, allowing tools and platforms to interact with models without being tightly coupled to specific technologies. The details of this interoperability solution are discussed in Chapter 7.

3.7 Research Scope

This research proposes a generic middleware solution designed to be adaptable across various model management approaches, providing a unified and scalable solution for seamless model manipulation. The proposed solution enables on-the-fly REST API provisioning, allowing efficient model access and interaction within web-based environments. By dynamically exposing models through RESTful services, the middleware enhances usability, making model management accessible even to non-expert users.

The proposed solution, allows users to upload metamodels and automatically generate REST APIs for model management. This process is fully automated, enabling seamless model creation, updating, deletion, and querying through REST APIs, eliminating the need for manual intervention. Furthermore, the middleware enables interoperability by facilitating seamless model exchange across heterogeneous modeling platforms and external model-based tools through standardized RESTful services. It also enhances

scalability by integrating efficient querying and retrieval mechanisms, enabling the handling of large-scale models without compromising performance.

It is important to clarify that the scope of this research is centered on model-based management rather than metamodel-based management. The proposed infrastructure operates at the model level, ensuring that models conforming to a given metamodel can be efficiently managed, accessed, and manipulated.

3.8 Conclusion

The state-of-the-art in existing model management solutions reveals several critical limitations. One of the most pressing challenges is the lack of dynamic REST API provisioning, which forces most solutions to rely on manual API configurations, making them unsuitable for dynamic and evolving web-based environments. Scalability remains another persistent issue, as existing model persistence and querying solutions struggle to efficiently handle large-scale and complex models, particularly within web-based environments. Interoperability is also a major constraint, as many tools are tightly coupled to specific technologies, limiting their ability to seamlessly exchange models across diverse modeling ecosystems. Furthermore, most model discovery tools rely on domain-specific languages (DSLs), requiring specialized expertise, thereby limiting accessibility for non-expert users.

This thesis addresses these challenges by focusing on three key areas. (i)Dynamic API provisioning and adaptability (RC1). The need for on-the-fly REST API generation to support any given domain model is examined in RQ1 and addressed in Chapter 4 and Chapter 5. We propose a middleware solution capable of automatically generating REST APIs, enabling efficient model management in distributed web-based environments while simplifying model querying and making model retrieval and manipulation accessible to non-expert users. (ii) Scalability in managing large and complex models (RC2). The issue of scalability is investigated in RQ2, with solutions presented in Chapter 4, where we propose a scalable middleware for efficient model querying. This is further evaluated in Chapter 6, which provides a comprehensive performance analysis, demonstrating the middleware's effectiveness in handling large-scale models. (iii) Seamless interoperability across heterogeneous model-based tools and platforms (RC3). The challenge of cross-platform model exchange and integration is explored in RQ3 and addressed in Chapter 7, where we introduce a bridging mechanism to enable interoperability between diverse modeling platforms and tools, overcoming vendor lock-in and model silos.

By addressing these key challenges, this thesis aims to bridge existing gaps in model management and establish a unified, flexible, and scalable solution that supports various modeling approaches and platforms while avoiding restrictive dependencies.

The following chapters present the proposed solution in detail, outlining its scalable architecture, dynamic capabilities, and implementation, all aimed at improving the efficiency and accessibility of model management while ensuring seamless interoperability across diverse modeling tools and platforms in web-based environments.

Chapter 4

Dynamic Provisioning of REST APIs for Model Management

To address the RQ1, which explores how to develop a solution capable of dynamically provisioning REST APIs to support model management in distributed environments, this chapter presents an approach for provisioning RESTful services tailored to model management. This approach enables seamless interaction with any given domain model and can be used to build a modeling platform providing modeling-as-a-service. We implemented this approach as a middleware solution adhering to the REST principles to provide a stateless and scalable service.

The rest of this chapter is organized as follows: Section 4.1 introduces the need for a middleware solution in model management, Section 4.2 presents the proposed approach, Section 4.3 covers tool support, and finally, Section 4.4 concludes the chapter.

4.1 Introduction

Web services offer a suitable solution to provide efficient access to data storage. Among the alternatives for developing distributed services (e.g., SOAP, or WSDL), there is a growing interest in using RESTful services. REST proposes stateless distributed services and relies on simple URIs and HTTP verbs to make Web services available to clients.

Over the years, MDE has matured and been adopted in various domains, such as automotive, aerospace, and IoT. The emergence of Web technologies has led to the development of Web-based modeling tools like Sirius Web[147] and Theia[148], which offer

environments for creating and editing models, as well as model-based approaches like WebRatio[1], providing model-based low-code solutions.

A common requirement when developing Web-based modeling tools is to support model management, that is, to store, retrieve, and manipulate models. Currently, approaches offering complete RESTful services for model management are a few and still limited, with EMF-REST [15] being one of unique notable and representative example. EMF-REST provides a RESTful API for EMF models, facilitating Web-based model management. However, it requires the provision of a specific model beforehand to generate the corresponding code for the REST API, which is a limitation in distributed modeling platforms where models are not known in advance and the API must be provided on the fly. This also corresponds to being able to distribute a single application that is domain-aware, while having a domain-agnostic approach would be helpful.

In this chapter, we present an approach to dynamic provisioning RESTful services for model management, allowing REST APIs to be provided on-the-fly for any model. These services can interact with any given domain model and can be used to build a modeling platform providing modeling-as-a-service. Following REST principles, the approach ensures a stateless and scalable service, using JSON to provide data to clients. In this contribution, we implemented the approach as a middleware, which can be used to expose RESTful services for model management. We have made available an implementation of our approach on GitHub repository [149].

4.2 Proposed Approach

This section presents an approach for provisioning a REST API middleware for a distributed modeling platform. We first describe the architecture of the proposed solution, followed by the automated provisioning process for creating the REST API middleware. Finally, we detail the operations supported by the middleware.

4.2.1 Architecture

Figure 4.1 illustrates the architecture of our approach, which comprises of the following components: (i) an *MDE Repository* for storing model-based artifacts, (ii) a *Middleware* which offers a REST API to access the models stored within the *MDE repository*, (iii) a *Discovery* component responsible for detecting metamodel updates and generating Java code, and (iv) *Server applications & Client* for interacting with the models.

[1] https://www.webratio.com/

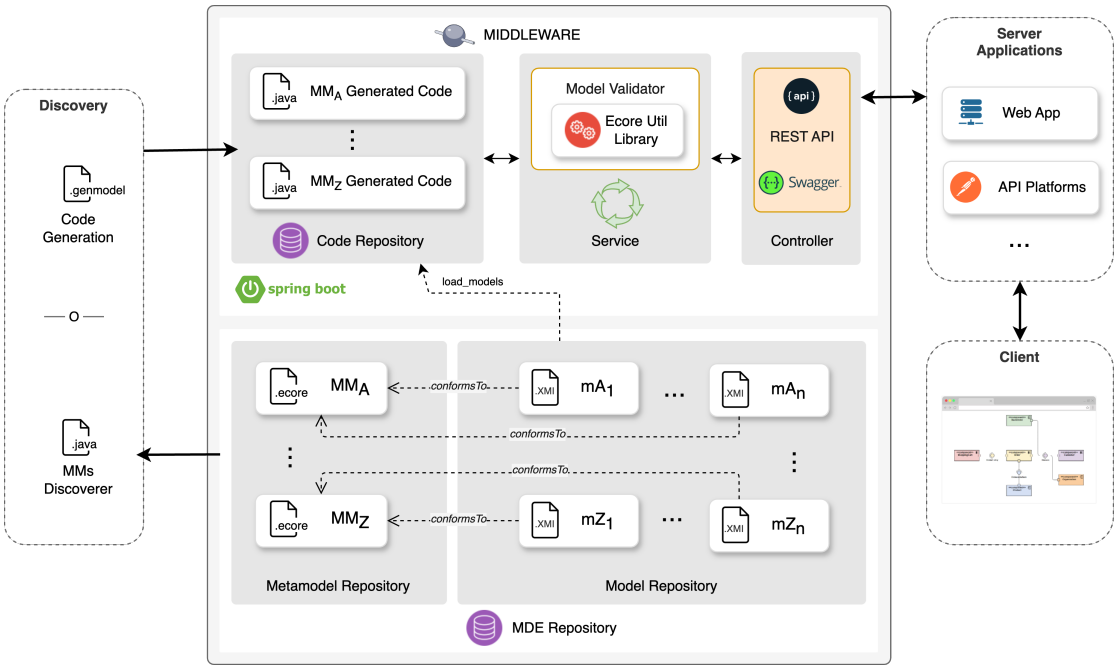

FIGURE 4.1: Architecture of our approach

(i) **MDE Repository**. This artifact stores model-based artifacts, primarily focusing on metamodels and their corresponding models of the domains interest. A metamodel can be defined as a domain description in terms of metaclasses, properties, and references; while models are specific instances of the metamodels, and comply with the structure and constraints defined in the metamodels.

Our approach is repository-agnostic, meaning it is not tightly coupled to any specific model repository. Instead, it is designed to interface with distributed repositories through REST APIs, enabling seamless access to models stored across various model-based platforms. This flexibility ensures broad adaptability, allowing the middleware to support multiple model repositories based on deployment requirements. Currently, our implementation is built on an Eclipse-based solution, utilizing the Eclipse Modeling Framework (EMF) as the core modeling technology. This choice ensures compatibility with widely used MDE tools while maintaining the ability to extend support for different repositories as needed. Although the current implementation is primarily Eclipse-based, the approach remains highly flexible and can be adapted to integrate with various model storage solutions across diverse MDE ecosystems.

(ii) **Middleware**. This artifact serves as an interface, thus offering a REST API to access the models stored within the *MDE repository*.The implemented middleware follows a three-layer architecture inspired by the Spring Boot framework [150], consisting of the code repository, service, and controller layers. Below, we describe the main responsibilities of each layer in our approach.

- **Code Repository**: Serving as the primary gateway for data access within the application, the Code Repository plays a pivotal role in interfacing with various data sources. Our approach's responsibilities extend to engaging not just with traditional databases, but also with MDE models. This layer retrieves, stores, and manipulates data, encapsulating the fundamental CRUD (i.e., Create, Read, Update, Delete) operations and translating them into model manipulation operations. It is important to note that this layer executes model operations in stateless mode, thus ensuring that each operation is self-contained and independent.

- **Service**: This layer bridges the Repository and Controller layers and encapsulates the application's core business logic. In a nutshell, model data is digested to be provided as a REST service. This layer also includes a *Model Validator* using *Ecore Util Library*, which evaluates whether the model respects the metamodel formal specification and constraints and whether it is suitable for further manipulation. If the model does not meet the necessary criteria, the *Model Validator* generates the corresponding error messages for the user.

  We have implemented a *Model Validator* that operates as a flagging system to ensure model compliance with predefined constraints in the metamodel. When a model violates a constraint, the validator flags it as incorrect and generates detailed error messages, which are returned to the client via the REST API. These messages specify the violated constraints, assisting a user in identifying and correcting inconsistencies.

- **Controller**: This layer provides the main interface for external applications. It manages incoming requests, processes them through the Service layer, and formulates the appropriate responses.

(iii) **Server Applications & Client**. The Controller layer, supported by Swagger documentation, exposes the metamodel and model elements as web services. This enables external applications, such as web apps and other API-dependent platforms, to interact with the models while shielding them from the complexities of MDE technologies.

(iv) **Discovery**. This component is responsible for generating Java code from an Ecore metamodel whenever an update occurs in the MDE Repository. The detailed process for generating the Java code is described in the subsection 4.2.2.

Operational Flow

An example of an operational flow is described as follows. The process begins at the controller layer, which handles incoming requests. The service layer processes the requests, often requiring data interactions with the code repository layer. Upon completion of the processing, the service layer relays the outcomes to the controller, which builds a response.

The proposed approach involves two different user roles: *Repository Maintainer* and the *Client User*. The *Repository Maintainer* is responsible for maintaining the repository, ensuring that it can handle the addition of new metamodels and updates to the existing ones, whereas the *Client User* interacts with the system through the external application to perform operations on the models.

4.2.2 Provisioning of REST APIs

The provisioning of REST APIs leverages on model reflection to dynamically provide REST APIs tailored to the metamodels and models stored in the repository. This automated provisioning process is triggered whenever there is an update in the Metamodel Repository. When a new metamodel is added or an existing one is updated, the process executes two steps: (1) updating the generator model (i.e., GenModel) and (2) generating the corresponding Java code. A *GenModel* is a configuration model used by EMF [123] to generate Java code from metamodels. Once the GenModel is updated, the system generates the corresponding Java code and places this code in the Code Repository. This integration ensures that the new or updated metamodel is fully integrated into our middleware without requiring redeployment or restarting the application.

To enable dynamic provisioning, model traversals are defined using generic REST API operations. For example, to retrieve all the instances of the metaclass $ClassName$ specified in the metamodel MM_A, we use the API operation $/MM_A/ClassName/all$, which dynamically traverses all models in the repository to return the required information. This configuration supports aggregated query specifications executed vertically on the entire repository, or restricted to specific models. To simplify interactions with the metamodels and enable model manipulation without requiring a deep understanding of the MDE underlying concepts, we rely on Swagger [87], a specification to document and present REST API endpoints.

It is important to clarify that the proposed approach is model-based management rather than metamodel-based management. Our approach is model-driven, meaning that the infrastructure we provide operates at the model level, ensuring that models conforming to a specific metamodel can be efficiently managed, accessed, and manipulated. The middleware is designed to enable users to upload new metamodels, upon which it automatically generates REST APIs that allow models to be created, updated, deleted, and queried accordingly. This process is entirely automated, eliminating manual intervention. When a metamodel is successfully uploaded, the middleware automatically registers it and dynamically provisions a REST API with standard CRUD operations for managing models conforming to it. Additionally, any changes to the metamodel automatically trigger the regeneration of the corresponding REST APIs to reflect the modifications.

4.2.3 Supported operations by the middleware

Our middleware supports a wide range of model manipulation operations, as detailed in Table 4.1. The *MDE Repository*, which hosts model instances, leverages EMF for fundamental CRUD (i.e., Create, Read, Update, Delete) operations. These operations are executed by leveraging the reflection API, which enables us to dynamically inspect and manipulate the properties and methods of the classes generated by the *GenModel* within the *Code Repository*.

The *Example* column in Table 4.1 provides concrete REST API usage examples for each supported operation, illustrating how model instances can be accessed, queried, modified, and deleted within the MDE Repository. For instance, a GET request to (1) *.../Family/Member/all* retrieves all family members in a family model stored in the repository, while a GET request to (2) *.../Family/Member/Parent/search?attributeName=firstName&searchValue=Stefano* filters and returns only a family where the parent's first name is *Stefano*.

Id	Operation	Description	HTTP Method	Endpoint	Example
1	READ	Read all elements from models that conform to the same metamodel	GET	.../{packageName}/{className}/all	.../Family/Member/all
2	READ	Filter an element of a specific model, filtered by a key attribute and its corresponding value	GET	.../{packageName}/{className} /{containmentName}/search? attributeName=⟨param1⟩ &attributeValue=⟨param2⟩	.../Family/Member/Parent/ search?attributeName=firstName& searchValue=Stefano
3	ADD	Add an element to a specific model	POST	.../{packageName}/{parentClass} /{childClass}/{xmlFileName} /newElement	.../Example/Member/Daughter/ FamilyModel/newElement
4	ADD	Add an element to a specific class by specifying the class with which it should be associated	POST	.../{packageName}/{className} /{xmlFileName}/addExisting? parentContainmentName=⟨param1⟩ &attributNameToMatch=⟨param2⟩ &attributValueToMatch=⟨param3⟩ &childClassName=⟨param4⟩ &childContainmentName=⟨param5⟩	.../Family/Member/ FamilyModel/addExisting? parentContainmentName= family&attributeNameToMatch= name&attributeValueToMatch= Stefanofamily&childClassName= Son&childContainmentName=sons
5	ADD	Add an element in a bidirectional relationship	POST	.../{packageName}/{parentClass} /{childClassName}/{xmlFileName} /newEopposite?fieldType=⟨param1⟩	.../Family/Member/Son/ FamilyModel/newEopposite? fieldType=Son
6	UPDATE	Update the element based on the key attribute and its corresponding value	PUT	.../{packageName}/{className} /{xmlFileName}/update? attributeName=⟨param1⟩ &attributeValue=⟨param2⟩ &updatedValue=⟨param3⟩	.../Family/Son/FamilyModel/ update?attributeName= firstName&attributeValue=Mario& updatedValue=Luca
7	DELETE	Delete the element based on the key attribute and its corresponding value	DELETE	.../{packageName}/{className} /{xmiFileName}/deleteByAttribute? attributeName=⟨param1⟩ &attributeValue=⟨param2⟩	.../Family/Son/FamilyModel/ deleteByAttribute? attributeName=firstName& attributeValue=Luca
8	DELETE	Delete an entire model by indicating the model name to be deleted	DELETE	.../{packageName}/{className} /{xmiFileName}/deleteClassByXMI	.../Family/Member/FamilyModel/ deleteClassByXMI

Table 4.1: Operations supported by our middleware

4.3 Tool Support

The development of web-based modeling tools leverages web technologies and MDE techniques to provide environments for creating and editing models or model-based low-code solutions. A critical requirement for these tools is efficient model management, which is essential in model-based system engineering. Model management involves the storage, retrieval, and manipulation of models in a fast and efficient manner. To address this need and enhance our middleware solution, we developed a platform-agnostic solution that offers RESTful services for model management. RESTful services offer stateless and scalable solutions, making them an ideal strategy for distributed services. However, existing approaches that provide comprehensive RESTful services for model management are still limited.

Our solution fills this gap by offering RESTful services that can interact with any given domain model. By providing modeling-as-a-service, it enables seamless model management in distributed environments. This tool serves as an integral part of our middleware, which provisions RESTful services for model management and facilitates interoperability across diverse platforms. The complete implementation code of our approach is available in the GitHub repository [149].

4.4 Conclusion

In this chapter, we have outlined an approach to dynamic provisioning of RESTful services for model management. Our solution enables interaction with any given domain model and can be used to build a modeling platform providing modeling-as-a-service. Furthermore, the approach relies on well-known technologies, such as EMF, Spring Boot, REST APIs, and Swagger, thus ensuring its accessibility and usability. The approach has been designed as a middleware that exposes REST APIs for distributed model management, bridging the gap between clients (e.g., web UI) and modeling infrastructure.

The next chapter presents a comprehensive evaluation of the proposed approach, focusing on its ability to automatically generate REST APIs on-the-fly for any given domain model and its applicability in real-world scenarios.

Chapter 5

Application and Evaluation of the Proposed Approach

In this chapter, we demonstrate the effectiveness of the proposed approach through a scenario-based case study. Using a Web-based modeling platform, we showcase how our middleware solution facilitates the automated generation of REST APIs and supports the manipulation of models via a web interface. We have made available an implementation and evaluation of our approach on the GitHub repository [149]. Additionally, a video demo of the entire workflow of generation and usage can be watched on https://youtu.be/HM98oJly7x8.

The remainder of the chapter is organized as follows. Section 5.1 provides the background concepts and the motivating scenario. Section 5.2 discusses the application of the approach presented in Chapter 4. Section 5.3 provides an explanation of how the approach is evaluated. Finally, Section 5.4 concludes the chapter.

5.1 Background and Motivating Example

MDE-based ecosystems support the software development process [151, 152] by using model-based artifacts built on top of metamodels, representing the abstract syntax of the application domain. Indeed, a software system is generally described using different models related to each other via model management operations (e.g., transformations).

In model-based system engineering, the system architecture, behavior, and other system aspects are designed using graphical (or textual) models representing different system views. For instance, a system can be designed as components, each connected to the others via connectors. This view can simplify the understanding of the system, especially

for stakeholder discussion [153]. A well-defined architectural model can be used not only for design-time analysis but also for generating parts of the system automatically [154]. An example of metamodel used for this aim is reported in Fig. 5.1 (a), where a simplified architectural description language is proposed. Indeed, an *ArchitecturalModel* is declared as the container of *Component*s and *Connector*s. Each metamodel concept is declared as a metaclass that can be used as a type of an instance in the corresponding model. An example of a model conforming to the metamodel is shown in Fig. 5.1 (b), where a model representing an architectural view of an IoT system is depicted. This model is screenshotted in the provided modeling editor in Eclipse EMF, which allows model manipulation via tree-based and properties views.

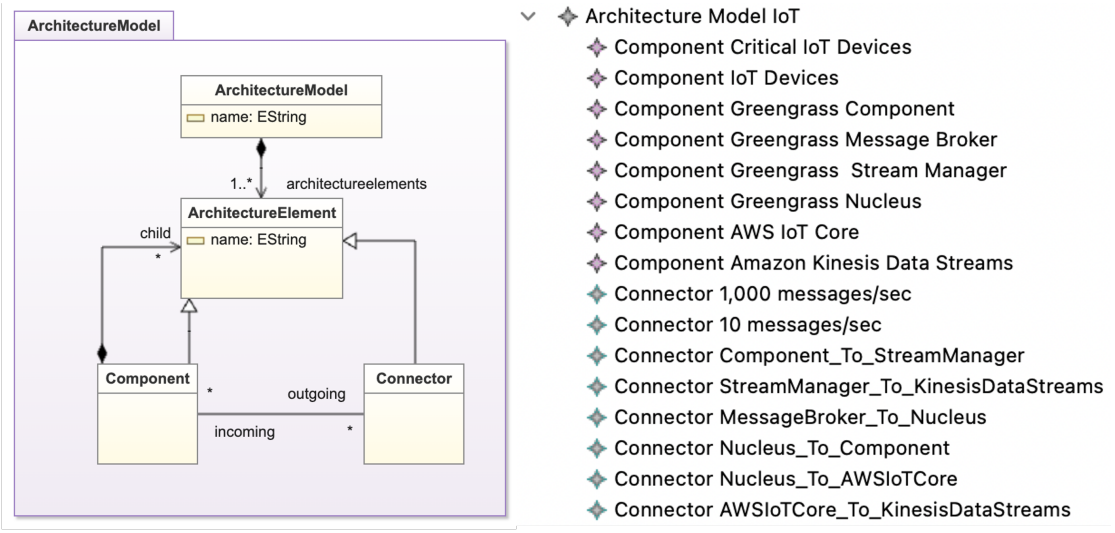

FIGURE 5.1: A model of the architecture in Fig. 5.2 and its metamodel for the architectural view of systems.

Inspired by the IoT architecture scenario by Bucaioni et al. [155], Fig. 5.2 depicts a scalable IoT architecture (based on Amazon Web Services, AWS) called *Greengrass Stream Manager* [156] to effectively handle thousands of critical messages per second from diverse IoT devices. In this scenario, *Critical IoT Devices* send thousands of messages per second to the *Greengrass Component* for processing, which then forwards them to *Greengrass Stream Manager*, and subsequently to *Amazon Kinesis Data Streams*. Less critical devices send data via the AWS IoT *Greengrass Message Broker* and *Greengrass Nucleus*. This data can either be directed to AWS IoT *Greengrass Stream Manager* at a lower priority or to *AWS IoT Core*, which then forwards it to *Amazon Kinesis Data Streams*. This IoT architecture can be easily represented with an instance of the metamodel shown in Fig. 5.1 (a), resulting in an XMI model shown in Fig. 5.1 (b).

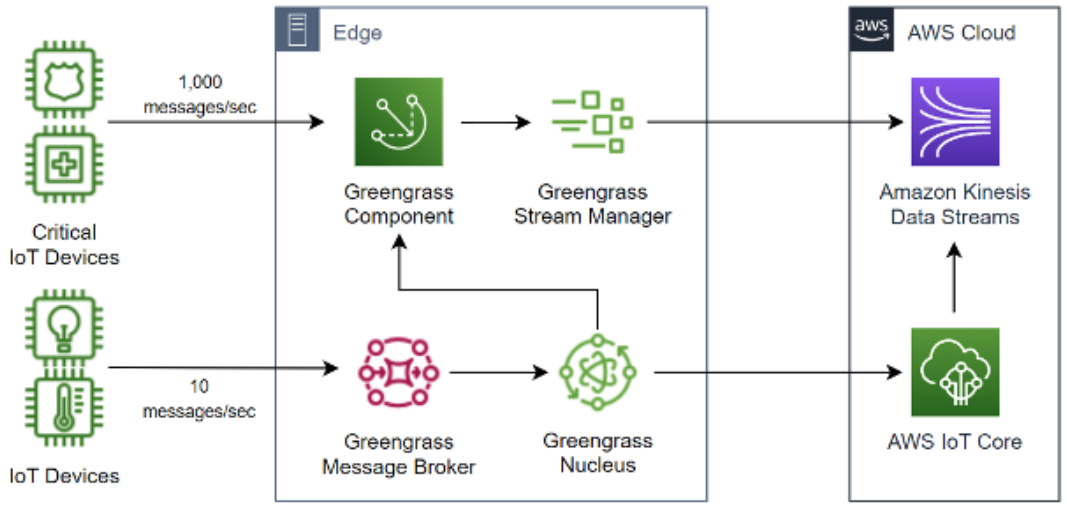

FIGURE 5.2: An example IoT Architecture proposed by Amazon.

Bucaioni et al. [157] propose a model-based approach for automatically checking the conformance of architectural models with Reference Architectures (RAs), for instance, the IoT architectural model proposed in Fig. 5.2 concerning a generic IoT reference architecture [158]. The approach includes a standalone platform where the architectural models are represented with an Architectural Description Language (ADL) inspired by UML component diagrams, similar to the one in Fig. 5.1 (a). However, distribution with a standalone platform does not contribute to model reuse, especially when this aspect is crucial as for architectural modeling. We believe that it would benefit from a re-engineering phase to provide an online toolkit with a service-based architecture. This would also enable the approach of editing models online and storing them in an online repository, enabling multiple advantages from an architectural knowledge base. This deployment poses the first question:

> *"How can we support and implement the design of the architectural models in a web-based environment?"*

Multiple modeling editors exist in the MDE panorama (e.g., Sirius [119], which also provides a web-based version). These tools are often used in a model-based environment since they are implemented on top of EMF [123]. However, these tools require advanced knowledge and skills to interact and build Web-based editors. This situation contrasts with current Web-based front-end development libraries, which offer a wide range of diagrammatical layouts and features (e.g., d3js[1], Chart.js[2] or gojs[3]), and do not

[1]https://d3js.org/
[2]https://www.chartjs.org/
[3]https://gojs.net/

require any modeling prior foundational knowledge. Currently, there is no automated way to interconnect a custom web UI, developed with standard front-end development technologies, with a modeling framework or service, as a modeling platform providing modeling-as-a-service. This is crucial for our case study and every other use case when the modeling repository is distributed, and we need to build a web interface or other service for interacting with the models. To overcome these limitations and support interaction with a distributed modeling platform, we pose the second question:

> *"Can we provide a modeling middleware that can expose REST API to enable automated distributed model management?"*

Having addressed the foundational aspects of these questions in the previous Chapter 4, the next sections demonstrate how our proposed middleware enables distributed model management by dynamically exposing REST APIs. We illustrate this approach through a motivating scenario based on an IoT Architecture shown in Figure 5.2 as a running example.

5.2 Application of the Proposed Approach

This section demonstrates an application of our approach to the IoT architecture scenario detailed in Section 5.1. A model representing this scenario, created using the EMF standalone modeling framework, is depicted in Fig. 5.1 (b).

We built a Web-based User Interface (Web UI) using libraries such as GoJS, leveraging pre-built components to create interactive diagrams and visualizations. Our middleware solution bridges the Web UI and the modeling platform, exposing RESTful services for model management. This setup allows users to manipulate models within the browser while ensuring persistent synchronization with the model repository. Interactive diagrams in the browser are generated by binding graphical objects from the Web-based libraries to our architectural model via REST API calls. For instance, to visualize our scenario in the Web UI, the middleware offers an operation for fetching components and connectors representing our model via REST API call, ensuring that all elements representing our scenario are retrieved and displayed in the browser as shown in Fig. 5.3. It is worth noting that any changes made to the model directly in the browser, whether adding, deleting, or updating elements, are automatically reflected in the *Model Repository*.

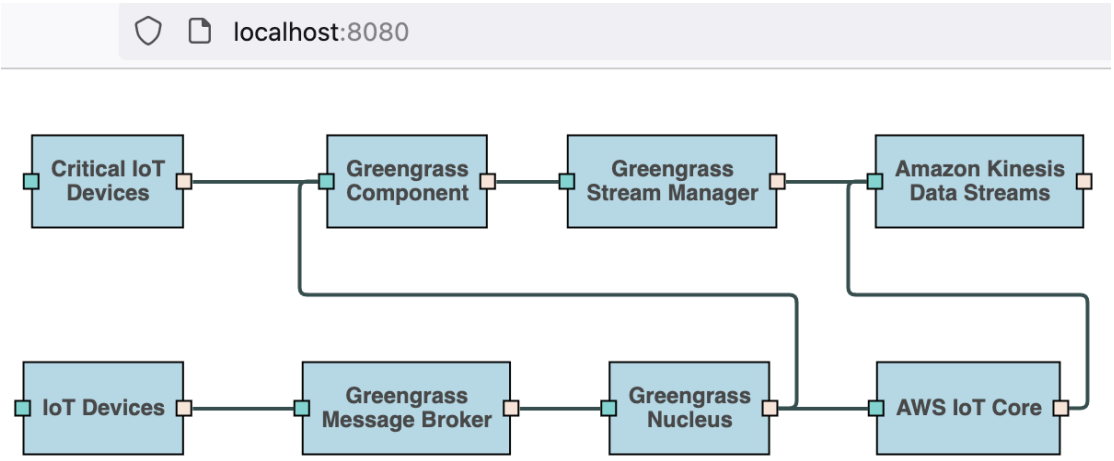

FIGURE 5.3: Web UI showing the loaded model corresponding to Fig. 5.1 (b).

The flow of a REST API call initiated by a client in the web browser is facilitated by our middleware. For example, in our running example, a client initiates a GET request to retrieve a list of *Components* and *Connectors* to populate the web-based editor. The client sends a GET request to the *Controller* (http://localhost:8080/Architecturemodeling/Component/all), as shown in Fig. 5.4 (a). The request is processed through the *Service* layer and *Repository code*, which communicates with the *MDE repository* to fetch the requested model elements. If the model is invalid, an error message is returned and can be propagated to the web interface; if valid, the list of components is sent back to the client as an HTTP response.

(a) GET all components (b) GET a specific model

FIGURE 5.4: Examples response of supported REST calls.

Our middleware supports working with multiple models simultaneously and transparently, enhancing flexibility and scalability. An operation specifically designed for filtering

particular models exists. For instance, the REST call in Fig. 5.4 (b) (http://localhost:8080/Architecturemodeling/ArchitectureModel/ArchitectureModel/search?attributeName=Name&searchValue=IoT) retrieves all components and connectors belonging to a specific model, i.e., the IoT model. This is possible thanks to the parameter *searchValue*, including the model's name to be filtered in the repository. This request will return a consistent response of elements only contained in the IoT model (see attribute name in the response body).

5.3 Evaluation

This section evaluates our approach in terms of effectiveness and coverage with regards to the requirements of the case study. The main research questions we aim to answer are:

RQ5.1: *Is the provided approach able to automate the generation of REST APIs?*

RQ5.2: *Given an application domain, is the provided approach able to support the automated model manipulation via REST APIs?*

Inspired by scenario-based testing [159], we defined a set of scenarios relevant to the running example presented in Section 5.1. Scenario-based testing is a software testing activity using pre-defined scenarios to generate test cases. Each scenario covers several steps to ensure that every system functionality is working as expected. If the functionality is supported, the test case(s) can be marked as passed. In our case, the identified scenarios check whether the generated REST APIs sufficiently cover the proposed need. Table 5.1 outlines these scenarios identified for the evaluation. We provide a category, a description of the scenario, the endpoint to test, an expected result, and an observed result for each scenario. The identified scenarios are heterogeneous, must be the most significant, and are driven by the running example to cover all the possible interactions the user may have with the editor. The details and results of this evaluation are also available in the GitHub repository [149].

5.3.1 Scenario Coverage and Rationale

The evaluation employed a scenario-based testing methodology, consisting of 10 carefully selected scenarios that reflect the primary functionalities and constraints of the proposed approach. These scenarios encompass a range of fundamental model management operations, including CRUD actions (i.e., loading, creating, updating, and deleting

Id	Category	Scenario	Endpoint	Expected Result	Observed Result
S1	Load a model	Get a complete model from the repository	.../Architecturemodeling/Component/all .../Architecturemodeling/Connector/all	A complete model consisting of both components and connectors is retrieved from the repository	The retrieved model, which includes both components and connectors is displayed in the web UI
S2	Create a component	Add a component to the existing model	.../Architecturemodeling/ ArchitectureModel/Component/ ArchitectureModel/newElement	A new component is added to the existing model *ArchitecturalModel.xmi*	The newly created component is reflected in the *ArchitecturalModel.xmi* in the repository
S3	Create a connect	Create a connector with a defined source and target elements to establish a link between components within the model	.../Architecturemodeling/ ArchitectureModel/Connector/ ArchitectureModel/newEopposite? fieldType=Component	A new connector is created via web UI and links components within the model repository	A connector is created through the web UI, and the link between components is observed on both the web UI and the model repository
S4	Update a component	Update the the name of the component	.../Architecturemodeling/Component/ ArchitectureModel/update? attributeName=name&attributeValue= Actuator&updatedValue=Sensor	Component name is updated from *Actuator* to *Sensor* via web UI	The component's name is modified via web UI and the changes are subsequently reflected in the model repository
S5	Delete an isolated component	Delete a component from the model that is not connected to others	.../Architecturemodeling/Component/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= Adaptor	A component named *Adaptor* is deleted from the model via web UI	The component is deleted from the model using the web UI and this deletion is also reflected on the model repository
S6	Delete a component	Delete the component from the model that is connected via connectors	.../Architecturemodeling/Component/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= Sensor	A component named *Sensor* is deleted from the model via web UI	The deletion of the component from the web UI leaves a pending connector with unset source or target references. A validation error is reported in the web UI, highlighting inconsistencies in the model
S7	Delete a connector	Delete the connector from the model	.../Architecturemodeling/Connector/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= Actuator_AWSIoTCore	A connector named *Actuator_AWSIoTCore* is deleted from the model via web UI	The connector is deleted from the model via web UI and the deletion is also reflected in the model repository
S8	Delete isolated components simultaneously	Delete two isolated components simultaneously by chaining API calls for components to be deleted	.../Architecturemodeling/Component/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= ABC .../Architecturemodeling/Component/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= XYZ	Components named *ABC* and *XYZ* are deleted from the model via web UI simultaneously	Two components are deleted from the model via web UI simultaneously and the deletion is also reflected in the model repository
S9	Delete components with their associated connectors simultaneously	Delete two components along with their associated connector simultaneously by chaining API calls for components and connector to be deleted	.../Architecturemodeling/Component/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= ABC .../Architecturemodeling/Connector/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= ABC_XYZ .../Architecturemodeling/Component/ ArchitectureModel/deleteByAttribute? attributeName=name&attributeValue= XYZ	Components named *ABC* and *XYZ* along with their associated connector named *ABC_XYZ* are deleted from the model via web UI simultaneously	Two components along with their associated connector are deleted from the model via web UI simultaneously and the deletion is reflected in the model repository
S10	Delete a model	Delete an entire model	.../Architecturemodeling/ ArchitectureModel/ArchitectureModel/ deleteClassByXMI	An entire model named *ArchitectureModel* is deleted via web UI	The complete model has been deleted through the web UI, and the modifications have been reflected back to the model repository

Table 5.1: Scenario-based testing to evaluate the approach

elements), and model consistency validation. Each scenario was designed to test specific endpoints, as detailed in Table 5.1, ensuring broad coverage of user interactions.

The selection of these scenarios was guided by the following criteria:

- **Representativeness**: The scenarios cover essential model management operations, ensuring that all fundamental interactions supported by the middleware are evaluated.
- **Heterogeneity**: The scenarios incorporate a diverse set of operations, ranging from basic CRUD interactions to more advanced use cases such as simultaneous operations.

- **Criticality**: The scenarios emphasize key functionalities, including constraint enforcement and validation error detection (e.g., scenarios S5 and S6), ensuring compliance with metamodel constraints, and preventing inconsistencies.

5.3.1.1 Completeness of Scenarios

The 10 scenarios were designed to comprehensively cover the typical operations normally performed when working with models, ensuring a structured evaluation of the middleware's capabilities. The following mapping demonstrates that the scenarios selected in Table 5.1 comprehensively cover the fundamental model management operations.

(i) Class-Level operations

- **Creating a class** → Creating a component (**S2**)
- **Updating a class** → Updating a component's attributes (**S4**)
- **Deleting a class** → Deleting a component (**S5**)

(ii) Reference-Level operations

- **Creating a reference (association/relationship)** → Creating a connector between components (**S3**)
- **Updating a reference** → Updating a connector (**not explicitly demonstrated in this scenarios**)
- **Deleting a reference** → Deleting a connector (**S7**)

(iii) Model-Wide operations

- **Loading a model** → Retrieving a complete model from the repository (**S1**)
- **Deleting a model** → Removing an entire model from the repository (**S10**)

(iv) Advanced operations

- **Deleting multiple classes simultaneously** → Deleting isolated components in batch operations (**S8**)
- **Deleting multiple classes along with their references** → Deleting components along with their associated connectors (**S9**)

Answer to RQ5.1. The results reported in Table 5.1, confirm that the approach can automate the generation of REST APIs (i.e., RQ5.1). As shown in Table 5.1, all scenarios yielded results consistent with the expected outcomes. The middleware successfully automated the generation of REST APIs and supported their usage for model manipulation. For instance, Scenario S1 demonstrated the capability to retrieve and load models, providing foundational support for user interactions with models stored in the repository. Scenario S6 highlighted the approach's ability to detect and report inconsistencies caused by deletions, ensuring model validity. Scenarios S8 and S9 demonstrated the middleware's capability to handle complex, simultaneous operations, highlighting its scalability and reliability.

Screenshots are also provided to show the scenarios' effects on the model before and after the REST API calls on the GitHub repository [149]. By applying the generation process described in Section 4.2, we could expose all the needed endpoints by the identified scenarios.

Answer to RQ5.2. We show, for each scenario, the instantiated endpoint used by the frontend application to populate the editor or to manipulate the modeled architecture (see *Endpoint* column in Table 5.1). For instance, scenario S1 requires to get a complete model from the repository. The editor initially loads the complete model from the repository, which involves calling two endpoints: one for loading the components and one for loading the connectors. The expected result is that the components and connectors are received as a response to the REST GET call, and the observed result is that the editor can show the loaded model corresponding to the XMI model on the repository. Scenarios S5 and S6 deal with situations that could trigger an inconsistency in the XMI model in the repository. In our running example, deleting a component could make the model inconsistent if the component is connected to others via connectors. If that is the case, the connectors will remain pending with an unset source or target, which is not allowed by the metamodel in Fig. 5.1 (a), where a connector can exist only if it is connected to a component. For this reason, the expected result is confirmed in S5, but the scenario also returns a validation error in S6, as expected. We can summarize that each scenario returned an observed result corresponding to the expected, thus addressing RQ5.2.

5.3.2 Limitations and Threats to Validity

The threats to validity in this experiment can be mentioned as internal or external. The threats to internal validity are influenced by independent variables concerning causality. Although we considered a limited number of scenarios, we tried to mitigate this threat

by heterogeneously selecting the most representative, covering all the possible endpoints we support, and interleaving them. Moreover, the scenarios have been written to cover all the possible actions provided by the editor in architectural modeling.

Threats to external validity are conditions that can limit the ability to generalize the experiment results. We used a specific running example, but we tried to mitigate this threat by differentiating the selected scenarios to cover all the possible usage of endpoints. This means we obtain the same results if we apply the same process to another case study. Moreover, the approach can be tailored to other modeling environments even though it is specific to models in the EMF framework. The approach relies on the EMF model generator, and this component provided by EMF could be replaced by another generator that supports the same generation features in another modeling environment.

5.4 Conclusion

In this chapter, we evaluated our approach to provisioning RESTful services for model management using a scenario-based methodology applied to a running example. We demonstrated the application of our approach within a web-based modeling platform and validated its effectiveness through scenario-based testing. The results confirmed that the approach is able to automate REST APIs provisioning and support efficient model manipulation. A video demo of the entire workflow of generation and usage can be watched on https://youtu.be/HM98oJly7x8.

Chapter 6

Scalability Evaluation and Experimental Results

This chapter presents a comprehensive evaluation of the approach presented in Chapter 4, focusing on its scalability. It outlines the methodology and experimental setup used to assess the performance of our approach in managing models of varying sizes and complexities. The evaluation is structured around two key dimensions: *intra-model* and *multi- model* traversals. *Intra-model* traversal refers to the capability of the approach to efficiently traverse a single model, while *multi-model* traversal assesses its ability to traverse across multiple models.

The remainder of this chapter is organized as follows: Section 6.2 outlines the steps leading to the experimental evaluations. Section 6.3 describes the dataset selection process used to assess the scalability of our approach. Section 6.4 discusses dataset preparation, validation, and model categorization. Section 6.5 details the experiment execution, covering both intra-model and multi-model traversals. Section 6.6 analyzes the obtained results, while Section 6.7 presents a comparative evaluation of our approach against existing solutions. Finally, Section 6.8 concludes the chapter.

6.1 Introduction

MDE has emerged as a transformative paradigm in software and system development, placing models at the core of the development process. This approach allows for high levels of abstraction, where executable code and other artifacts are generated directly from models, streamlining development process and reducing the likelihood of human errors. However, as MDE adoption grows, efficiently managing, querying, and manipulating

increasingly large and complex models has become a significant challenge, posing bottlenecks for model management operations. Consequently, scalable techniques for handling large models are essential to support the widespread adoption of MDE in industry and ensure its applicability in increasingly large-scale and complex scenarios [13, 78].

The scalability of model management tools and techniques is a critical concern, particularly in distributed and web-based environments where traditional tools often struggle to handle large-scale models efficiently, leading to performance bottlenecks and inefficiencies when querying, and manipulating models [2, 160]. To address these challenges, the middleware presented in Chapter 4 introduces a dynamic and scalable solution that leverages RESTful APIs for model management. This middleware is designed with a layered architecture that ensures a stateless and scalable service, capable of handling and querying large and complex models containing thousands of elements. By utilizing this middleware solution, users can seamlessly interact with models, enabling efficient retrieval, manipulation, and querying of model elements without requiring advanced expertise, which is often a prerequisite for traditional model query languages.

To evaluate the scalability of the proposed approach, this chapter presents a comprehensive set of experiments assessing its performance in handling models of varying sizes and complexities. This evaluation is guided by two dimensions: *intra-model* and *multi-model* traversals. Intra-model traversal refers to the ability of the approach to efficiently process operations within a single model, while multi-model traversal examines the middleware's capacity to traverse multiple models.

These dimensions are fundamental in assessing the capability of our solution to address scalability challenges in distributed and web-based environments. The experiments offer a comprehensive analysis of the middleware's performance, demonstrating its effectiveness in handling increasingly large and complex models while maintaining efficiency and ensuring seamless model management operations.

6.2 Experimental design

The experimental evaluation of our approach follows a structured methodology to ensure a comprehensive and reliable assessment. The process begins with the selection of datasets containing models of varying sizes and complexities, followed by model categorization. Finally, the experiments are executed by performing REST API queries across both *intra-model* and *multi-model* traversals. This evaluation assesses the capability of our approach to efficiently handle models of different scales, including large and complex models.

The evaluation process is structured in the following steps:

1. **Dataset selection:** This step outlines the selection of datasets used to evaluate the scalability of our approach and describes the experimental environment. Two distinct datasets were chosen: (i) ModelSet [161], a labeled dataset of software models, and (ii) GraBaTs dataset, a well-known set of models proposed from the Graph-Based Tools (GraBaTs) 2009 contest [162].

2. **Dataset preparation:** The *ModelSet* dataset comprises 5,120 UML models, all sourced from public repositories. A dataset-cleaning process was conducted to retain only valid models, resulting in a final selection of 1,388 valid models. The *GraBaTs* dataset, contains five large models, labeled Set0 to Set4. The smallest model (Set0) contains 70,447 model elements, while the largest (Set4) consists of 4,961,779 model elements. The selection of these two datasets serves a dual purpose: (i) to assess the capability of our approach in handling a high volume of models, for instance when querying multiple models simultaneously, and (ii) to evaluate its efficiency in processing very large models with thousands or millions of elements.

3. **Model categorization:** Models from ModelSet were categorized based on size and complexity using a quartile-based classification, dividing them into four groups: *Small*, *Medium*, *Large*, and *ExtraLarge*, according to their structural characteristics such as the number of classes, attributes, methods, and relationships. This categorization ensures a diverse range of complexities for comprehensive performance assessment.

4. **Experimental execution:** REST API queries were executed for both *intra-model* and *multi-model* traversals across different sizes of models and categories. Intra-model traversal involved retrieving elements from a single model, while multi-model traversal aggregated elements across multiple models. Two types of API calls were executed to traverse a single model and multiple models. Execution times were measured to evaluate the efficiency and scalability of our approach. Additionally, to provide a comparative evaluation, the same operations were performed using plain EMF, as a baseline comparison against our middleware.

5. **Results analysis:** The final phase involved interpreting the obtained results by analyzing the execution times of the queries and comparing performance across different model sizes and categories. Additionally, the results obtained using our middleware approach were compared with those from plain EMF.

By following this structured methodology, the evaluation ensures comprehensive testing, and the results provide valuable insights into how the approach performs under different levels of model complexity and traversal scenarios. The findings are essential for assessing the feasibility of the proposed approach for managing large-scale models in a distributed environment.

6.3 Dataset selection

Two distinct datasets were selected for the evaluation: ModelSet[161] and the GraBaTs dataset[162]. The choice of these datasets serves a dual purpose: (i) to assess the capability of our approach in handling a high volume of models when performing multi-model queries, using ModelSet, and (ii) to evaluate its efficiency in processing very large models containing thousands or even hundreds of thousands of elements within a single model, using the GraBaTs dataset.

6.3.1 GraBaTs dataset

To evaluate the efficiency of our approach in processing very large models containing thousands or even hundreds of thousands of elements, we choose the Graph-Based Tools (GraBaTs) 2009 contest dataset [162], a well-established benchmark widely referenced in the literature [2, 11, 79]. These models conform to the JDTAST metamodel, which is specifically designed for modeling Java source code. The JDTAST metamodel that defines these models is composed of three packages: the *Core package* includes metaclasses that represent logical units such as projects, packages or types; the *DOM package* includes metaclasses for representing abstract syntax trees for Java source code, e.g. compilation units, methods, etc.; finally, the *PrimitiveTypes package* includes metaclasses that represent Java primitive types such as String or Integer.

The dataset comprises five progressively larger models, labeled Set0 to Set4, with sizes ranging from 70,447 model elements in Set0 to 4,961,779 model elements in Set4. This dataset provides a robust test case for assessing the scalability of our approach when handling increasingly complex models. Table 6.1 presents a detailed breakdown of each model's size, including the number of Java classes represented (i.e., TypeDeclaration elements) and the total number of model elements contained.

Name	Java Classes	Model Elements
Set0	14	70,447
Set1	40	198,466
Set2	1,605	2,082,841
Set3	5,796	4,852,855
Set4	5,984	4,961,779

Table 6.1: GraBaTs Dataset

6.3.2 ModelSet dataset

To validate the scalability and efficiency of our approach, we conducted another experiment using a large-scale dataset derived from ModelSet [161], which is a labeled dataset of software models. This dataset comprises 10,580 models, including 5,460 Ecore metamodels and 5,120 UML models, all extracted from public sources. For this study, we focused on UML models, as they represent real-world instances serialized in the XMI standard, aligning with the objectives of this study. The Unified Modeling Language (UML), proposed by OMG [29], is a widely adopted general-purpose modeling language used in software development. It offers an EMF-based implementation and supports diverse diagram types, such as class diagrams, interaction diagrams, and use case diagrams, among many others.

The ModelSet dataset provides several features, including the domain of each model and key metrics such as the number of elements, references, classes, attributes, packages, enumerations, properties, associations, components, and data types. These metrics are essential for determining model complexity, a key factor in the categorization process used in our experiment.

Additionally, ModelSet provides a Python-based library (modelset-py)[1], which offers a convenient interface for extracting dataset contents into structured data frames. This streamlined the data preparation and exploration process, allowing us to analyze and preprocess the dataset efficiently. This experiment aims to evaluate the effectiveness of our approach in handling both intra-model and multi-model traversals.

[1]https://models-lab.github.io/blog/2021/modelset/

Environment

To carry out our experiments, we used the following environment: Our experiments were performed using the Eclipse Modeling Framework (EMF) [123], a modeling platform for implementing our approach. The evaluation was executed in an environment running Java VM 17.0, 8 GB memory, and an 8-core CPU, running on Mac OSX Sonoma. For model preparation and analysis, we used Jupyter Notebook [163], which facilitates efficient preprocessing, exploration, and categorization of the dataset.

6.4 ModelSet: Dataset preparation and model categorization

In the following subsections, we describe the preparation and categorization of the ModelSet dataset to ensure its suitability for our experiments. Since the models in ModelSet were sourced from public repositories, some could not be readily instantiated in our environment. To address this, a dataset-cleaning process was conducted to filter out non-conformant models. Once a refined dataset was established, a model categorization process was applied to classify the models based on their structural complexity.

6.4.1 Dataset preparation

The selected ModelSet dataset consists of 5,120 UML models, all sourced from public repositories and serialized in the XMI format. However, during the preparation phase, we encountered challenges with some models, which could not be instantiated in our experimental environment. These issues were primarily due to missing UML profiles, resulting in errors such as *FeatureNotFoundException* and *UnresolvedReferenceException*. These exceptions occurred because the required UML profiles were absent from the dataset, making the affected models non-conformant with the UML metamodel.

To address this issue, a dataset cleaning process was undertaken to retain only those models that could be successfully instantiated using the UML metamodel in our environment. To streamline this process, we developed an automated script to validate each model.

This script performed the following key tasks:

- **Model conformance check**: Validated the model against the rules and constraints defined by its UML metamodel.

- **Error identification**: Detected models that failed due to missing profiles or unresolved references.
- **Classification of models**: Marked models as valid or invalid based on their ability to conform to the metamodel.

A model was considered invalid if it violated the constraints defined by UML metamodel or failed to instantiate due to missing or unresolved elements.

After the cleaning process, a total of 1,388 models were identified as valid and suitable for inclusion in the experiments. These models constituted the final dataset for our evaluation. By filtering out invalid models, we ensured a reliable and consistent foundation for assessing the performance of our middleware solution, minimizing potential distortions caused by non-conformant or incomplete models.

6.4.2 Model categorization

To systematically evaluate scalability, the selected valid UML models were categorized based on their structural complexity. This classification was based on established metrics from the literature on measuring the structural complexity of UML class diagrams [164, 165]. The selected metrics offer a comprehensive assessment of model complexity and its influence on scalability, considering key structural features such as the number of classes, attributes, methods, associations, aggregations, and dependencies.

Normalization: Prior to calculating complexity scores, the dataset was normalized to ensure comparability across all features. Linear normalization was applied to scale each feature's values to a comparable range, eliminating biases caused by differing magnitudes. The normalization formula [166] used is as follows:

$$x_{\text{normalized}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Where x represents the original value, $x_{\min}$ is the minimum value, and $x_{\max}$ is the maximum value of the feature.

Each feature was normalized independently using column-based normalization. This approach preserved the relative significance of structural features such as the number of classes, relationships, attributes, and methods, ensuring that no single attribute dominated the complexity calculations.

– **Total Number of Classes (NC):** Represents the total count of classes within the model.

- **Total Number of Relationships (NR):** Includes associations, generalizations, aggregations, and dependencies.
- **Total Number of Attributes (NA):** Sum of all attributes across classes.
- **Total Number of Methods (NM):** Sum of all methods across classes.

The complexity score (C) for a model was calculated as:

$$C = NC + NR + NA + NM$$

Model categorization: After calculating the complexity scores, the models were classified into four distinct categories using a quartile-based approach: **Small, Medium, Large, and ExtraLarge**.

Small Models are models with complexity scores within the first quartile. *Medium Models* are models with complexity scores between the first and second quartiles. *Large Models* are models with complexity scores between the second and third quartiles. *ExtraLarge Models* are models with complexity scores in the top quartile.

Category	Small	Medium	Large	ExtraLarge	Total
Value	365	355	347	321	1388

TABLE 6.2: Categorization of models based on complexity

The distribution of models across these categories is presented in Table 6.2. These categories represent varying levels of complexity, with each group containing models that fall within specific quartile ranges of complexity scores. It is important to note that the number of models within each category varies due to the non-uniform distribution of complexity scores in the dataset. Although quartiles theoretically divide the data into four equal parts, the actual number of models per category depends on how the complexity scores are distributed. As a result, some categories may contain more models than others, reflecting the non-uniform distribution of complexity in the dataset.

This categorization structures the models based on their complexity, enabling a systematic evaluation of performance across different complexity levels. This approach ensures a thorough scalability assessment and provides valuable insights into how our approach manages models of varying complexity and scale.

6.5 Experiment execution

To evaluate the performance of our approach, we aim to address two key research questions:

RQ1: *How does our approach perform when performing intra-model traversals?*

This research question evaluates the efficiency of our approach in traversing a single model. It examines operations such as loading and retrieving model elements, assessing the approach's capability to effectively handle model navigation and querying within a single model.

RQ2: *How does our approach perform when performing multi-model traversals?*

This research question evaluates the performance of our approach in handling multiple models and processing a high volume of models efficiently.

6.5.1 API endpoints used in the evaluation

To evaluate the performance and scalability of our approach, REST API queries were executed to perform model operations (e.g., loading, querying, traversing) across two key dimensions: *intra-model* and *multi-model* traversals.

6.5.1.1 Intra-Model traversals

In this evaluation, a REST API query was executed within a single model to retrieve all its elements. This was applied to the GraBaTs dataset, which consists of large models containing thousands to millions of elements. The goal was to assess the middleware's efficiency in processing individual models and to analyze how performance scales with increasing model complexity.

The intra-model traversals were evaluated using the following API endpoint:

GET .../model/elements/{modelname}

This endpoint retrieves all elements within a specified model, allowing us to measure the time required to traverse models of different sizes. By systematically analyzing the response times, we examined how well the middleware performs when handling large-scale models.

6.5.1.2 Multi-Model traversals

For the multi-model traversal evaluation, REST API queries were executed across different model categories (i.e., Small, Medium, Large, and ExtraLarge), as categorized in Subsection 6.4.2. This evaluation was conducted using the ModelSet dataset, which consists of thousands of UML models sourced from public repositories. To assess scalability in this dataset, we also calculated the average traversal time for a single model within each category.

The evaluation aimed to measure the middleware's capability to process a high volume of models within each category efficiently. This is particularly important for real-world scenarios where model management tools must handle large-scale repositories containing thousands of models while ensuring efficient retrieval and processing.

The multi-model evaluation was performed using the following API endpoint:

GET .../model/elements/{directory}

This endpoint retrieves all elements from all models stored in a specified directory, enabling efficient bulk retrieval operations. By executing queries over a large dataset, we analyzed the middleware's scalability, particularly in high-load scenarios where multiple models are accessed, queried, and processed simultaneously. The insights gained from this analysis provide a comprehensive understanding of how our approach scales as the number and size of models increase.

Environment

The execution of REST APIs queries was performed using Postman[a], a widely used tool for testing APIs. Postman was employed to execute the queries and measure the performance of the middleware in both intra- and multi-model contexts. The tool provided detailed performance metrics, including execution time for each query, enabling a thorough assessment of our approach under different scenarios and varying levels of model complexity.

[a] https://www.postman.com/

6.5.2 PlainEMF implementation

To provide a baseline for comparison, we implemented custom Java code using the Eclipse Modeling Framework (EMF) for both intra-model and multi-model traversals. This PlainEMF-based solution serves to highlight the performance trade-offs between

our middleware approach and direct model manipulation through traditional Java-based methods.

Intra-Model Traversals: For intra-model traversals, we implemented Java code that loads a single XMI file into memory using EMF's ResourceSet API. The traversal process involves recursively navigating through the model's elements using EMF's EObject and EReference mechanisms. The Java code was optimized to minimize traversal time by:

- Utilizing efficient data structures, such as HashSet to avoid revisiting elements and prevent infinite loops.
- Restricting traversal to containment references to avoid unnecessary navigation of non-essential links.

The execution time was measured from the initial loading of the model until all elements were traversed and extracted.

Multi-Model Traversals: For multi-model traversals, we extended the intra-model logic to handle multiple XMI files within a specified directory. The Java program iteratively loads each model file and applies the same recursive traversal strategy used in the intra-model scenario.

Performance Measurement

The performance of the PlainEMF solution was evaluated based on the execution time required to traverse models in both intra-model and multi-model scenarios. For intra-model traversal, execution time was recorded for each individual model, whereas for multi-model traversal, the total time taken to process all models within a directory was measured. This evaluation provides insights into the efficiency of PlainEMF in handling different model traversal scenarios.

6.6 Results analysis

This section presents a detailed analysis of the results obtained from the intra-model and multi-model traversal evaluations. The findings provide insights into the performance and scalability of our approach, highlighting its efficiency in handling models of varying complexity and scale.

6.6.1 Intra-model traversals

In this evaluation, we measured the execution time required to traverse models of varying sizes across both datasets (GraBaTs and ModelSet). The goal was to assess the efficiency of our approach in handling model navigation and querying within individual models. Additionally, we compared the obtained results against PlainEMF to evaluate performance differences and analyze the impact of our middleware on execution time.

6.6.1.1 Intra-model traversal: GraBaTs dataset

Table 6.3 presents the execution time (in seconds) required to traverse models of varying sizes (Set0 to Set4) using our middleware solution and compares it against PlainEMF. The findings indicate that execution time is directly influenced by the number of model elements, with larger models requiring more processing and resulting in longer traversal durations. As model complexity increases, both approaches experience a noticeable rise in execution time, highlighting the impact of model size on performance. As model size increases from 70,447 elements (Set0) to 4,961,779 elements (Set4), both approaches exhibit a significant rise in execution time.

However, the results demonstrate a clear performance difference between the two approaches, with PlainEMF consistently outperforming our middleware in terms of raw execution speed. PlainEMF achieves significantly lower execution times, ranging from 1.63 seconds (Set0) to 89.14 seconds (Set4). In contrast, our middleware requires more processing time due to the added overhead of REST API communication and serialization, with execution times ranging from 7.93 seconds (Set0) to 403.2 seconds (Set4).

These results highlight the trade-off between a direct, in-memory approach (PlainEMF) and a REST-based middleware solution. While PlainEMF achieves faster execution times by directly accessing models in memory, it requires significant development effort to implement model queries, often demanding complex coding and a deep understanding of EMF's underlying model manipulation mechanisms. In contrast, the middleware solution, despite its additional processing overhead, provides a more accessible and user-friendly approach. By exposing models through REST APIs, it enables effortless model querying and manipulation with a single click, eliminating the need for complex implementation efforts. This ease of use makes the middleware particularly advantageous for non-expert users and applications that require seamless model access across distributed environments.

Name	Java Classes	Model Elements	PlainEMF	Middleware
Set0	14	70,447	1.63	7.93
Set1	40	198,466	3.18	11.91
Set2	1,605	2,082,841	17.22	94.2
Set3	5,796	4,852,855	66.34	327.6
Set4	5,984	4,961,779	89.14	403.2

TABLE 6.3: Intra-model traversals: GraBaTs dataset - Execution Time (in seconds)

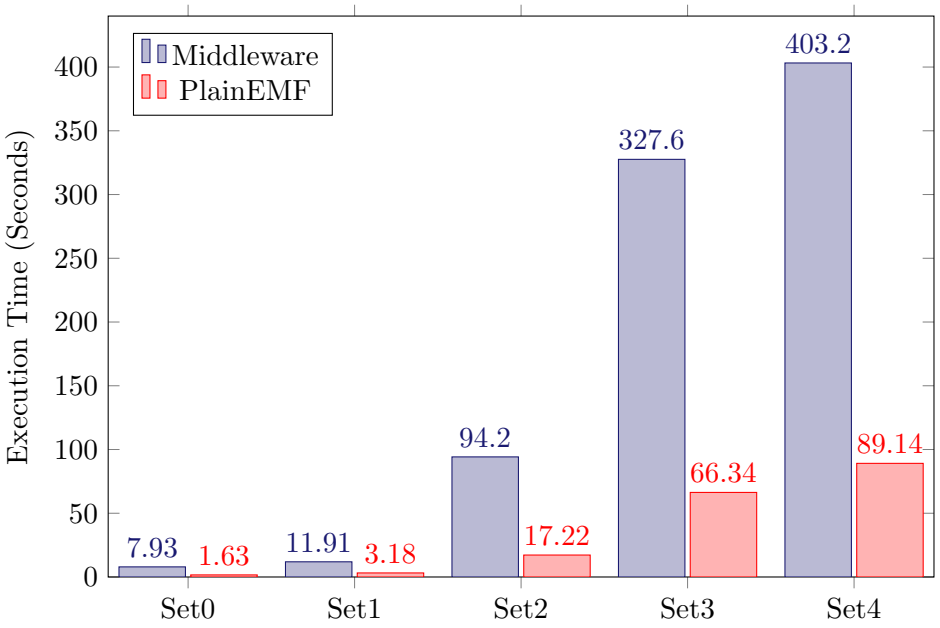

FIGURE 6.1: Execution Time Comparison - Middleware vs. PlainEMF

6.6.1.2 Intra-model traversal: ModelSet

The execution times presented in Table 6.4 illustrate the performance results for traversing a single model within each category of ModelSet. The evaluation was conducted across all models in each category, and the average execution time required to traverse a single model was calculated. As expected, execution time increases with model size. For instance, traversing a Small model using the Middleware takes an average of 1.02 seconds, whereas an ExtraLarge model requires 2.40 seconds. This increase is attributed to the growing number of model elements, which expands from 71 in Small models to 198 in ExtraLarge models.

A comparison with PlainEMF reveals that it consistently achieves faster execution times across all model categories. For example, PlainEMF completes traversal of Small models in 0.8 seconds, whereas the Middleware requires 1.02 seconds ($\sim 1.5\times$ slower).

Category	Elements	Middleware	PlainEMF
Small	71	1.02	0.8
Medium	89	1.10	0.84
Large	110	1.35	0.91
ExtraLarge	198	2.40	1.47

TABLE 6.4: ModelSet - Average execution time (in seconds) per model

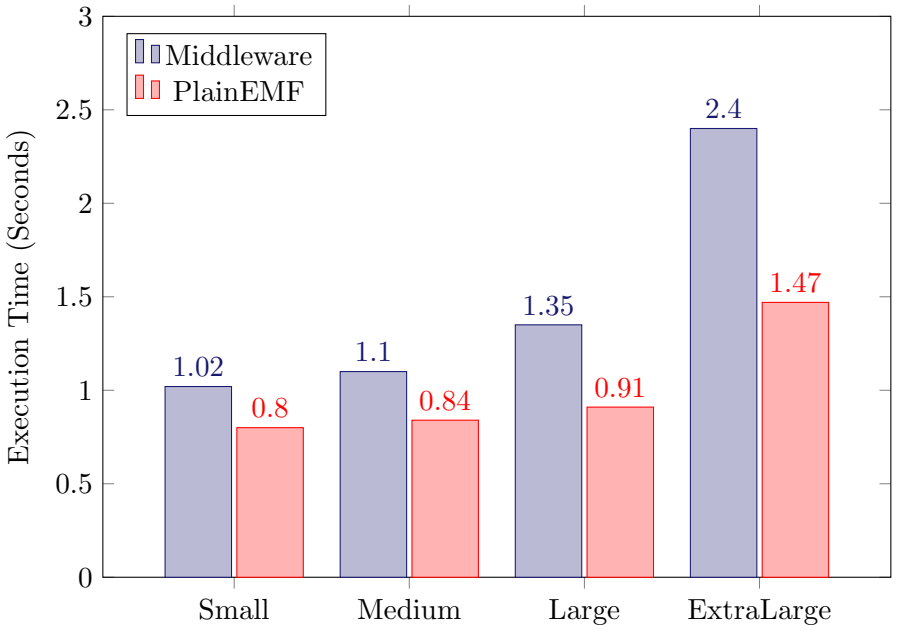

FIGURE 6.2: ModelSet - Execution Time Comparison: Middleware vs. PlainEMF

Observation

The intra-model evaluation across GraBaTs and ModelSet datasets confirms that execution time increases with model size, as larger models require more time to traverse due to their higher number of elements. GraBaTs models, being significantly larger, exhibit the highest traversal times.

While PlainEMF outperforms our approach in terms of execution speed, our middleware offers greater flexibility and ease of use by eliminating the need for manual Java code implementation. Unlike PlainEMF, where developers must write custom traversal logic, our approach enables querying via REST APIs with just a single click, making it more accessible and user-friendly.

6.6.2 Multi-model traversals

Category	Models	Classes	Attributes	Methods	Relationships	Elements	Middleware	PlainEMF
Small	365	180	3,878	128	4,401	25,815	6.20	4.78
Medium	355	1,170	4,930	15	4,464	31,740	6.51	4.99
Large	347	252	8,167	211	6,525	38,287	7.81	5.26
ExtraLarge	321	1,945	14,970	5,292	8,951	63,553	12.83	7.86
Dataset	**1,388**	**3,547**	**31,945**	**5,646**	**24,341**	**159,395**	**33.35**	**22.89**

TABLE 6.5: ModelSet - Multi-model Traversal Execution Time (in minutes)

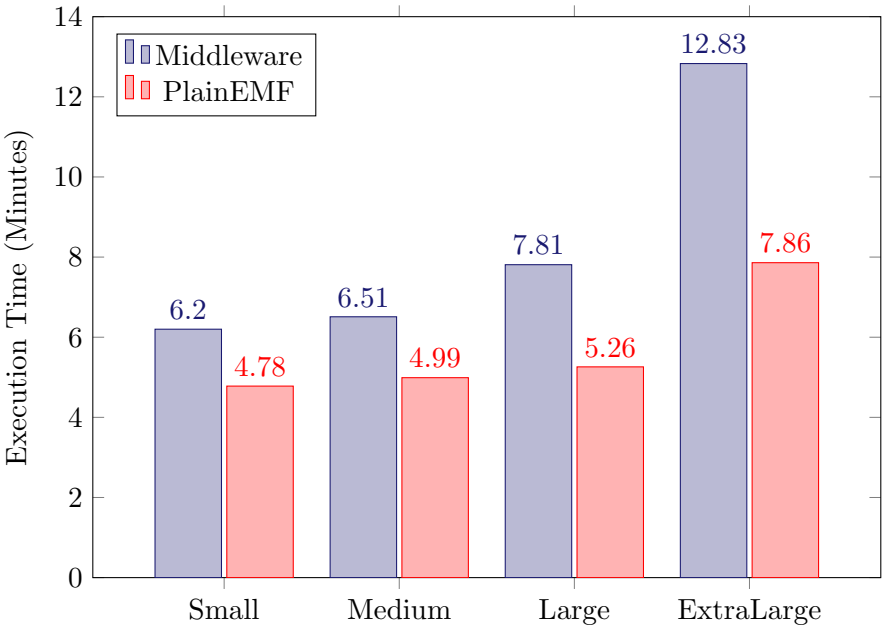

FIGURE 6.3: ModelSet - Execution Time Comparison: Middleware vs. PlainEMF

For the multi-model traversal, REST API queries were executed to process all models within each category of the ModelSet dataset (Small, Medium, Large, and ExtraLarge). Table 6.5 presents the cumulative execution time required to process all models in each category, comparing the performance of our Middleware approach against PlainEMF.

The results indicate that execution time increases with model complexity and size. For instance, processing all Small models (365 models with 25,815 elements) took 6.20 minutes using Middleware, compared to 4.78 minutes with PlainEMF. As model complexity increases in the ExtraLarge category (321 models with 63,553 elements), the cumulative execution time rises significantly to 12.83 minutes for Middleware and 7.86 minutes for PlainEMF.

Observation

Overall, the dataset-level totals highlight that Middleware took 33.35 seconds to process all 1,388 models, while PlainEMF completed the same task in 22.89 seconds. This performance gap is expected, as Middleware introduces overhead from REST API processing, REST API communication, and data serialization. Conversely, PlainEMF leverages direct in-memory access, yielding faster processing times.

6.6.3 Discussion of results: Querying with PlainEMF vs. Middleware

This section compares the results obtained from the *PlainEMF* implementation with those from our *Middleware* approach, evaluating both raw performance and the development effort required for each method. While *PlainEMF* demonstrates superior execution speed due to direct in-memory processing, it necessitates significant coding effort, particularly for complex queries. In contrast, our *Middleware* introduces processing overhead but offers a more user-friendly, scalable solution through REST APIs.

The performance evaluation reveals that model traversal using *PlainEMF* consistently outperforms the *Middleware* in terms of execution time. This is expected, as *PlainEMF* benefits from direct, in-memory model access, avoiding the REST API communication process and serialization overhead inherent in RESTful approaches. However, this speed advantage comes with notable trade-offs. Efficient model querying in *PlainEMF* requires significant development effort. Developers must manually write Java code to query models, handle references, and optimize traversal logic. This process is time-consuming and demands a high level of technical expertise, especially when dealing with intricate or large-scale models. In contrast, our *Middleware* approach abstracts these complexities, allowing users to query models seamlessly via REST APIs with just a single click. This dramatically reduces the development time and effort required, making it more accessible and user-friendly. While the execution time is slower due to the inherent overhead of API-based querying, the benefits in terms of usability, flexibility, and scalability often outweigh the raw performance cost.

In summary, while *PlainEMF* offers faster execution times, it struggles with scalability when handling large and complex models and requires significant technical expertise in EMF and Java. In contrast, the *Middleware* solution is web-based, user-friendly, and highly scalable. It enables model manipulation and querying directly through the web with just a simple click, making it easily accessible to both technical and non-technical users.

6.7 Comparison with current approaches

In this section, we position our approach (i.e., Middleware) by comparing it to the existing model discovery tools, based on the key features identified in the literature [14].

The Table 6.6 compares several features of different model discovery tools: PlainEMF [46], InQuery [110], MDEForge-Search [14], Morse [108], Moogle [109], and our approach (i.e., Middleware). The checkmarks (✓) indicate that a tool supports a feature, while the cross marks (✗) indicate that the feature is not supported.

Feature	PlainEMF	InQuery	MDEForge-Search	Morse	Moogle	EMF-REST	Middleware
Model Conformance	✓	✓	✓	✓	✓	✓	✓
Technology Independence	✗	✓	✓	✓	✓	✓	✓
Interoperability (REST API support)	✗	✗	✓	✗	✓	✓	✓
Query Mechanism	✓	✓	✓	✓	✓	✓	✓
Scalability Support	✗	✓	✓	✗	✗	✓	✓
Model-as-service	✗	✗	✓	✗	✓	✓	✓
Dynamic Provisioning of REST APIs	✗	✗	✗	✗	✗	✗	✓

TABLE 6.6: Model discovery tools - Supported Features

(i) **Model Conformance**: Model conformance ensures that models align with their respective metamodels, allowing metamodels to be reused for the creation of subsequent models. Our Middleware supports this feature, ensuring consistency and reusability.

(ii) **Technological Independence**: This feature highlights the ability of tools to discover artifacts independent of underlying technologies or data models. Our middleware supports this through its generic discovery approach, fostering flexibility and adaptability across model-based platforms.

(iii) **Interoperability**: Interoperability facilitates the integration of model management tools with third-party applications and platforms. Middleware supports REST API-based interoperability, enabling seamless communication and reuse of models across diverse model-based tools.

(iv) **Query Mechanism**: This feature reflects the tool's capability to search and retrieve modeling artifacts via established APIs. Middleware provides a dynamic and query mechanism to enhance model discovery and manipulation.

(v) **Scalability Support**: This feature ensures that the system can handle large and complex models. Middleware is designed to be scalable, efficiently managing large and complex models in distributed environments.

(vi) **Model-as-a-Service**: This feature allows model management operations to be provided as web services, enabling external tools and applications to interact with models on the web. By supporting this paradigm, the middleware facilitates seamless model manipulation, retrieval, and exchange in web-based environments.

(vii) **Dynamic Provisioning of REST APIs**: This feature enables the automatic generation of REST APIs for model manipulation, allowing for flexible and on-the-fly model management operations. Middleware dynamically generates REST APIs on-the-fly for model management operations.

In summary, Table 6.6 highlights the capability of our solution (i.e., middleware) over existing model management solutions. Unlike traditional approaches that lack dynamic REST API provisioning, our middleware enables seamless, on-the-fly model management without manual intervention. It also ensures technology independence, allowing integration across diverse model-based tools and platforms. While existing solutions provide scalability and query mechanisms, they often lack interoperability through REST APIs and Model-as-a-Service (MaaS) capabilities. In contrast, our middleware supports both, enabling efficient model access, manipulation, and exchange within web-based environments.

By addressing interoperability, scalability, and dynamic provisioning, our middleware delivers a flexible and comprehensive solution for modern model management needs.

6.8 Conclusion

This chapter presented a comprehensive evaluation of the scalability and performance of our middleware approach for model management. The experiments were conducted on two distinct datasets, GraBaTs and ModelSet, assessing the middleware's capability to handle both large, complex models and high volumes of models. The evaluation focused on two key dimensions, intra-model traversals, which involved querying individual models, and multi-model traversals, which assessed performance across multiple models.

The results have shown that PlainEMF consistently outperformed our middleware in raw execution speed due to its direct memory access. However, this performance advantage comes with a trade-off. PlainEMF requires extensive development effort, as every query must be manually implemented in Java. In contrast, our middleware eliminates the need for custom code by enabling seamless model querying via REST APIs, making model management more accessible to non-expert users. For multi-model traversals, the middleware demonstrated strong scalability, efficiently processing large volumes of

models. While PlainEMF maintained faster execution times, the performance gap was narrower in this scenario, highlighting the middleware's effectiveness in handling large volumes of models efficiently.

In the next chapter, we explore how this middleware can be extended to support interoperability with workflow automation tools, further demonstrating its flexibility and potential for broader applications.

Chapter 7

Bridging Workflow Automation Tools and EMF Modeling Ecosystems

Interoperability remains a persistent challenge in MDE, especially in heterogeneous environments where models are created and managed across diverse tools, vendors, or platforms. Traditional MDE tools often rely on proprietary languages or frameworks, complicating cross-platform model exchange and integration. This fragmentation is further amplified in distributed environments, where models are stored across multiple repositories and needed by external applications. These challenges underscore the importance of addressing RQ3, which aims to develop a solution that ensures seamless interoperability, enabling consistent exchange and manipulation of models across diverse model-based tools and platforms. REST promotes interoperability by employing a uniform interface that relies on URIs to uniquely identify resources and facilitates their manipulation through HTTP methods and standardized serialization formats. Our middleware as presented in Chapter 4, addresses these interoperability gaps by enabling external applications, such as web apps and other API-dependent platforms, to interact with models via REST APIs. By enabling models to be accessed, updated, and shared across platforms, the middleware bridges MDE and web-based ecosystems. In this chapter, we present how our approach can bridge two different technical spaces, bridging EMF ecosystem and Workflow Automation Tools (WATs).

The remainder of this chapter is organized as follows. Section 7.1 provides an overview of WATs and MDE ecosystems. Section 7.2 introduces a running example. Section 7.3 demonstrates how the running example can be addressed using the MDE ecosystem, while Section 7.4 explores its solution within WATs. Section 7.5 presents the bridging

mechanism between WATs and MDE ecosystems. Finally, Section 7.6 discusses key findings, and Section 7.7 concludes the chapter.

7.1 Introduction

A workflow specification is a detailed description of the steps, activities, and tasks that need to be completed in an automated way to achieve a specific goal. Recently, no-code Workflow Automation Tools (WATs), such as Zapier[1], IFTTT[2], NodeRed[3], N8N[4], and Integromat/Make.com[5] emerged for their cost-effectiveness by providing pre-built integrations and the possibility to create connectors to any service with REST APIs for data manipulation. They represent a successful attempt to adopt low-code development platforms (LCPDs) to create and manage workflows and integration processes in a cost-effective and streamlined way. LCPDs allow unskilled users to develop applications and workflows, even visually, without writing complex and extensive code [93].

Model-Driven Engineering [7] (MDE) is often compared to the low-code movement since both aspire to improve and further automate the software development process by raising the level of abstraction and leveraging automation [93]. However, model-driven technologies target more language designers and offer them plenty of expressiveness at the risk of not being usable for non-technical people. Di Ruscio et al. [93] highlighted that being the two disciplines conceptually close, there is substantial potential for cross-pollination.

This chapter explores data manipulation in WATs and discusses it compared with model management operations in MDE. We consider for this discussion two of the most used ecosystems in both disciplines, i.e., Eclipse Epsilon [167], as a framework for implementing pure MDE ecosystems; and N8N as one of the prominent WATs and the related ecosystem of services. This comparison outlines that they target different audiences and offer different levels of automation, complexity, and flexibility, and keeping the benefits of both approaches would be ideal. For this reason, we explore the bridge between WATs and MDE ecosystems.

[1] https://zapier.com
[2] https://ifttt.com
[3] https://nodered.org/
[4] https://n8n.io/
[5] https://www.make.com

7.2 Running example

This section describes a motivating example that will be then implemented using two different solutions, one based on MDE tools and another based on WATs, to then compare and discuss how we can combine both solutions[6].

Scenario. The departments of a university have international collaborations with professors and researchers who can be invited to visit hosting research groups (e.g., for giving a course or for invited visiting positions). A university faculty member can open an "invited guest position" on the system, where he/she must express information related to the guest position, including the period of stay, title of collaboration, hours of lectures, and course title. The remuneration can be calculated according to the department's rules and period of stay and collaboration. After this information has been set up, an invitation letter is generated and eventually sent to the invitee to specify the financial arrangement and all the information that can be useful and used by the invitee to motivate their visit. As long as we specify the course title, the dates, and an abstract, a web page can be generated and published online to announce the course or the novel collaboration. Concerning the payment, when a new guest is invited, a new payment must be emitted, and this should be managed by another system when the collaboration has been completed and confirmed.

7.3 Solving the example: the MDE version

In the following, we present the implemented model management operations to support the described scenario, implemented with a modeling ecosystem. In MDE, a modeling ecosystem [169] is composed of a large number of artifacts pursuing a common scope that are usually defined starting from a metamodel, which represents its nucleus [169]. A model management framework defines and implements model management operations, such as transformation or code generation, with the main aim of manipulating modeling artifacts [170] to pursue a common goal. Even if a *minimal* MDE infrastructure can be defined as a set of models and a transformation built on top of their metamodels pursuing a pre-defined scope, modeling ecosystems are usually more complex and articulated. Tools and languages for manipulating models are usually chained or composed to implement the modelers' goals, creating links between modeling elements belonging to different models [171].

In our case study, we use Eclipse Modeling Framework (EMF) [123] and Epsilon [167] for the modeling environment, and the ETL language to implement the model-to-model

[6] A detailed description of the scenario is provided in [168].

transformation, as well as the EGL language for model-to-code, and model-to-text transformations. In addition, model validation and the interaction with external services are done with EVL and EOL, respectively. Finally, ANT is used for concatenating the various operations. In *GuestInvitation* and *PersonRemuneration* metamodels (cf. Figures 7.1 and 7.2), all the concepts for the systems managing the guest invitation and payments are formalized, as described before.

Figure 7.1 presents the metamodel designed based on the domain specification, with an example instance that represents the guest invitation process in a university setting. Additionally, Figure 7.2 illustrates the personal remuneration metamodel with an example instance, capturing all the concepts necessary for managing payment processes within the system.

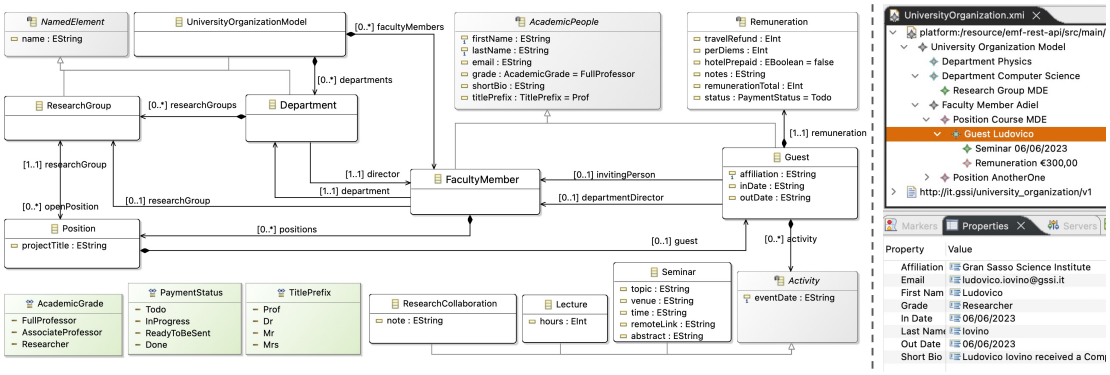

FIGURE 7.1: Guest Invitation metamodel with an instance model

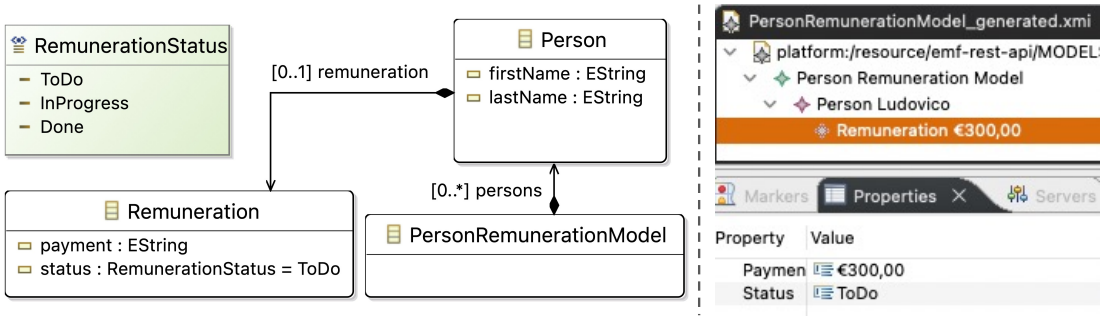

FIGURE 7.2: PersonRemuneration Target Metamodel with a model instance

To fully support the case study specification, we would need to implement (1) a model transformation from the *Guest Invitation* to the *PersonRemuneration* metamodel, (2) a code generator generating web pages for the course/collaboration, (3) a document generator for generating the invitation letter, (4) a script invoking an external Java service for sending emails, and finally (5) all the operations must be supported by validation scripts, assuring that the model instances are correct. The complete implementation details, including code and examples, are available in the repository [168].

Model Transformation. Model transformations are usually defined with languages and tools that transform one model into another. In EMF [46], ATL (Atlas Transformation Language) [68], QVT (Query/View/Transformation) [60], VIATRA (M2M) [172], Epsilon Transformation Language (ETL), Epsilon Object Language (EOL) are some of the tools that can be used to implement a transformation. ATL, ETL, and VIATRA are rule-based model transformation languages in which transformational mappings are expressed via rules, where the modeler can set source and target metamodel elements. These transformation languages express rules with text-based syntaxes, whereas others use graphical editors to offer an intuitive way of representing rules. In Henshin [173], for instance, objects are referred to as nodes, and links between objects are as edges. To support the required scenario, an ETL model-to-model transformation can be developed to generate an instance of the *PersonRemuneration* metamodel from the *GuestInvitation* metamodel.

In a nutshell, each instance of *Guest* in the input model is transformed into a new instance of *Person* in the output model, with the same information, e.g., *firstName* and *lastName* based on the structure of the target metamodel. Also, a new *Remuneration* instance will be generated, including the overall amount of remuneration the guest is receiving.

Model Validation. Data validation within an execution flow provides benefits such as timely error detection, enhanced result reliability, prevention of costly errors, improved performance, data security, and enhanced user experience. When it comes to the execution flow, incorporating data validation can bring numerous advantages. For instance, it can help detect errors in a timely manner, thus helping prevent costly mistakes down the line. Also, it can help ensure the reliability of results, which is critical for many applications or to prevent unexpected behavior. By validating data, the system can ensure that the data is clean and well-structured. Furthermore, incorporating data validation can enhance the user experience, as it may help ensure that it is intuitive and user-friendly. Data validity checks can be performed in the EMF environment using the Epsilon Validation Language (EVL). In our case study, for instance, two validation rules can be defined in the context of the *Remuneration* metaclass to verify (1) if the total amount to be remunerated is positive and (2) if the application status is in the *ReadyToBeSent* state. This prevents remunerating from being computed from negative amounts or when the process is incomplete.

Code generation. Template-Based Code Generation (TBCG) is a code synthesis technique generating code from templates, which are high-level specifications mixing static parts and dynamic parts filled with variables updated from models [174]. In the MDE

ecosystems, some TBCG examples are Acceleo, EGL, or XPand. In an EMF environment, we use Epsilon's EGL transformation language, which the EGX coordination language invokes. Thus an EGX script with a rule for each instance of the *Guest* metaclass in the input model creates an HTML file using the corresponding EGL template where HTML code is interleaved with model navigation constructs. The EGL template generates HTML code by means of the placeholders that fetch data directly from Guest instances.

Document generation. Document generation is an automation task related to transformations and code generation, as they can generate any string or format. Indeed, code generation is also called *Model to Text* transformations. In the MDE ecosystems, we can implement document generation in multiple ways and using multiple languages and tools. Usually, EMF Acceleo or EGL can generate documents precisely as we have developed the source code before. In document generation, a set of rules defined in an EGX file create the invitation letter for the guests. These rules generate a document using the *Guest* instances, and the target language can be a markup language with the extension ".md" or even a Word document. As for the previous task, the EGL template file has a structure that generates text with placeholders allowing the navigation to the input model. This document can then be used in our case study for attaching the invitation letter to the email addressed to the *Guest* that can be automatically sent.

Interaction with external services. To automate the sending of emails to the guest, the Epsilon Object Language (EOL) allows the creation of objects of the underlying programming environment using native types. For instance, an EOL script can use an external service to send an email via Java (e.g., Google API or SMTP library in Java). A Java class could provide a basic configuration for the SMTP host, port, and other required data to send emails programmatically.

Workflow automation in Epsilon. When all these artifacts have been developed, model management activities must be combined together to form the entire workflow. The Epsilon framework provides a set of Apache ANT tasks for assembling multi-step automated build processes. The ANT script reproduces the previous steps, that is, it first loads the input model, and then the target model to be provided as output in the model-to-model transformation is also set. The workflow, therefore, involves validating the input model that may trigger, if positively evaluated, the model-to-model and model-to-code transformations. The invitation email is generated, and finally, through EOL, the external Java service that sends the emails with the input model information is called.

7.4 Solving the example: the WAT version

Workflow automation tools play a significant role in the low-code landscape by offering a visual and declarative approach to designing and orchestrating business processes and automation workflows. Within the realm of low-code development, these tools serve as a fundamental framework to define the flow, logic, integration, and dependencies of tasks and activities within software applications. Workflow tools are useful for automating repetitive tasks, streamlining complex business processes, and enabling efficient collaboration between users and systems. Through their visual representation of workflow logic, these tools enable users to define and connect various steps, actions, and decision points, resulting in the automated and standardized execution of tasks. This automation streamlines operations, reduces manual effort, eliminates errors, and enhances productivity by handling routine activities like data transformation, validation, notifications, and approvals. Moreover, workflow tools often come with a wide range of pre-built components and connectors, simplifying the seamless integration of external systems, databases, APIs, and services. This integration capability empowers users to rapidly build end-to-end solutions by leveraging existing functionality and resources. We use N8N as WAT because of its main advantage compared to the other platforms; it can be self-hosted as a deployment option.

Nodes are the key building blocks of a workflow in N8N, and they can be used to perform a range of actions, including starting the workflow, fetching or sending data to the next node, and manipulating data. In this section we show how to implement the same scenario reported in sec. 7.2 with N8N. In Fig. 7.3, a node mapping data from a Notion[7] to an Airtable[8] record is defined. Both tools are Low-Code Platforms (LCPDs) for building data-driven applications, offering an easy way to configure data schema. They offer multiple views but share a tabular view in which first the user can define the entities, and then the table can be filled with records. In these two tools, the data structure corresponds to our metamodels, *Guest Invitation* and *Person Remuneration*, respectively.

In the screenshot in Fig. 7.3, the node workflow configuration allows us to define the mapping for transforming data from the Notion entities, i.e., guest invitations, to the Airtable ones, representing the remuneration data structure. In particular, this node configuration can be considered a transformation from guests to a flat list of persons that can be used to manage and keep track of remuneration. With the Notion tool, the guest invitations are defined, and with Airtable, the remuneration records are managed, and this node, shown in the center of the picture, automatically flows data.

[7] www.notion.so
[8] www.airtable.com

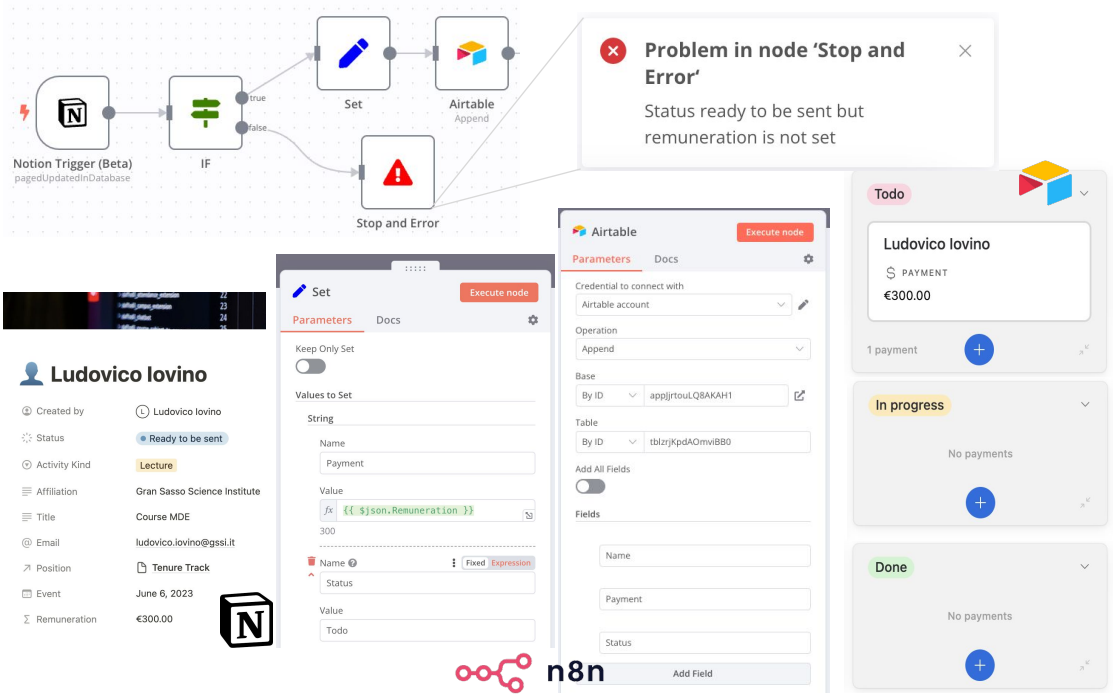

FIGURE 7.3: Model Transformations in N8N.

In this example, we determined how N8N implements transformations by establishing a visual mapping between nodes. Rule-based model transformations have a straightforward way of defining model feature mapping; in rule bindings, the model slots are filled by expressing the mapping directly. Instead, in N8N, syntactic aspects are demanded by the node receiving data. This means that every node can have a different setting and influence the definition of the mapping. For instance, the Airtable node expects to receive data with fields corresponding to the same data structure. Indeed, the guest's name will be automatically propagated. In contrast, the payment must be filled with the value of the *remuneration* and needs an additional trick to propagate values. The action is performed by using a node SET, which is used to set custom variables, and then by setting a variable named *Payment*, the Airtable node will be able to set correctly the value of the field labeled with the same name. This workaround is effective but requires an additional node to implement what could be obtained with a single M2M transformation. The target visualization based on a Kanban view can be easily realized with an additional code generator transforming the output of the transformation into the required view.

As seen in the model validation, in N8N, a node that validates data to be prepared to start the remuneration activity can be created. This validation node checks if there is anything to refund by verifying if the total amount of money to remunerate is greater than zero and if the request status is *ReadyToBeSent*. This is easily done using a conditional node, i.e., IF checking if the total amount of remuneration is greater than

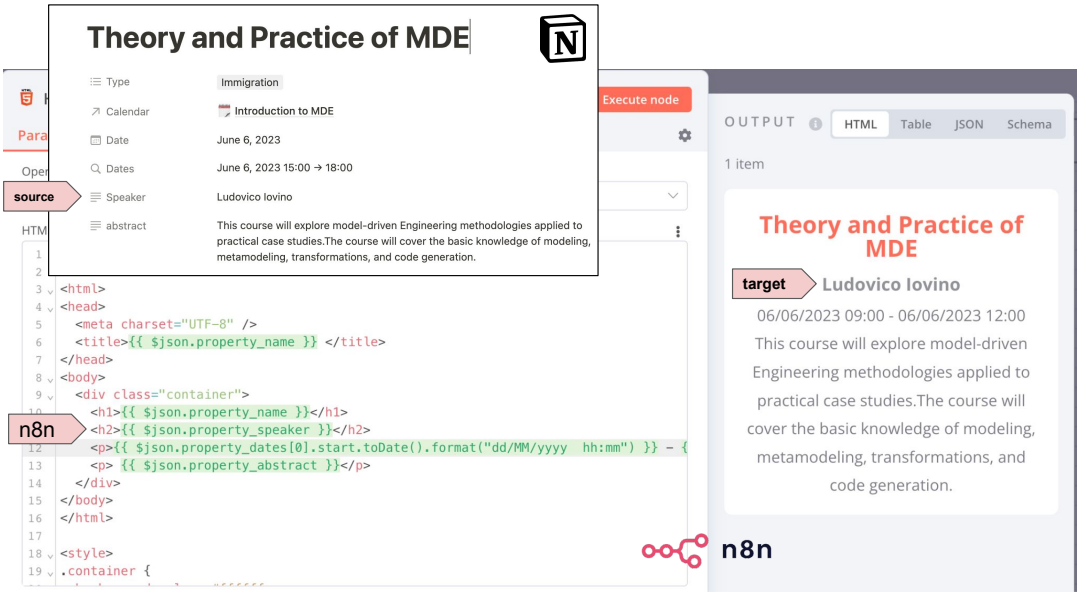

FIGURE 7.4: Code generation in N8N (HTML).

0 and the record status is *ready to be sent*. If this check fails, an error will appear, as shown in Fig 7.3 (error popup).

This is again a very similar result to the one that can be obtained with EVL rules, but in this case, the modeler can express all the constraints in a single script with a very powerful syntax allowing the specification of constructs that can reach a high level of complexity. N8N and, in general, WATs offer node for conditional constructs that can be chained with the same result but with a higher level of effort to express complex validation constraints.

Concerning the code generation for getting the web view showing the required information retrieved from the models, a specific node in N8N permits the generation of HTML. This node, from the given Notion database, generates an HTML page containing information about the scientific collaboration. The template interleaves HTML (target platform language) with the JavaScript code used to access data propagated from the previous nodes. The generated target can be exactly the same as a web page generated with EGL. This is a case limit for workflow and N8N since WATs are often cloud-based (and web-based), and then it would not be possible to target other languages for code generation than HTML, which can be rendered internally by the browser. Indeed, in N8N, we could not find any other nodes for code generation[9].

The last part of the scenario needed to complete the example is document generation and email sending to the guest. This is supported by a GoogleDoc node, where the user

[9]Custom nodes could be developed, but this is out of scope.

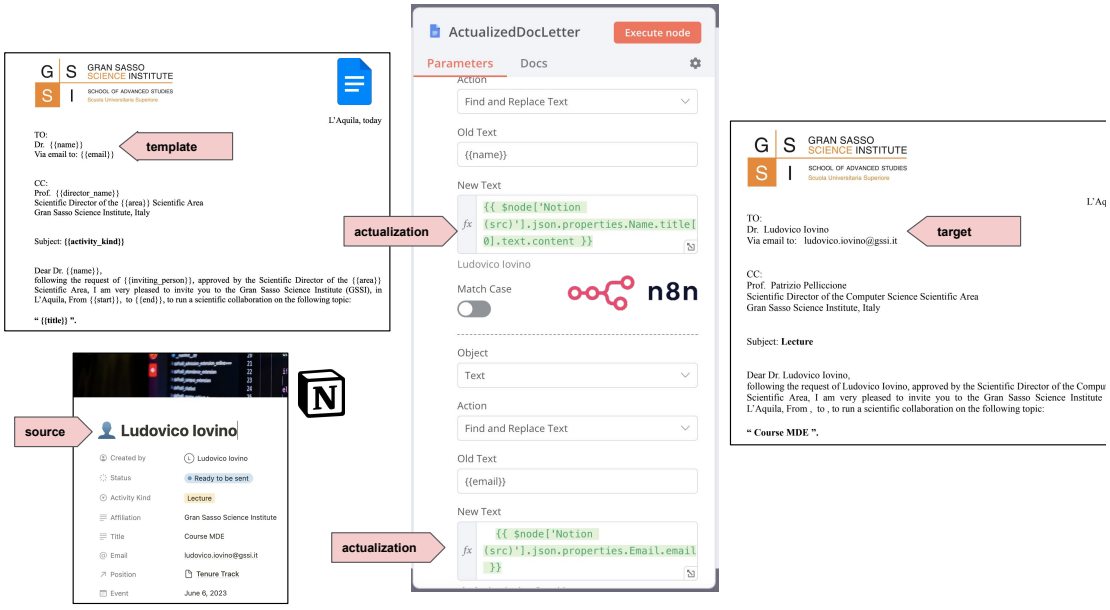

FIGURE 7.5: Document generation in N8N.

can use *search and replace* to fill values in an online Google Doc. To the best of our knowledge[10], this is the only way to implement document generation in this setting. We used a "template" version of GoogleDoc, where we manually specified placeholders with a pre-defined syntax, i.e., {{placeholder}}, that will be replaced at runtime with values retrieved from the data flow when the workflow is executed. Even though it is effective, the main drawback of this method is that the user has to specify an object mapping for each placeholder.

This is quite demanding concerning a code generator engine such as EGL or Acceleo. Then the workflow goes on with a PDF generation node from the GoogleDoc and a Gmail node sending the email to the guest with the attached invitation letter.

7.5 Bridging WATs and MDE ecosystems

We propose a bridging mechanism between WATs and MDE ecosystems by leveraging the EMF-REST technology. We created the EMF-REST node, which was integrated with the N8N platform, to allow communication and enable bidirectional manipulation of models between EMF and WATs and to effectively merge the benefits of both worlds.

The Figure 7.6 illustrates the interoperability achieved between WATs and the EMF Ecosystem through the use of our middleware solution. This middleware bridges the communication gap between the two technical spaces by exposing REST APIs for EMF

[10]The same method can be used with Microsoft SharePoint documents

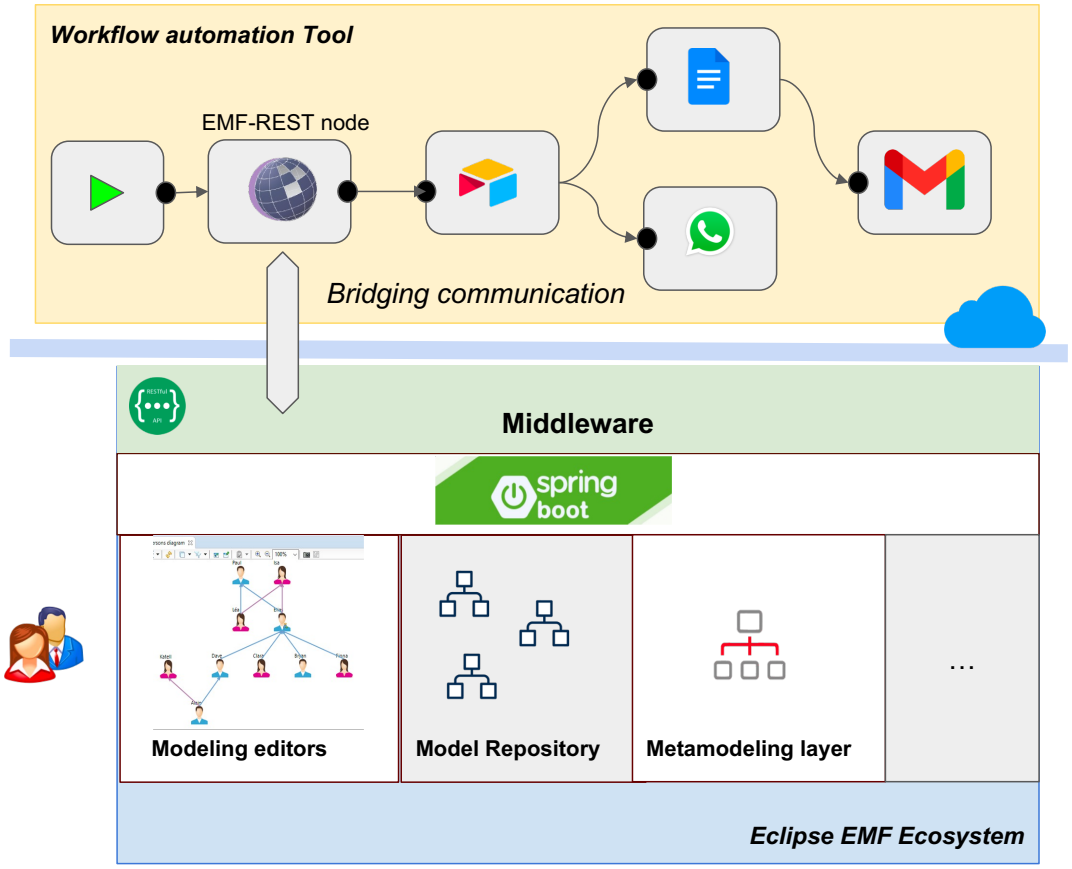

FIGURE 7.6: Bridging WATs and MDE Ecosystems

models, facilitating seamless interaction and data exchange. A key component of this solution is the *EMF-REST node* (shown in the figure), which enables Workflow Automation Tools to access and pass EMF models from the EMF Ecosystem. This node acts as an intermediary, translating the operations and data formats between the WATs and the EMF models, ensuring compatibility and ease of integration. By leveraging the exposed REST APIs offered by our middleware, users can efficiently interact with the modeling repository and metamodeling layers within the EMF Ecosystem, integrating modeling workflows into broader automated processes. This approach demonstrates how the middleware effectively bridges these technical spaces, enabling modelers and modeling editors to work seamlessly with WATs, thus enhancing interoperability and promoting efficient workflow automation.

Figure 7.7 illustrates the integration between EMF and WAT using the REST APIs offered by our middleware. As can be seen, on the left side, is an instance of a model representing a *University Organization* visualized within the EMF editor. This instance model contains various elements such as departments, faculty members, and guests, along with their properties, such as the guest's affiliation, email, and biography. On the right side, the WAT interface displays the custom *EMF REST* node that we developed.

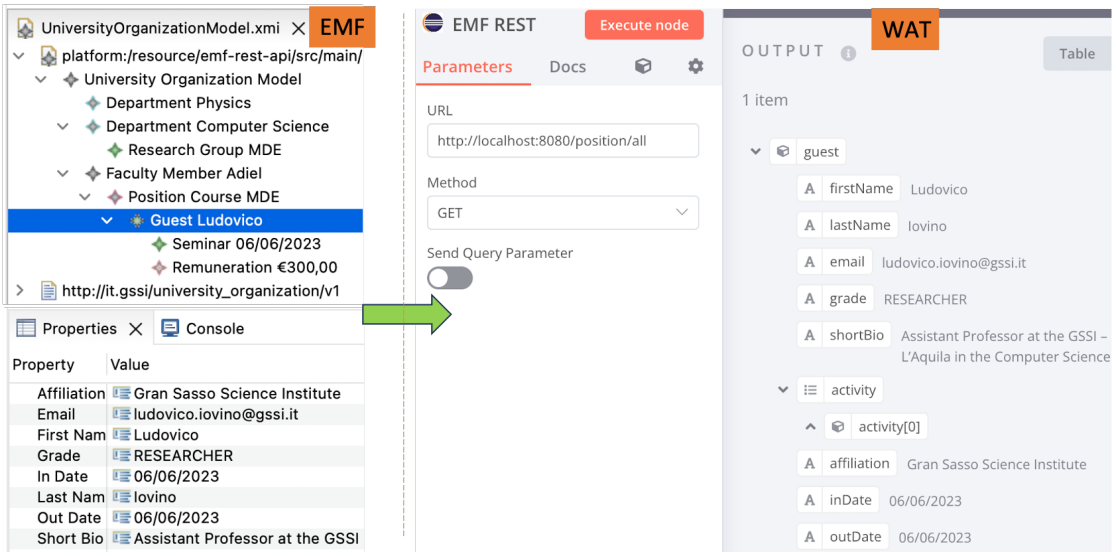

FIGURE 7.7: Integration between EMF and WATs via the REST APIs exposed by our Middleware.

This node uses the REST APIs exposed by our middleware to enable seamless interaction with the EMF models. In this example, the node performs a GET operation to retrieve data from the EMF model, which is then presented in the structured output format on the right-hand side of the figure. The data includes detailed information about a specific guest, such as their name, email, role, and activity records.

7.5.1 Seamless Model Exchange between EMF and WATs through REST APIs

The middleware developed in this research exposes RESTful endpoints that facilitate model retrieval, creation, updating, and deletion. This enhances automation and interoperability between EMF and external systems, such as WATs. By leveraging these REST operations, WATs can dynamically interact with EMF models through the EMF-REST node we created. The following operations are typically performed to enable seamless model exchange. We will use the university collaboration scenario described in Section 7.2 to illustrate these operations.

Model Retrieval: WATs issue GET requests to fetch model elements from the EMF repository. For example, retrieving guest details to generate automated invitation letters.

Model Creation: New model elements are created via POST requests, allowing external data inputs, such as registering a new guest to be automatically added to the EMF model.

Model Update: PUT requests modify existing model elements based on user inputs. For instance, updating collaboration details or adjusting course schedules in response to changes.

Model Deletion: DELETE operations remove model elements, from the EMF repository, such as deleting a guest once collaboration is completed.

The table below illustrates how these REST operations facilitate model exchange between EMF and WATs using a practical university collaboration scenario.

Id	Category	Scenario	Endpoint	Expected Result	Observed Result
S1	Model Retrieval	Retrieve all invited guests from the EMF model to be used in WAT to generate invitation letters.	.../universitymodel/ invitedGuestPositions/all	A list of all invited guest positions is retrieved and used to generate automated invitation letters.	The retrieved guest positions are displayed in the WAT interface and invitation letters are generated.
S2	Model Creation	Add a new invited guest to the EMF model through the WAT interface.	.../universitymodel/ invitedGuestPositions/ newElement	A new invited guest position is added to the EMF model repository.	The newly created invited guest position is visible in the EMF model repository.
S3	Model Update	Update the details of an invited guest position (e.g., lecture date or collaboration period).	.../universitymodel/ invitedGuestPositions/ update?attributeName= lectureDate& updatedValue=14Feb2024	The invited guest position details are updated in the EMF model repository via WAT.	The updated details are reflected in the EMF model and displayed in the WAT interface.
S4	Model Deletion	Delete an invited guest position from the EMF model once the collaboration is completed.	.../universitymodel/ invitedGuestPositions/ deleteByAttribute? attributeName= firstName& attributeValue=Stefano	The invited guest position is deleted from the EMF model repository via WAT.	The invited guest position is successfully removed from both the EMF model and WAT interface.

Table 7.1: Model Exchange between EMF and WATs

7.5.2 Development effort for bridging EMF and WATs

The integration of EMF and WATs was accomplished through a structured and efficient development process, completed in approximately two days. This process involved several key steps, including exploratory research to analyze how WATs nodes exchange data, identifying necessary adjustments to facilitate the seamless exchange of EMF models, and ultimately developing the EMF-REST Node. This node enables interoperability between the EMF ecosystem and WATs, allowing for seamless model exchange. Since WATs (e.g., N8N) utilize JSON for data exchange and our middleware natively exposes EMF models in JSON format through REST APIs, the integration required minimal modifications and no extensive data transformation. This inherent compatibility streamlined the development process, ensuring seamless communication between EMF models and WATs such as N8N. The EMF-REST Node, primarily implemented in JavaScript, was designed to support CRUD operations (GET, POST, PUT, DELETE), enabling WATs to efficiently interact with and manipulate EMF models. The entire implementation was completed within a single day, highlighting the flexibility of the proposed middleware and its ability to facilitate interoperability with external workflow automation platforms with minimal development effort.

7.6 Discussion

We can present some findings and comparisons based on this small-scale experiment. Both development methodologies can help solve the motivating example but with some divergences, similarities, and opportunities.

Workflow propagates data through the nodes, which can be selectively used at all stages. If we consider a workflow as a model transformation chain, this will enable multiple input sources. Instead, rule-based transformation languages can be limited to the number of source/target models in the direct transformation. For instance, ETL can have transformation rules that consume elements from different source models but cannot have multiple source parameters as source elements of the rule. It represents a substantial similarity since, in N8N, the user can select the node from which to obtain data to propagate and transform to the next node. This is similar to what can be done in ATL when the modeler declares the input model and uses OCL [175] to navigate it in the transformation.

Specific syntax for M2M. In EMF, M2M can be expressed with visual or textual syntax, depending on the transformation language, whereas in WATs, mappings are always specified with predefined forms and node concatenation. Besides, in EMF, models are queried, for example, via OCL or EOL, whereas N8N uses Javascript (or proprietary micro syntax in other WATs).

Limited code generation in WAT. Code generation is divided into two subgroups: generating code and generating HTML. Generating code does not seem possible with N8N. However, HTML code generator can be very easy to use, and the node provided by N8N is very similar to an EGX or an Acceleo template.

Similar document generation capabilities. Document generation is possible either way, even though with N8N, we have found a workaround based on searching and replacing strings in manually created templates that over-complicate the specification. Generating documents in EMF requires writing a template that involves model navigation expressions able to retrieve the relevant information to be injected in the targeted system. Searching and replacing in a document requires the specification of multiple fields in the N8N form for each value in which we need a placeholder accessing data. In template-based code generators like Acceleo or EGL, placeholders are specified textually, with less effort when placeholders are many.

Domain-specific syntaxes and editors in MDE. Modeling ecosystems can include multiple artifacts devoted to expressing the models in a domain-specific way. For instance, models can be expressed with DSLs that can be graphical or textual, depending

on the nature of the application domain [176] with great flexibility. WATs are limited to pre-defined custom nodes bridging existing tools in this case, but the composition of services is assured by nature, whereas in EMF, we had to use another tool. Most of the data-driven specification tools that can be used for data management offer a tabular or pre-defined view that cannot be extended much (e.g., Notion or Airtable). On the other side, modeling editors can be very advanced and customized. For instance, editors offering custom palettes and validation constraints, or even form-based editors [177], can easily be developed in EMF.

> The main takeaway message is that integrating services in WATs can be effective and somewhat effortless, but some limitations can hinder their full potential. At the same time, MDE ecosystems can offer custom modeling artifacts development that can fit some domain requirements not implementable in WATs. Given these premises, we can suppose that getting the best of both worlds can be of great benefit.

To this end, we build upon the middleware approach introduced in Chapter 4, which bridges technical spaces by exposing REST APIs for models. This middleware facilitates the generation of RESTful services from models, enabling efficient model management in distributed environments. Most WATs offer integration by exploiting REST API to build custom nodes that can be used in the workflow specification. Leveraging our middleware, which exposes REST APIs for EMF models, we developed a custom node, that can be used within WATs to read and write EMF models. This approach allows users to integrate MDE platforms with WATs without requiring platform migration, thereby preserving the unique advantages of each ecosystem.

The complete implementation details, including code and examples, are available in the repository [168]. Additionally, a video demonstration showcasing the integration of EMF Ecosystems and WATs can be accessed at https://www.youtube.com/watch?v=C0NdMI4h0Eg.

7.7 Conclusion

This chapter compared the MDE ecosystems, which use modeling artifacts to support software development processes, with the ecosystem of services provided by WATs. With the help of a motivating example, we explored how data is manipulated in WATs and how it relates to model management operations in MDE, then we discussed the commonalities and divergence between the two methodologies. While both methodologies provided solutions for the given example, we identified potential opportunities for bridging the two

approaches. We proposed a bridging mechanism between WATs and MDE ecosystems by leveraging our middleware solution. We created the EMF-REST node, which was integrated with the N8N platform, to allow communication and enable bidirectional manipulation of models between EMF and WATs and to effectively merge the benefits of both approaches.

Chapter 8

Conclusion and Future Work

This dissertation addresses fundamental challenges in model management within distributed, web-based environments, with a focus on scalability, interoperability, and dynamic model interaction and manipulation. In the context of MDE, managing large and complex models poses significant challenges, including efficient model storage, manipulation, querying, and seamless integration across diverse modeling tools and platforms.

To overcome these challenges, this dissertation presented a comprehensive middleware solution that advances the state-of-the-art in MDE by enabling dynamic, scalable, and interoperable model management. The proposed approach supports on-the-fly REST API provisioning, facilitating seamless model access and interaction while reducing the need for manual intervention. By addressing key limitations in existing solutions, this research contributes to making model management more flexible, efficient, and accessible in web-based environments.

The key contributions of this work are summarized as follows: (i) Dynamic REST API provisioning for model management. This research proposed a middleware capable of dynamically provisioning REST APIs for any given domain model. Unlike static approaches that required predefined models and manual intervention, the solution enabled on-the-fly API generation, allowing seamless interaction with models. This contribution addressed the challenges outlined in RQ1, facilitating adaptive and efficient model management in web-based environments. (ii) Scalable and stateless model management. A stateless, scalable architecture was designed and implemented to efficiently handle large and complex models. The middleware supported advanced querying mechanisms while maintaining high performance, even for large-scale models. This contribution addressed RQ2, demonstrating how the proposed approach effectively managed large and complex models while ensuring scalability and performance. (iii) Interoperability Across Heterogeneous Modeling Platforms. The proposed middleware enhanced interoperability by

enabling seamless communication and model exchange across diverse model-based tools and platforms. This contribution addressed RQ3, ensuring seamless model exchange across various tools and promoting broader adoption of MDE practices in distributed systems.

Through a comprehensive analysis of existing solutions and the identification of key research challenges, this dissertation bridged significant gaps in current model management solutions. By addressing dynamic API provisioning, scalability, and interoperability, the proposed middleware provided a flexible and adaptive solution capable of scaling with increasing model complexity, supporting evolving models, and integrating seamlessly across heterogeneous tools and platforms. These advancements pave the way for broader adoption of MDE in distributed, web-based environments, making model management more efficient, accessible, and adaptable to evolving software development needs.

8.1 Future Work

While this dissertation addresses fundamental challenges in model management, several open questions and opportunities for further exploration remain. Future research could focus on several key areas to further enhance the proposed middleware.

Handling metamodel evolution and API versioning. The work presented in this thesis focuses on model-based management rather than metamodel-based management. Since models conform to a metamodel that can evolve over time, changes such as adding attributes, modifying relationships, or updating constraints may introduce inconsistencies in existing models and their corresponding REST APIs. Future work will address the challenges associated with metamodel evolution and its impact on dependent artifacts, including API versioning, conflict resolution during metamodel evolution, and the automated adaptation of existing models. These aspects will be part of our future work, with the aim of extending the proposed middleware to dynamically accommodate metamodel changes while ensuring the seamless adaptability of existing models and their corresponding REST APIs.

Support erroneous models. Currently, our approach does not provide an automatic correction mechanism for erroneous models. Instead, we have implemented a Model Validator that flags models violating predefined constraints and returns detailed error messages via the REST API, assisting users in manually identifying and resolving inconsistencies. As part of future work, we plan to introduce transactional support, allowing

models to exist in a temporarily incorrect state until corrections are applied. Additionally, we aim to implement a rollback mechanism, enabling users to revert models to a previously valid state when modifications introduce unresolvable inconsistencies. These enhancements will improve model resilience, usability, and error recovery, making the validation process more flexible and user-friendly.

Expanding domain applications. As part of future work, we plan to explore additional case studies in diverse domains, such as automotive and aerospace, where growing model complexity and scalability demands pose unique challenges. These evaluations will further assess the middleware's adaptability and ensure its applicability across a broader range of domains. In addition, we plan to evaluate and extend the middleware by integrating it with widely used modeling tools, such as Sirius, Papyrus, and others. This expansion will enhance cross-platform compatibility, enabling seamless interoperability across diverse modeling environments and fostering broader adoption within the MDE community. Furthermore, we plan to explore distributed model storage and provide transparent access to models across different repositories.

Scalability for ultra-large models. Scalability for managing ultra-large models remains a critical challenge as models grow in size and complexity. Future research could explore advanced techniques such as model partitioning, load balancing, and adaptive caching strategies to improve performance. Implementing graph-based partitioning algorithms could enable efficient distribution of model elements across multiple nodes, optimizing query execution and retrieval processes. Additionally, incremental query processing and parallel execution of model manipulations could help reduce latency and resource consumption. Leveraging cloud-native solutions and container orchestration platforms (e.g., Kubernetes) could further enhance scalability by dynamically allocating resources based on workload demands in distributed environments.

Enhancing model security. Ensuring model security in distributed environments is a critical aspect that cannot be overlooked in model management. Future work could focus on integrating a security feature into our middleware. As models in domains such as healthcare, finance, and autonomous systems become increasingly complex and interconnected, safeguarding against unauthorized access and protecting sensitive data becomes paramount. One promising direction is the implementation of secure model retrieval, where the middleware integrates authentication hooks to verify users or systems requesting a model. This could be complemented by role-based access control and multi-factor authentication mechanisms. Additionally, incorporating audit logging to monitor model usage and ensure that sensitive models remain protected, as well as implementing blockchain-based integrity verification, could provide transparency and ensure models are tamper-proof.

Supporting domain-specific customization. One significant area of exploration is supporting domain-specific customization while preserving the middleware's generic approach. This could involve developing modular extensions that are tailored to specific domains, such as automotive, healthcare, finance, or aerospace, without compromising the system's flexibility. For example, in the automotive sector, models representing autonomous vehicle systems may require specialized APIs for real-time safety verification and sensor fusion. Developing customizable plug-in architectures or microservice-based extensions could make the middleware adaptable to industry-specific requirements while maintaining its core functionality.

Integration of AI-driven techniques into model management. The integration of AI-driven techniques into model management offers promising avenues to enhance querying, validation, and transformation processes. Traditional model querying with languages like OCL [28] or EOL [51] can be complex and require specialized knowledge. While OCL is commonly used for querying, it also plays a crucial role in defining constraints and restrictions within models to ensure their validity and consistency. The integration of AI-driven techniques into model management offers a promising way to enhance querying, validation, and transformation processes. In particular, Large Language Models (LLMs) can simplify model querying by enabling interaction through natural language, significantly reducing the learning curve and improving accessibility. For example, instead of writing intricate queries, users can ask AI to retrieve specific model elements based on criteria like attribute counts or relationships to particular concepts. Moreover, LLMs can support OCL constraints by interpreting natural language inputs to generate appropriate OCL expressions or validate existing constraints. This allows non-experts to define and enforce model restrictions more intuitively. Furthermore, context-aware assistance powered by LLMs can provide intelligent suggestions or refine queries based on user intent and the model's structure. This approach makes model querying more intuitive and accelerates the adoption of Model-Driven Engineering practices.

Bibliography

[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[2] Konstantinos Barmpis and Dimitrios S Kolovos. Comparative analysis of data persistence technologies for large-scale models. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 33–38, 2012.

[3] Sitepoint. REST API call. https://www.sitepoint.com/rest-api/, 2024. [Online; accessed October 2024].

[4] Swagger. API Documentation. https://editor.swagger.io/, 2024. [Online; accessed October 2024].

[5] n8n. Workflow Automation Tool. https://n8n.io/, 2024. [Online; accessed November 2024].

[6] Bente Anda, Kai Hansen, Ingolf Gullesen, and Hanne Kristin Thorsen. Experiences from introducing uml-based development in a large safety-critical project. *Empirical Software Engineering*, 11:555–581, 2006.

[7] Douglas C Schmidt et al. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.

[8] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19:5–13, 2020.

[9] Charlotte Verbruggen and Monique Snoeck. Practitioners' experiences with model-driven engineering: a meta-review. *Software and Systems Modeling*, 22(1):111–129, 2023.

[10] Konstantinos Barmpis and Dimitrios S Kolovos. Evaluation of contemporary graph databases for efficient persistence of large-scale models. *J. Object Technol.*, 13(3): 3–1, 2014.

[11] Javier Espinazo Pagán and Jesús García Molina. Querying large models efficiently. *Information and Software Technology*, 56(6):586–622, 2014.

[12] Antonio Jiménez-Pastor, Antonio Garmendia, and Juan de Lara. Scalable model exploration for model-driven engineering. *Journal of Systems and Software*, 132: 204–225, 2017.

[13] Dimitrios S Kolovos, Louis M Rose, Nicholas Matragkas, Richard F Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, pages 1–10, 2013.

[14] Arsene Indamutsa, Juri Di Rocco, Lissette Almonte, Davide Di Ruscio, and Alfonso Pierantonio. Advanced discovery mechanisms in model repositories. *Software: Practice and Experience*, 2024.

[15] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. EMF-REST: Generation of Restful APIs from Models. In *ACM Symp. on Applied Computing*, pages 1446–1453, 2016.

[16] Adiel Tuyishime, Francesco Basciani, Javier Luis Cánovas Izquierdo, and Ludovico Iovino. Dynamic provisioning of rest apis for model management. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 1335–1337. IEEE, 2024.

[17] Adiel Tuyishime, Francesco Basciani, Javier Luis Canovas Izquierdo, and Ludovico Iovino. Enhancing model management with automated rest api generation. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 36–40, 2024.

[18] Adiel Tuyishime, Francesco Basciani, Ludovico Iovino, Javier Luis Cánovas Izquierdo, Jordi Cabot, and Alfonso Pierantonio. Bridging workflow automation tools and emf modeling ecosystems. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 893–897. IEEE, 2023.

[19] Adiel Tuyishime, Francesco Basciani, Amleto Di Salle, Javier Luis Cánovas Izquierdo, and Ludovico Iovino. Streamlining workflow automation with a model-based assistant. In *50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2024). (To appear)*, 2024.

[20] Adiel Tuyishime, Javier Luis Cánovas Izquierdo, Maria Teresa Rossi, Martina De Sanctis, et al. Modeling linked open data (poster). In {*STAF*} *2022 Workshop Proceedings: 10th International Workshop on BidirectionalTransformations*

{*(BX*} *2022), 2nd International Workshop on Foundationsand Practice of Visual Modeling* {*(FPVM*} *2022) and 2nd InternationalWorkshop on* {*MDE*} *for Smart IoT Systems (MeSS 2022)(co-located withSoftware Technologies: Applications and Foundations federation ofconferences* {*(STAF*} *2022)), Nantes, France, July 5-8, 2022*, volume 3250. CEUR-WS. org, 2022.

[21] Samira Silva, Adiel Tuyishime, Tiziano Santilli, Patrizio Pelliccione, and Ludovico Iovino. Quality metrics in software architecture. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 58–69. IEEE, 2023.

[22] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43: 139–155, 2015.

[23] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[24] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2013.

[25] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective model driven engineering. In *UML 2003-The Unified Modeling Language. Modeling Languages and Applications: 6th International Conference, San Francisco, CA, USA, October 20-24, 2003. Proceedings 6*, pages 175–189. Springer, 2003.

[26] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006. doi: 10.1109/MC.2006.58.

[27] Object Management Group (OMG). Meta Object Facility. https://www.omg.org/spec/MOF/2.0/PDF/, 2024. [Online; accessed October 2024].

[28] Object Management Group. Object constraint language (ocl). http://www.omg.org/spec/OCL/, 2024. [Online; accessed November 2024].

[29] OMG MOF 2 XMI Mapping Specification version 2.4.1. XML Metadata Interchange. https://www.omg.org/spec/XMI/2.4.1/, 2024. [Online; accessed October 2024].

[30] Tony Clark, Paul Sammut, and James Willans. Applied metamodelling: a foundation for language driven development. *arXiv preprint arXiv:1505.00149*, 2015.

[31] Martin Grossman, Jay E Aronson, and Richard V McCarthy. Does uml make the grade? insights from the software development community. *Information and Software Technology*, 47(6):383–397, 2005.

[32] David Budgen, Andy J Burn, O Pearl Brereton, Barbara A Kitchenham, and Rialette Pretorius. Empirical evidence about the uml: a systematic literature review. *Software: Practice and Experience*, 41(4):363–392, 2011.

[33] Michel RV Chaudron, Werner Heijstek, and Ariadi Nugroho. How effective is uml modeling? an empirical perspective on costs and benefits. *Software & Systems Modeling*, 11:571–580, 2012.

[34] Brian Dobing and Jeffrey Parsons. How uml is used. *Communications of the ACM*, 49(5):109–113, 2006.

[35] Miroslaw Staron. Adopting model driven software development in industry–a case study at two companies. In *International Conference on Model Driven Engineering Languages and Systems*, pages 57–72. Springer, 2006.

[36] Luciane Telinski Wiedermann Agner, Inali Wisniewski Soares, Paulo Cézar Stadzisz, and Jean Marcelo Simão. A brazilian survey on uml and model-driven practices for embedded software development. *Journal of systems and software*, 86(4):997–1005, 2013.

[37] Tomaž Kosar, Sašo Gaberc, Jeffrey C Carver, and Marjan Mernik. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*, 23:2734–2763, 2018.

[38] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001.

[39] David S Wile. Supporting the dsl spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.

[40] Paul Hudak. Domain-specific languages. *Handbook of programming languages*, 3 (39-60):21, 1997.

[41] Steffen Zschaler, Dimitrios S Kolovos, Nikolaos Drivalos, Richard F Paige, and Awais Rashid. Domain-specific metamodelling languages for software language engineering. In *Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers 2*, pages 334–353. Springer, 2010.

[42] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[43] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[44] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[45] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-specific development with visual studio dsl tools*. Pearson Education, 2007.

[46] Eclipse Foundation. Eclipse Modelling Framework (EMF). https://eclipse.dev/modeling/emf/, 2024. [Online; accessed October 2024].

[47] Eclipse Foundation. Eclipse Marketplace. https://marketplace.eclipse.org/, 2024. [Online; accessed November 2024].

[48] Eclipse Foundation. Eclipse Foundation. https://www.eclipse.org/org/foundation/, 2024. [Online; accessed October 2024].

[49] Dimitriois Kolovos. *An extensible platform for specification of integrated languages for model management*. PhD thesis, Citeseer, 2008.

[50] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Eclipse development tools for epsilon. In *Eclipse summit Europe, eclipse modeling symposium*, volume 20062, page 200, 2006.

[51] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (eol). In *European conference on model driven architecture-foundations and applications*, pages 128–142. Springer, 2006.

[52] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings 1*, pages 46–60. Springer, 2008.

[53] Dimitrios S Kolovos. Establishing correspondences between models with the epsilon comparison language. In *Model Driven Architecture-Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings 5*, pages 146–157. Springer, 2009.

[54] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Merging models with the epsilon merging language (eml). In *International conference on model driven engineering languages and systems*, pages 215–229. Springer, 2006.

[55] Epsilon. Epsilon Validation Language (EVL). https://eclipse.dev/epsilon/doc/evl/, 2024. [Online; accessed November 2024].

[56] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The epsilon generation language. In *Model Driven Architecture–Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings 4*, pages 1–16. Springer, 2008.

[57] Epsilon. Model Migration (Flock). https://eclipse.dev/epsilon/doc/flock/, 2024. [Online; accessed November 2024].

[58] Epsilon. Epsilon Model Connectivity Layer. https://eclipse.dev/epsilon/doc/emc/, 2024. [Online; accessed October 2024].

[59] Louis Rose, Esther Guerra, Juan De Lara, Anne Etien, Dimitris Kolovos, and Richard Paige. Genericity for model management operations. *Software & Systems Modeling*, 12:201–219, 2013.

[60] MOF 2.0 Query / Views / Transformations. *OMG: RFP, OMG Document*, ad/2002-04-10, 2002.

[61] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[62] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. On the evolution of ocl for capturing structural constraints in modelling languages. *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, pages 204–218, 2009.

[63] Carlos Noguera and Viviane Jonckers. Model querying with query models. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 181–184, 2015.

[64] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. In *Theory and Practice of Model Transformations: 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings 4*, pages 167–182. Springer, 2011.

[65] Richard F Paige, Nicholas Matragkas, and Louis M Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, 2016.

[66] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In *International Conference on the Unified Modeling Language*, pages 134–148. Springer, 2001.

[67] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20, 2006.

[68] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, 2006.

[69] Eclipse. Acceleo. https://eclipse.dev/acceleo/, 2024. [Online; accessed November 2024].

[70] Eclipse Foundation. EMF Compare. https://eclipse.dev/emf/compare/, 2024. [Online; accessed October 2024].

[71] Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for uml models. 2005.

[72] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, Fiona AC Polack, and Simon Poulding. Epsilon flock: a model migration language. *Software & Systems Modeling*, 13:735–755, 2014.

[73] Eclipse Foundation. Emfstore. https://eclipse.dev/emfstore/, 2024. [Online; accessed October 2024].

[74] Eclipse Foundation. The cdo model repository (cdo). https://eclipse.dev/cdo/, 2024. [Online; accessed October 2024].

[75] Eclipse Foundation. The Epsilon Merging Language (EML). https://eclipse.dev/epsilon/doc/eml/, 2024. [Online; accessed October 2024].

[76] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032, 2014.

[77] Konstantinos Barmpis and Dimitris Kolovos. Hawk: Towards a scalable model indexing architecture. In *Proceedings of the workshop on scalability in model driven engineering*, pages 1–9, 2013.

[78] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Scalability: The holy grail of model driven engineering. In *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, pages 10–14, 2008.

[79] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. A repository for scalable model management. *Software & Systems Modeling*, 14(1): 219–239, 2015.

[80] Maximilian Koegel and Jonas Helming. Emfstore: a model repository for emf models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 307–308, 2010.

[81] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs: Services for a Changing World*. " O'Reilly Media, Inc.", 2013.

[82] Dan Woods, Greg Brail, and D Jacobson. Apis: A strategy guide, 2011.

[83] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[84] R Fielding, M Nottingham, and J Reschke. Rfc 9110: Http semantics, 2022.

[85] Alen Simec and Magdalena Maglicic. Comparison of json and xml data formats. In *Central European Conference on Information and Intelligent Systems*, page 272. Faculty of Organization and Informatics Varazdin, 2014.

[86] OpenAPI Initiative. OpenAPI Specification. https://spec.openapis.org/oas/v3.1.1.html, 2024. [Online; accessed October 2024].

[87] Swagger. Swagger, Design and document APIs. https://swagger.io/, 2024. [Online; accessed October 2024].

[88] RAML. RESTful API Modeling Language. https://raml.org/, 2024. [Online; accessed October 2024].

[89] OpenAPI Initiative. OpenAPI Initiative. https://www.openapis.org/, 2024. [Online; accessed October 2024].

[90] Meg Fryling. Low code app development. *Journal of Computing Sciences in Colleges*, 34(6):119–119, 2019.

[91] Massimo Tisi, Jean-Marie Mottu, Dimitrios S Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*, 2019.

[92] Clay Richardson, John R Rymer, Christopher Mines, Alex Cullen, and Dominique Whittaker. New development platforms emerge for customer-facing applications. *Forrester: Cambridge, MA, USA*, 15, 2014.

[93] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21(2):437–446, 2022.

[94] Notion. Notion. https://www.notion.com/, 2024. [Online; accessed November 2024].

[95] Airtable. Airtable. https://www.airtable.com/, 2024. [Online; accessed November 2024].

[96] Slack. Slack. https://slack.com/, 2024. [Online; accessed November 2024].

[97] Jordi Cabot. Positioning of the low-code movement within the field of model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–3, 2020.

[98] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3): 271–304, 2009.

[99] Eclipse Foundation. Teneo/Hibernate. https://wiki.eclipse.org/Teneo/Hibernate/, 2024. [Online; accessed October 2024].

[100] OrientDB. OrientDB, Hybrid Document-Store and Graph NoSQL. https://orientdb.org/, 2024. [Online; accessed October 2024].

[101] Neo4J. Neo4J, Graph NoSQL Database. https://neo4j.com/, 2024. [Online; accessed October 2024].

[102] MongoDB. MongoDB, Document-Store NoSQL Database. https://www.mongodb.com/, 2024. [Online; accessed October 2024].

[103] HBase. Apache HBase. https://hbase.apache.org/, 2024. [Online; accessed October 2024].

[104] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2): 12–14, 2010.

[105] Felicien Ihirwe, Arsene Indamutsa, Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. Cloud-based modeling in iot domain: a survey, open challenges and

opportunities. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 73–82. IEEE, 2021.

[106] Francesco Basciani, Pierantonio Alfonso, Ludovico Iovino, et al. Model repositories: Will they become reality? a position statement. In *Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 29, 2015*, volume 1563, pages 37–42. CEUR-WS, 2015.

[107] José Antonio Hernández López and Jesús Sánchez Cuadrado. Mar: a structure-based search engine for models. In *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems*, pages 57–67, 2020.

[108] Ta'id Holmes, Uwe Zdun, and Schahram Dustdar. Morse: A model-aware service environment. In *2009 IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 470–477. IEEE, 2009.

[109] Daniel Lucrédio, Renata P de M. Fortes, and Jon Whittle. Moogle: a metamodel-based model search engine. *Software & Systems Modeling*, 11:183–208, 2012.

[110] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28–October 3, 2014. Proceedings 17*, pages 653–669. Springer, 2014.

[111] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Exploring model repositories by means of megamodel-aware search operators. In *MoDELS (Workshops)*, pages 793–798, 2018.

[112] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, and Thomas H Kolbe. Automated and transparent model fragmentation for persisting large models. In *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings 15*, pages 102–118. Springer, 2012.

[113] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. Distributing relational model transformation on mapreduce. *Journal of Systems and Software*, 142:1–20, 2018.

[114] Ábel Hegedüs, Gábor Bergmann, Csaba Debreceni, Ákos Horváth, Péter Lunk, Ákos Menyhért, István Papp, Dániel Varró, Tomas Vileiniskis, and István Ráth. Incquery server for teamwork cloud: Scalable query evaluation over collaborative model repositories. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 27–31, 2018.

[115] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Francis Bordeleau, and Jordi Cabot. Wapiml: towards a modeling infrastructure for web apis. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 748–752. IEEE, 2019.

[116] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Openapitouml: a tool to generate uml models from openapi definitions. In *International Conference on Web Engineering*, pages 487–491. Springer, 2018.

[117] Javier Luis Cánovas Izquierdo, Frédéric Jouault, Jordi Cabot, and Jesús García Molina. Api2mol: Automating the building of bridges between apis and model-driven engineering. *Information and Software Technology*, 54(3):257–273, 2012.

[118] Eclipse Foundation. Eclipse Sirius Web. https://eclipse.dev/sirius/sirius-web.html, 2024. [Online; accessed October 2024].

[119] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.

[120] Rodrigo Bonifácio, Thiago M Castro, Ricardo Fernandes, Alisson Palmeira, and Uirá Kulesza. Neoidl: A domain-specific language for specifying rest services. In *SEKE*, pages 613–618, 2015.

[121] Florian Haupt, Dimka Karastoyanova, Frank Leymann, and Benjamin Schroth. A model-driven approach for rest compliant services. In *2014 IEEE International Conference on Web Services*, pages 129–136. IEEE, 2014.

[122] Amirhossein Deljouyi and Raman Ramsin. Mdd4rest: Model-driven methodology for developing restful web services. In *MODELSWARD*, pages 93–104, 2022.

[123] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *Emf: Eclipse Modeling Framework*. 2008.

[124] Nora Koch and Sergej Kozuruba. Requirements models as first class entities in model-driven web engineering. In *Current Trends in Web Engineering: ICWE*

2012 International Workshops: MDWE, ComposableWeb, WeRE, QWE, and Doctoral Consortium, Berlin, Germany, July 23-27, 2012, Revised Selected Papers 12, pages 158–169. Springer, 2012.

[125] Marco Brambilla, Jordi Cabot, and Michael Grossniklaus. Tools for modeling and generating safe interface interactions in web applications. In *Web Engineering: 10th International Conference, ICWE 2010, Vienna Austria, July 5-9, 2010. Proceedings 10*, pages 482–485. Springer, 2010.

[126] Xhevi Qafmolla and Viet Cuong Nguyen. Automation of web services development using model driven techniques. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 3, pages 190–194. IEEE, 2010.

[127] Andrea Schauerhuber, Manuel Wimmer, and Elisabeth Kapsammer. Bridging existing web modeling languages to model-driven engineering: a metamodel for webml. In *Workshop proceedings of the sixth international conference on Web engineering*, pages 5–es, 2006.

[128] E Michael Maximilien, Hernan Wilkinson, Nirmit Desai, and Stefan Tai. A domain-specific language for web apis and services mashups. In *Service-Oriented Computing–ICSOC 2007: Fifth International Conference, Vienna, Austria, September 17-20, 2007. Proceedings 5*, pages 13–26. Springer, 2007.

[129] José Matías Rivero, Sebastian Heil, Julián Grigera, Martin Gaedke, and Gustavo Rossi. Mockapi: an agile approach supporting api-first web application development. In *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings 13*, pages 7–21. Springer, 2013.

[130] Nírondes AC Tavares and Samyr Vale. A model driven approach for the development of semantic restful web services. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, pages 290–299, 2013.

[131] Anne Geraci. *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries*. IEEE Press, 1991.

[132] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4:171–188, 2005.

[133] MN Wicks and RG Dewar. A new research agenda for tool integration. *Journal of Systems and Software*, 80(9):1569–1585, 2007.

[134] Harold Ossher, William Harrison, and Peri Tarr. Software engineering tools and environments: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 261–277, 2000.

[135] Susan Elliott Sim. Next generation data interchange: Tool-to-tool application program interfaces. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 278–280. IEEE, 2000.

[136] Yimin Bao and Ellis Horowitz. A new approach to software tool interoperability. In *Proceedings of the 1996 ACM symposium on Applied Computing*, pages 500–509, 1996.

[137] Brian Elvesæter, Axel Hahn, Arne-Jørgen Berre, and Tor Neple. Towards an interoperability framework for model-driven development of software systems. In *Interoperability of enterprise software and applications*, pages 409–420. Springer, 2006.

[138] Heiko Kern and Stefan Kühne. Integration of microsoft visio and eclipse modeling framework using m3-level-based bridges. In *Proceedings of Second Workshop on Model-Driven Tool and Process Integration (MDTPI) at ECMFA, CTIT Workshop Proceedings*, pages 13–24. Citeseer, 2009.

[139] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Model-driven tool interoperability: An application in bug tracking. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29-November 3, 2006. Proceedings, Part I*, pages 863–881. Springer, 2006.

[140] Néjib Moalla, Hanène Chettaoui, Yacine Ouzrout, Frédéric Noël, and Abdelaziz Bouras. Model driven architecture to enhance interoperability between product applications. In *PLM'08*, pages 380–392. Inderscience Enterprises LTD, 2008.

[141] Yu Sun, Zekai Demirezen, Frédéric Jouault, Robert Tairas, and Jeff Gray. A model engineering approach to tool interoperability. In *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers 1*, pages 178–187. Springer, 2009.

[142] Heiko Kern and Stefan Kühne. Model interchange between aris and eclipse emf. 2007.

[143] Tian Zhang, Frédéric Jouault, Jean Bézivin, and Xuandong Li. An mde-based method for bridging different design notations. *Innovations in Systems and Software Engineering*, 4:203–213, 2008.

[144] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. Model bus: Towards the interoperability of modelling tools. In *Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004. Revised Selected Papers*, pages 17–32. Springer, 2005.

[145] Object Management Group (OMG). Unified Modeling Language Superstructure Specification. https://www.omg.org/spec/UML/2.1.2/Superstructure/PDF, 2024. [Online; accessed October 2024].

[146] Hugo Bruneliere, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin. Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools. In *European Conference on Modelling Foundations and Applications*, pages 32–47. Springer, 2010.

[147] Obeo. Sirius. https://eclipse.dev/sirius/, 2024. [Online; accessed March 2024].

[148] Theia. Theia. https://theia-ide.org/, 2024. [Online; accessed March 2024].

[149] (Online material). Dynamic Provisioning of REST APIs for Model Management . https://github.com/tuadiel6/DynamicEMF-REST-API/blob/main/Scenario.md, 2024. [Online; accessed June 2024].

[150] Spring Boot. Spring Boot. https://spring.io/projects/spring-boot, 2024. [Online; accessed March 2024].

[151] Önder Babur, Aishwarya Suresh, Wilbert Alberts, Loek Cleophas, Ramon Schiffelers, and Mark van den Brand. Model Analytics for Industrial Mde Ecosystems. In *Model Management and Analytics for Large Scale Systems*, pages 273–316. 2020.

[152] Valdemar Vicente Graciano Neto, Fabio Basso, Rodrigo Pereira dos Santos, Noor Hasrina Bakar, Mohamad Kassab, Claudia Werner, Toacy Oliveira, and Elisa Yumi Nakagawa. Model-driven Engineering Ecosystems. In *Int. Workshop on Software Engineering for Systems-of-Systems and Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems*, pages 58–61. IEEE, 2019.

[153] Srdjan Stevanetic, Muhammad Atif Javed, and Uwe Zdun. Empirical Evaluation of the Understandability of Architectural Component Diagrams. In *Working Conf. on Software Architecture*, pages 1–8. 2014.

[154] David Garlan and Bradley Schmerl. Using Architectural Models at Runtime: Research Challenges. In *Europ. Workshop on Software Architecture*, pages 200–205, 2004. ISBN 978-3-540-24769-2.

[155] Alessio Bucaioni, Amleto Di Salle, Ludovico Iovino, Leonardo Mariani, and Patrizio Pelliccione. Continuous conformance of software architectures. In *21st IEEE International Conference on Software Architecture (ICSA)*, 2024. doi: 10.1109/ICSA59870.2024.00019.

[156] Amazon. Greengrass Stream Manager. https://docs.aws.amazon.com/greengrass/?id=docs_gateway, 2024. [Online; accessed March 2024].

[157] Alessio Bucaioni, Amleto Di Salle, Ludovico Iovino, Ivano Malavolta, and Patrizio Pelliccione. Reference Architectures Modelling and Compliance Checking. *Software and Systems Modeling*, 22(3):891–917, 2023.

[158] Jasmin Guth, Uwe Breitenbücher, Michael Falkenthal, Paul Fremantle, Oliver Kopp, Frank Leymann, and Lukas Reinfurt. *A Detailed Analysis of IoT Platform Architectures: Concepts, Similarities, and Differences*, pages 81–101. 2018.

[159] Baikuntha Narayan Biswal, Pragyan Nanda, and Durga Prassad Mohapatra. A Novel Approach for Scenario-based Test Case Generation. In *Int. Conf. on Information Technology*, pages 244–247. IEEE, 2008.

[160] Konstantinos Barmpis and Dimitrios S Kolovos. Towards scalable querying of large-scale models. In *Modelling Foundations and Applications: 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings 10*, pages 35–50. Springer, 2014.

[161] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. ModelSet: a dataset for machine learning in model-driven engineering. *Softw. Syst. Model.*, 21(3):967–986, 2022. URL https://doi.org/10.1007/s10270-021-00929-3.

[162] Frédéric Jouault, J Sottet, et al. An amma/atl solution for the grabats 2009 reverse engineering case study. In *5th International Workshop on Graph-Based Tools, Grabats*, pages 123–137, 2009.

[163] Jupyter. Jupyter Notebook. https://jupyter.org/, 2024. [Online; accessed November 2024].

[164] Ma Esperanza Manso, Marcela Genero, and Mario Piattini. No-redundant metrics for uml class diagram structural complexity. In *Advanced Information Systems Engineering: 15th International Conference, CAiSE 2003 Klagenfurt/Velden, Austria, June 16–20, 2003 Proceedings 15*, pages 127–142. Springer, 2003.

[165] Bhawana Mathur and Manju Kaushik. Empirical analysis of metrics using uml class diagram. *International Journal of Advanced Computer Science and Applications*, 7(5), 2016.

[166] Peshawa Jamal Muhammad Ali, Rezhna Hassan Faraj, Erbil Koya, Peshawa J Muhammad Ali, and Rezhna H Faraj. Data normalization and standardization: a technical report. *Mach Learn Tech Rep*, 1(1):1–6, 2014.

[167] Eclipse Epsilon website. https://eclipse.dev/epsilon/. Accessed: 2023-08-21.

[168] Ludovico Iovino, Basciani Francesco, and Tuyishime Adiel. Model management operations in MDE. https://github.com/gssi/emf_workflow_project/tree/main, 2023.

[169] Ludovico Iovino, Alfonso Pierantonio, and Ivano Malavolta. On the Impact Significance of Metamodel Evolution in MDE. *J. Object Technol.*, 11(3):3–1, 2012.

[170] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A.C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Int. Conf. on Engineering of Complex Computer Systems*, pages 162–171, 2009.

[171] Anas Abouzahra, Ayoub Sabraoui, and Karim Afdel. Model composition in Model Driven Engineering: A systematic literature review. *Information and Software Technology*, 125:106316, 2020.

[172] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA-visual automated transformations for formal verification and validation of UML models. In *Int. Conf. on Automated Software Engineering,*, pages 267–270. IEEE, 2002.

[173] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *Int. Conf. on Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[174] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems Structures*, 52:43–62, 2018. ISSN 1477-8424.

[175] Mark Richters and Martin Gogolla. OCL: Syntax, semantics, and tools. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.

[176] Santiago Meliá, Cristina Cachero, Jesús M Hermida, and Enrique Aparicio. Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: an empirical pilot study. *Software Quality Journal*, 24:709–735, 2016.

[177] Lorenzo Bettini. Developing user interfaces with EMF parsley. In *Int. Conf. on Software Paradigm Trends (ICSOFT-PT)*, pages 58–66. IEEE, 2014.