TU WIEN Informatics

Encoding Semantic Information in Conceptual Models for Machine Learning Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Philipp-Lorenz Glaser, BSc
Matrikelnummer 11776175

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Mitwirkung: Univ.Ass. Syed Juned Ali, BSc MSc

Wien, 4. Juni 2025

Philipp-Lorenz Glaser	Dominik Bork

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Encoding Semantic Information in Conceptual Models for Machine Learning Applications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Philipp-Lorenz Glaser, BSc
Registration Number 11776175

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Assistance: Univ.Ass. Syed Juned Ali, BSc MSc

Vienna, June 4, 2025

Philipp-Lorenz Glaser	Dominik Bork

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Philipp-Lorenz Glaser, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 4. Juni 2025

Philipp-Lorenz Glaser

Acknowledgements

First and foremost, I am deeply grateful to my parents for their emotional and financial support, not only during this thesis, but throughout my entire academic journey.

Special thanks go to my girlfriend, Valentina, for her constant emotional support, patience, and encouragement during challenging times, despite not being able to spend as much time together as we would have wished.

I would also like to sincerely thank my supervisors, Associate Prof. Dr. Dominik Bork and Univ.Ass. Syed Juned Ali BSc MSc, for their guidance, constructive feedback, and the knowledge they generously shared throughout the course of this work.

Finally, I would like to thank TU Wien, especially the Faculty of Informatics, for providing an excellent academic environment and the opportunity to pursue my studies.

Kurzfassung

Die Integration von Conceptual Modeling (CM) und Machine Learning (ML) hat ein wachsendes Forschungsfeld hervorgebracht, das als Machine Learning for Conceptual Modeling (ML4CM) bekannt ist. In diesem werden ML-Techniken zur Unterstützung von Modellierungsaufgaben wie Klassifizierung, Vervollständigung oder Reparatur eingesetzt. Ein entscheidender Faktor bei diesen Anwendungen ist die Transformation konzeptueller Modelle in ML-kompatible Repräsentationen, sogenannte Encodings. Dabei gibt es eine Vielzahl an Encoding-Strategien, die je nach Anwendungsfall auf unterschiedliche Informationsquellen innerhalb der Modelle zurückgreifen. Bestehende Arbeiten im ML4CM Bereich neigen jedoch dazu, Encodings als fixiert zu behandlen und konzentrieren sich vorwiegend auf die Optimierung von ML-Algorithmen und deren Hyperparametern. Folglich werden Encodings und deren interne Konfigurierbarkeit kaum systematisch evaluiert, was die Auswahl und Anpassung geeigneter Encodings für spezifische Aufgaben erschwert.

Diese Arbeit schließt diese Lücke durch die Entwicklung und Evaluierung konfigurierbarer semantischer Encodings für konzeptuelle Modelle. Insbesondere wird untersucht, wie sich semantische Informationen systematisch extrahieren und in ML-kompatible Repräsentationen überführen lassen. Dazu wird die Methodik der Design Science Research angewandt und das bestehende CM2ML Framework um einen ArchiMate Parser sowie vier Encoder erweitert: Bag-of-Words (BoW), Term Frequency (TF), Embeddings und Triples. Jeder Encoder erfasst unterschiedliche semantische Aspekte und unterstützt umfangreiche Konfigurationsmöglichkeiten, um Experimente und aufgabenspezifische Anpassungen zu ermöglichen. Darüber hinaus lassen sich alle Encodings im Framework interaktiv visualisieren, wodurch sich Parametereffekte in Echtzeit nachvollziehen und die codierten Merkmale den Bestandteilen im Modell zuordnen lassen.

Zur Evaluation kombiniert die Arbeit einen qualitativen Vergleich anhand definierter Kriterien mit einer quantitativen Analyse in zwei repräsentativen ML-Aufgaben. Die erste Aufgabe, Dummy-Klassifizierung, nutzt TF-Encodings um Dummy Views von gültigen Views zu unterscheiden und untersucht die Auswirkung gängiger NLP-Parameter. Die zweite Aufgabe, Node-Klassifizierung, zielt auf die Vorhersage von Elementtypen basierend auf lokalem Kontext und verwendet Triple-Encodings, angereichert mit Embeddings für Namen und One-Hot-Vektoren für Typen. Die Ergebnisse zeigen die Eignung der Encodings für spezifische ML4CM-Aufgaben und belegen, dass bestimmte Konfigurationen die Modellleistung signifikant beeinflussen können.

Abstract

The integration of Conceptual Modeling (CM) and Machine Learning (ML) has given rise to a growing research field known as Machine Learning for Conceptual Modeling (ML4CM), where ML techniques are applied to support modeling tasks such as classification, completion, or repair. A crucial factor in these applications is the transformation of conceptual models into ML-compatible representations, called encodings. A wide variety of encoding strategies exist that draw on different information sources within conceptual models, depending on the specific use case. However, existing ML4CM studies tend to treat encodings as fixed and focus predominantly on tuning ML algorithms or hyperparameters. Consequently, encoding strategies and their internal configuration options receive limited scrutiny during evaluation, making it difficult for researchers and practitioners to select and adapt optimal encodings for specific tasks.

This thesis addresses this gap by developing and evaluating a set of configurable semantic encodings for conceptual models. Specifically, it investigates how semantic information (e.g. names, types, contextual relationships) within models can be systematically extracted and transformed into ML-compatible representations. The work adopts the Design Science Research methodology and extends the CM2ML framework with an ArchiMate parser and four semantic encoders: Bag-of-Words (BoW), Term Frequency (TF), Embeddings, and Triples. Each encoder captures distinct semantic aspects and supports extensive configurability to enable experimentation and task-specific adaptation. Furthermore, all encodings can be interactively visualized within the framework, offering real-time insight into parameter effects and traceability to link encoded features back to their source model elements.

To evaluate the proposed encodings, the thesis combines a qualitative comparison based on defined criteria with a quantitative assessment through two representative ML tasks. The first task, dummy classification, employs TF encodings to distinguish dummy views from valid ones and explores the impact of common NLP parameters and weighting schemes. The second task, node classification, aims to predict element types based on local context, using triple encodings enriched with word embeddings for element names and one-hot vectors for types. The results demonstrate the suitability of the encodings for specific ML4CM tasks and that certain encoding configurations can have a substantial influence on model performance.

Contents

CHAPTER 1

Introduction

Conceptual Modeling (CM) is a foundational technique used in various scientific disciplines to create abstract, often graphical, representations of real-world domain objects [RAB+15, MBWM24]. A conceptual model provides a structured way to describe entities, relationships, and constraints within a specific domain, aiding in the analysis, design, and communication of complex systems [DLPS18]. CM is widely used, for example, in information system design [Oli07], data modeling [Tha13], and business process modeling [RRIG09]. Models also play an essential role in the field of Model-Driven Engineering (MDE), where they serve as central artifacts throughout the entire software development lifecycle [BCW17], supporting tasks such as code generation, validation, and system integration. Models in this context are typically expressed using elements of formal modeling languages such as the Unified Modeling Language (UML) [MRRR02] or ArchiMate [LPJ10], which, in turn, conform to platform-specific metamodels such as Ecore [SBMP08] or ADOxx [FRK12].

Parallel to advances in CM, Machine Learning (ML) has gained significant traction in various fields due to its ability to identify patterns, classify data, and make predictions from large datasets without explicit rule-based programming [JM15]. ML techniques have been successfully applied in areas such as Natural Language Processing (NLP) [OMK20] or image recognition [PK17]. As both fields continue to evolve, their convergence has led to new interdisciplinary applications and research opportunities.

Recently, the intersection of CM and Artificial Intelligence (AI), particularly ML, has become a growing research focus [BAR23, MS21], leading to three different research streams: *(i)* Conceptual Modeling for Machine Learning (CM4ML), where CM techniques are used to enhance ML applications, *(ii)* Machine Learning for Conceptual Modeling (ML4CM), where ML techniques are applied to improve CM tasks, and *(iii)* CM & ML, where existing solutions from both fields are combined to solve specific problems. This thesis is focused on ML4CM research, specifically in leveraging ML techniques to assist with modeling-related tasks such as model classification [NRR+19] (categorizing

models into predefined classes), model completion [BCG+21, WSS22] (predicting missing elements in incomplete models), or model repair [BHI+20] (identifying and correcting inconsistencies within models).

Although ML has shown promising results in these applications, the effectiveness of ML is highly dependent on the way conceptual models are encoded for learning tasks. Since ML algorithms require structured machine-readable input (e.g., feature vectors), conceptual models must first be transformed into appropriate representations, known as *encodings*. An encoding defines how information from different information sources within conceptual models is represented, affecting how well an ML algorithm can learn from and generalize over the data. A recent literature review [AGPB23] categorizes existing ML4CM encodings into two types: *structural encodings*, focusing on graph-based features, capturing model topology and relationships, and *semantic encodings*, which incorporate textual and semantic content such as lexical terms, metamodel semantics, and ontologies.

The review also reveals inconsistencies in existing encoding strategies, partly due to the diversity of ML tasks and application domains, leading to heterogeneous and often ad hoc approaches. The evident lack of standardized transformation approaches limits the generalizability of models and results in crucial knowledge being overlooked during training. This highlights the need for a systematic approach to encoding, particularly in capturing semantic aspects. This gap forms the underlying motivation and constitutes the main problem addressed in this thesis, namely, how to effectively encode semantic information in conceptual models for ML applications.

1.1 Methodology

This thesis adopts the **Design Science Research (DSR)** paradigm as its methodological foundation. DSR is well suited for research that aims to create and evaluate purposeful artifacts in response to identified problems, especially in engineering disciplines such as software engineering and information systems [HMPR04, PTRC07]. The process follows an iterative cycle of problem identification, artifact construction, evaluation, and refinement. In the context of this thesis, the artifacts are a set of semantic encoders integrated into the Conceptual Models to Machine Learning (CM2ML) framework, designed to transform conceptual models into ML-compatible representations. In the following, we describe the key steps of the applied DSR process.

1.1.1 Problem Identification

This research is motivated by the observation that the encoding of conceptual models for ML is often inconsistent, underexplored, and underutilized. Although there are various encoding approaches, they are typically adjusted to specific tasks, and different configurations or other encodings are rarely evaluated. In many cases, studies focus on comparing ML models but neglect to examine the effects of encoding strategies

themselves. As a result, the full potential of information embedded in conceptual models is not utilized effectively.

This thesis addresses this gap by focusing on semantic encodings, i.e., encodings that extract and transform lexical content from conceptual models. A preliminary review of the literature and discussions with conceptual modeling experts (thesis supervisors) helped define a set of concrete research questions (see Section 1.2) that guide this work.

The main goal is to develop a set of semantic encoders that transform conceptual models to ML-compatible representations. In addition, the semantic encoders should be highly configurable and generic to support experimentation and broader applicability. For explainability and manual inspection, the encodings should be visualizable and provide traceability to trace encoded information back to originating model elements.

1.1.2 Solution Design

The solution was implemented in two main stages. First, a parser for ArchiMate models was developed and integrated into the CM2ML framework. It supports two commonly used Extensible Markup Language (XML) formats and transforms models into the framework's graph-based Intermediate Representation (IR), which serves as the basis for subsequent encoding steps.

In the second stage, four semantic encoders were implemented: Bag-of-Words (BoW), Term Frequency (TF), Embeddings, and Triples. Each encoder captures different aspects of model semantics, including lexical, structural, distributional, and contextual information. The encoders were iteratively extended to support a variety of configuration parameters and to ensure compatibility with all supported execution environments of CM2ML, including the interactive visualizer. Furthermore, a common term extractor was implemented, serving as a reusable preprocessing component for nearly all semantic encoders.

1.1.3 Evaluation

The evaluation assessed the developed artifacts from two complementary perspectives. First, *functional validation* was performed through automated unit tests to ensure the correctness of the implemented parser and encoders. All components were tested on a real-world dataset of ArchiMate models to verify compatibility.

Second, a *qualitative comparative analysis* was performed to assess the characteristics and suitability of each encoder. A set of comparative criteria was defined to systematically describe and contrast the encoders. Additionally, two encoders were evaluated in the context of two illustrative ML tasks to demonstrate their practical effectiveness and to explore the effects of selected parameter configurations on encoding behavior and ML performance.

1.2 Research Questions

This thesis is guided by two Research Questions (RQs) that address the feasibility and characteristics of semantic encodings for conceptual models in the context of ML.

> **RQ1**: *To what extent can semantic information in conceptual models be extracted and encoded into suitable representations for machine learning applications?*

This question examines the feasibility of capturing different forms of semantic information and transforming them into ML-compatible representations. It is addressed through the implementation of four semantic encoders that reflect different encoding strategies. The results are discussed in Section 4.6, particularly in relation to the types of semantics each encoder captures and how they contribute to ML applicability.

> **RQ2**: *How do semantic encoding strategies compare in terms of their ability to preserve information and support machine learning tasks?*

This question focuses on comparing the implemented encoders in terms of different characteristics and on investigating their strengths and weaknesses for ML applications. It is addressed through a comparative analysis presented in Section 4.6, followed by an evaluation in Chapter 5, where selected encoding configurations are assessed in two representative ML tasks.

1.3 Scope

This thesis is situated within the field of ML4CM, specifically addressing the challenge of encoding conceptual models for ML applications. The work does not aim to develop new ML algorithms or optimize existing models for specific CM tasks. Instead, it is concerned with the encoding process itself, a necessary prerequisite to any ML-based pipeline. The aim is to enable conceptual models to be effectively used in ML applications and provide configurable transformation mechanisms for experimentation.

The research concentrates on semantic encodings, which are designed to capture the meaning of model elements by leveraging lexical terms, contextual usage, and metamodel-level semantics. These encodings contrast with structural encodings, which focus primarily on graph-based topology and connectivity of elements. Structural encodings have been addressed in prior work, notably by Müller [Mül24], and are not the focus of this thesis.

Furthermore, this work mainly targets ArchiMate models, a modeling language widely used in enterprise architecture. Although the semantic encoders are implemented generically within the CM2ML framework and could, in principle, be applied to other modeling languages such as UML or Ecore, this thesis limits its scope to ArchiMate.

Finally, due to the extensive configuration space introduced by the implemented encoder parameters, a full combinatorial evaluation of all parameter settings is out of scope.

Instead, selected configurations were varied in the evaluation to illustrate their effects on encoder behavior and ML performance.

1.4 Contributions

This thesis makes several contributions at the intersection of CM and ML, focusing on the encoding of semantic information in conceptual models and the evaluation of such encodings.

First, the **CM2ML framework** was extended to support a new modeling language and multiple encoding strategies. A dedicated **ArchiMate parser** was developed, supporting two common XML serialization formats and offering configurable parameters (e.g., for processing relationships/views or filtering certain types). The extension of the framework also includes a set of reusable components, such as a shared term extractor and a utility module that provides common NLP functionality (e.g., tokenization, embedding retrieval, word similarity).

Second, the thesis introduces a set of **semantic encoders**, implemented as part of CM2ML and grounded in encoding strategies from the existing literature. These include: *(i)* a BoW encoder for token- and sentence-based representations, *(ii)* a TF encoder supporting uni-, bi-, and n-gram variants with flexible aggregation, *(iii)* an embeddings encoder using pre-trained word vectors with optional pooling, and *(iv)* a triples encoder that captures relational structure, enriched with type encodings and embeddings. All encoders are highly configurable to support systematic experimentation and customization.

Third, the thesis presents both a **qualitative and quantitative comparison of semantic encodings**. The qualitative comparison applies a defined set of criteria (e.g., granularity, structure, interpretability, ML suitability) to assess the strengths and trade-offs of each encoder type. Quantitatively, two representative ML tasks are implemented to evaluate and compare selected encoding configurations: *(i)* dummy view classification using TF encodings, and *(ii)* node classification using triple encodings with embeddings and one-hot vectors for types. These tasks are evaluated on a (labeled) dataset of ArchiMate models to demonstrate how specific parameters influence learning performance and provide a blueprint for future benchmarking.

Finally, the thesis **answers two research questions**, showing how semantic information in conceptual models can be systematically extracted and encoded into ML-compatible formats, and how different encoding strategies compare in their ability to preserve relevant information and support learning tasks.

1.5 Thesis Structure

The thesis is organized as follows:

- **Chapter 2** introduces relevant background and related work, including core concepts from CM, the ArchiMate language, and ML. It also reviews the related literature on semantic encoding strategies, establishing a foundation for the encoder designs and evaluations presented in later chapters.
- **Chapter 3** presents the CM2ML framework, detailing its usage modes, technical architecture, and the newly developed ArchiMate parser. This chapter establishes the implementation basis for the semantic encoders.
- **Chapter 4** describes the implemented semantic encoders. It introduces the shared term extractor, followed by a presentation of each encoder with details on implementation, output, parameters, and visualization. The chapter concludes with a qualitative comparison of the encoders using defined evaluation criteria.
- **Chapter 5** reports the evaluation of selected encoders. It introduces the experimental setup and dataset, followed by two evaluation tasks: dummy classification using TF encodings, and node classification using triple encodings. Both tasks quantitatively assess the impact of encoding configurations on ML performance.
- **Chapter 6** concludes the thesis with a summary, answers to the research questions, and a discussion of limitations and future work.

CHAPTER 2

Background & Related Work

This chapter provides the theoretical foundation for the work presented in this thesis. It introduces the key concepts and formalisms to understand the problem domain and the proposed solutions. In addition, it contextualizes the thesis within existing research by reviewing relevant literature.

This chapter is structured as follows. Section 2.1 introduces CM and discusses the ArchiMate language, which forms the basis for the models processed in this thesis. Section 2.2 presents relevant ML principles, with a particular focus on the ML models used in the evaluation of this thesis. Section 2.3 reviews related work in the field of ML4CM, with a focus on semantic encoding strategies.

2.1 Conceptual Modeling

CM is an abstraction technique for representing knowledge about a system, domain, or process using formally defined notation and semantics. Its purpose is to capture relevant information in a way that provides a shared understanding of complex domains and supports communication between stakeholders, analysis, and automation. CM has relevance in various engineering disciplines, such as information system design [Oli07], data modeling [Tha13], or business process management [DRMR13]. CM underpins MDE, a software engineering approach in which models are treated as first-class artifacts throughout the software development lifecycle [BCW17]. In MDE, conceptual models serve not only as documentation, but also as sources for code generation, validation, transformation, and simulation.

A typical conceptual model is composed of entities (or elements), relationships between those entities, attributes that describe their properties, and potentially additional constraints that restrict valid configurations. Models are expressed using formal modeling languages, defined by a metamodel that specifies the allowed constructs, their types, and

relations. The most prevalent metamodel is Ecore from the Eclipse Modeling Framework (EMF) [SBMP08], which provides an entire ecosystem of modeling tools, such as the Object Constraint Language (OCL) [CG12] or the ATLAS Transformation Language (ATL) for model transformations [JAB+06]. Well-known modeling languages include UML for designing software architectures [OE20], Business Process Model and Notation (BPMN) for process modeling [CT12], and Entity-Relationship (ER) diagrams for data modeling [Che76].

In the remainder of this section, we discuss ArchiMate, a widely used modeling language to represent Enterprise Architecture (EA).

2.1.1 ArchiMate

In the context of this thesis, ArchiMate serves as the main modeling language for the parser and semantic encoders developed and evaluated as part of this work. ArchiMate is a modeling language specifically designed for EA, providing a standardized notation to represent and analyze alignment between business processes, application systems, and technological infrastructure.

ArchiMate originated from a research project and has since been adopted and maintained by the ArchiMate Forum from The Open Group[1], a global consortium focused on developing open standards. Over the years, several extensions have been made to ArchiMate to accommodate evolving modeling needs. Extensions include the addition of motivation modeling, implementation and migration concepts, and physical elements. As of now, the most recent version is ArchiMate 3.2 [Gro], which is also used in this thesis.

The goal of ArchiMate is to provide a coherent and integrated modeling language for describing different aspects of EA [LPJ10]. It supports the specification of multiple interrelated architectural domains (e.g., business, application, technology) and allows the creation of viewpoint-specific representations for various stakeholders. Through its layered and aspect-oriented structure, ArchiMate enables architects to address both strategic and operational concerns across business and IT systems [Lan17].

The full ArchiMate framework is shown in Figure 2.1 and is organized along two dimensions: layers and aspects. The language defines six **layers** (Strategy, Business, Application, Technology, Physical, and Implementation & Migration), each representing a different level of abstraction within the enterprise. These layers are intersected by **aspects** that classify elements based on layer-independent characteristics: *Active Structure* (structural elements that display behavior), *Behavior* (activities, functions, and processes), and *Passive Structure* (objects on which behavior is performed). Composite elements are not directly tied to a single aspect and may combine two or more aspects. An additional *Motivation* aspect spans over all layers and captures the reasoning behind architectural decisions.

[1] https://www.opengroup.org/archimate-forum/ (Accessed: 28.05.2025)

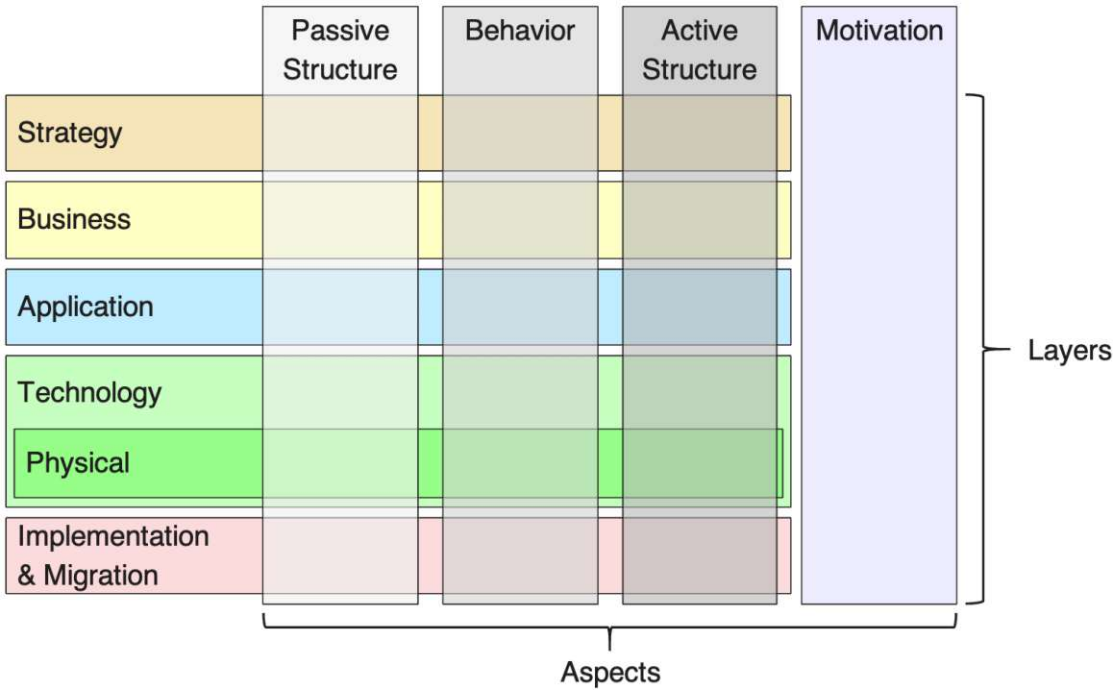


Figure 2.1: ArchiMate Full Framework (from [Gro])

An example ArchiMate model is shown in Figure 2.2. The model is an excerpt from the ArchiSurance case study [JBQ12], providing a layered view of various business processes to handle insurance claims. Elements are colored by layer: yellow for business, blue for application, and green for technology. Arrows between elements indicate relationships such as *Triggering*, *Assignment*, or *Access*. The view shows different business actors (e.g., *Customer*) assigned to business roles (e.g., *Insurant*), which are being served business services (e.g., *Claims Registrations*) that are realized by business processes (e.g., *Register*). In the lower layer, business processes are supported by application services (e.g., *Customer Data Management*) that are realized by application components (e.g., *CRM System*). The lowest layer shows how technology services support the application layer through technology services (e.g., *Database Management*), system software (e.g., *Database Management System*), and devices (e.g., *Blade System*). Information can also be represented on different abstraction levels, for example, as artifacts (e.g., *Customer Database Tables*), data objects (e.g., *Customer Data*), and business objects (e.g., *Customer Information*). Note that the example only shows a single view and, in practice, multiple viewpoints are used to represent an EA.

ArchiMate models are typically stored in XML files, however, the XML schema is often tool-specific. In this thesis, we focus on two XML formats: *(i)* the Archi Tool Storage Format (file extension `.archimate`), used by the open-source modeling tool Archi[2], and

[2] `https://www.archimatetool.com/` (Accessed: 28.05.2025)

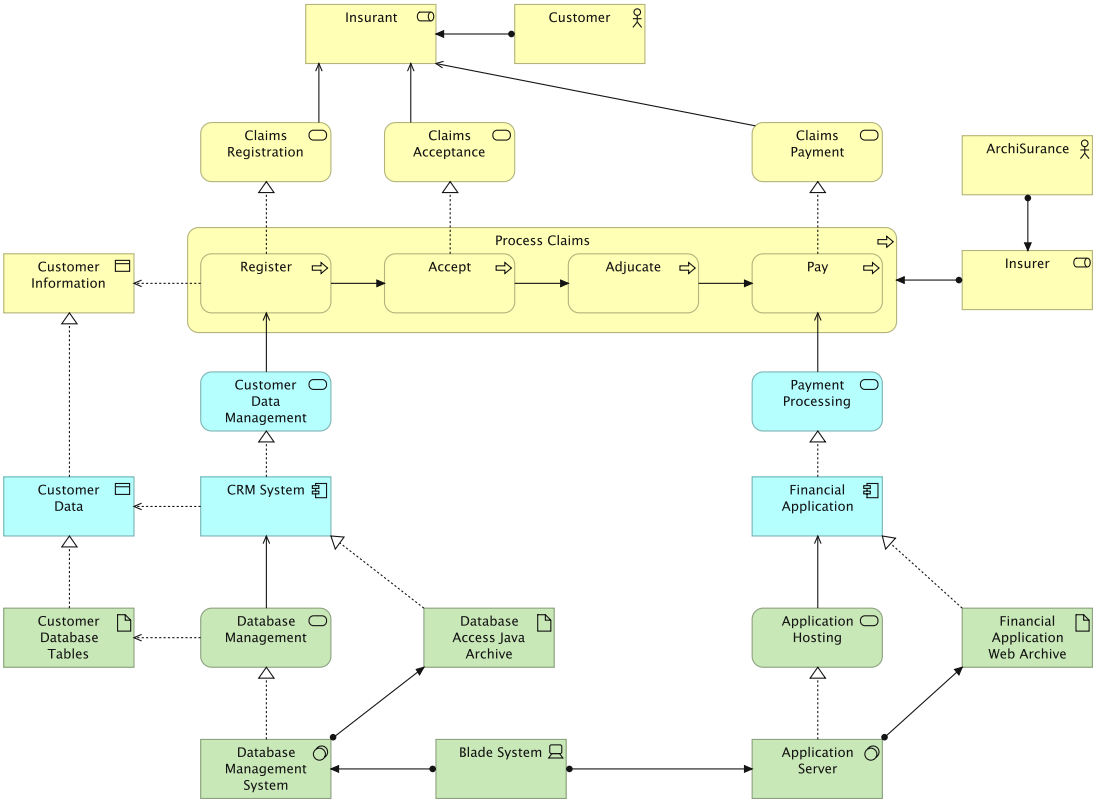


Figure 2.2: ArchiMate Example Model (excerpt from [JBQ12])

(ii) the Open Group ArchiMate Model Exchange Format[3] (file extension `.xml`), which provides a standardized exchange format for interoperability between modeling tools.

2.2 Machine Learning

ML has become a central paradigm in data-driven problem solving and has been increasingly applied to CM tasks in recent years. To understand how semantic encodings contribute to these tasks, it is essential to understand the basic principles and assumptions that underlie ML.

ML is a subset of AI and refers to the study of algorithms that enable computers to improve their performance on a given task through experience, typically by learning from data rather than relying on explicitly defined rules [Mit97]. Unlike traditional programming, ML models discover patterns in data representations, making the choice and quality of those representations, known as *encodings*, a critical factor in their effectiveness.

[3] `https://www.opengroup.org/xsd/archimate/` (Accessed: 28.05.2025)

ML methods are commonly grouped into three categories: *supervised learning*, where models are trained on labeled examples to perform tasks such as classification or regression, *unsupervised learning*, which infers patterns from unlabeled data, e.g., through clustering or dimensionality reduction, and *reinforcement learning*, where agents learn optimal behavior through feedback signals. Supervised and unsupervised methods are directly relevant to the evaluation of semantic encodings, whereas reinforcement learning plays only a minor role in this thesis.

The remainder of this section introduces the ML models used in the evaluation of this thesis.

2.2.1 Machine Learning Models

This subsection provides a high-level overview of the ML algorithms used in the evaluation presented in Chapter 5. The selected models are commonly used in traditional ML approaches and cover a range of learning paradigms, from linear and probabilistic classifiers to non-linear ensemble methods and basic Neural Networks (NNs).

For each model, the underlying principles, characteristics, advantages, and limitations are discussed. This background should help in understanding evaluation results and interpretation of performance differences between models and encoding strategies.

Logistic Regression

Logistic Regression (LR) is a supervised learning algorithm used for binary classification and, by extension, for multi-class classification problems. It is one of the most widely used classification models due to its simplicity and interpretability.

The algorithm models the probability that a given input belongs to a certain class by applying the logistic (sigmoid) function to a linear combination of input features. Formally, the predicted probability is given by:

$$P(y = 1 \mid x) = \frac{1}{1 + e^{(wx+b)}} \tag{2.1}$$

where x is the input vector, w is the weight vector and b the bias term. During training, the model minimizes the logistic loss (cross-entropy loss) using optimization techniques such as gradient descent to learn feature weights. For multiclass classification, it is extended using the softmax function.

LR is a linear, parametric model that is computationally efficient and works well for linearly separable data. However, it struggles with non-linear relationships and is sensitive to outliers, as the magnitude of feature values influences both the convergence speed and the learned coefficients.

Support Vector Machines

Support Vector Machines (SVMs) are supervised learning algorithms mainly used for classification, and to some extent for regression. The idea is to find a decision boundary (hyperplane) that maximally separates the classes in the feature space. For linearly separable data, this hyperplane is chosen to maximize the margin (i.e., the distance between the hyperplane and the nearest data points from each class).

For linearly separable data, the hard-margin SVM is used, which solves the following optimization problem:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{such that} \quad y_i(wx_i - b) \geq 1, \quad \text{for all } i = 1, \ldots, n \tag{2.2}$$

For non-linearly separable data, soft-margin SVMs can be used which optimize using the hinge loss function, controlled via a regularization parameter C. In addition, SVMs can also use kernels (e.g., linear, polynomial, radial basis function) to project input data into a higher-dimensional space to work with data that cannot be separated by a hyperplane in its original space (i.e., find non-linear decision boundaries).

In contrast to LR, SVMs are deterministic and can be linear or non-linear depending on the chosen kernel. They can model complex decision boundaries and are robust against overfitting (especially in high-dimensional spaces) with proper regularization, but they can be difficult to tune. SVMs also scale poorly with very large datasets, however, the final model is typically sparse, depending only on a subset of the training data.

Random Forests

Random Forests (RFs) are supervised learning algorithms suitable for both classification and regression tasks. They are an ensemble method using multiple decision trees, each trained on a random subset (bootstrap) of the training data and features. The final prediction is made by aggregating the outputs of individual trees (majority voting for classification and averaging for regression). This combination of bagging (bootstrap aggregation) and feature randomness helps prevent overfitting and improves accuracy.

RFs are non-linear, non-parametric, and deterministic. They are not sensitive to feature scaling, as decision trees rely on relative feature comparisons and split on thresholds, not absolute distances. Training is computationally expensive and memory-intensive with a large number of trees, but this can often be parallelized and inference is fast. RFs generally work well with minimal parameter tuning and can deal with high-dimensional or missing data.

k-Nearest Neighbors

The K-Nearest Neighbors (KNN) algorithm is a simple instance-based method, mainly used for classification but also applicable to regression tasks. It relies on similarity

between data points and makes predictions based on the closest examples in the training data. In particular, the algorithm stores all training data and, at prediction time, it finds the k closest instances to a given input using a distance metric (e.g., Euclidean or cosine). For classification, the predicted label is the most frequent one among the k neighbors (majority class). For regression, the prediction is the average of their target values. There is no explicit training phase, and all computation is performed at prediction time, making it a lazy learning algorithm.

KNN is non-linear, non-parametric, and deterministic unless ties are broken randomly. It is highly sensitive to feature scaling, since distance calculations are affected by the magnitude of features. Therefore, normalization or standardization is typically required for good performance. It is a lazy learning algorithm with no explicit training phase, but inference can be computationally expensive for large datasets, as the distance to all training points must be computed at prediction time. Tuning can also be difficult with the choice of an appropriate distance metric and the tuning of the parameter k, the number of neighbors considered, which strongly influences model bias and variance.

Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a type of Feedforward Neural Network (FNN) used for classification and regression tasks. It is composed of an input layer, one or more hidden layers, and an output layer. Each neuron in a layer is connected to all neurons in the subsequent layer through weighted edges. The model learns by adjusting these weights during training using backpropagation and stochastic gradient descent to minimize a loss function (e.g., cross-entropy for classification). MLP can model complex decision boundaries through non-linear activation functions (e.g., ReLU, sigmoid) between layers.

Multi-Layer Perceptrons (MLPs) are non-linear, parametric, and typically deterministic unless dropout or stochastic components are used. They are sensitive to feature scaling, therefore, normalization of inputs is strongly recommended. Training can be computationally expensive, especially with large networks or datasets, but inference is generally fast once the model is trained. MLPs require large amounts of data to generalize well and are prone to overfitting on small datasets. They are flexible, but several hyperparameters have to be carefully tuned to achieve optimal performance, such as the number of hidden layers, the number of neurons per layer, activation functions, or learning rate.

2.3 Related Work

The application of ML techniques to conceptual models has gained increasing attention in recent years, giving rise to the field of ML4CM. In this context, conceptual models are first transformed into ML-compatible representations, i.e., encodings, which are then used as input to train ML algorithms for specific tasks, such as classification, clustering, recommendation, or anomaly detection. The choice of encoding directly constrains the type and amount of information from which an ML model can learn and generalize.

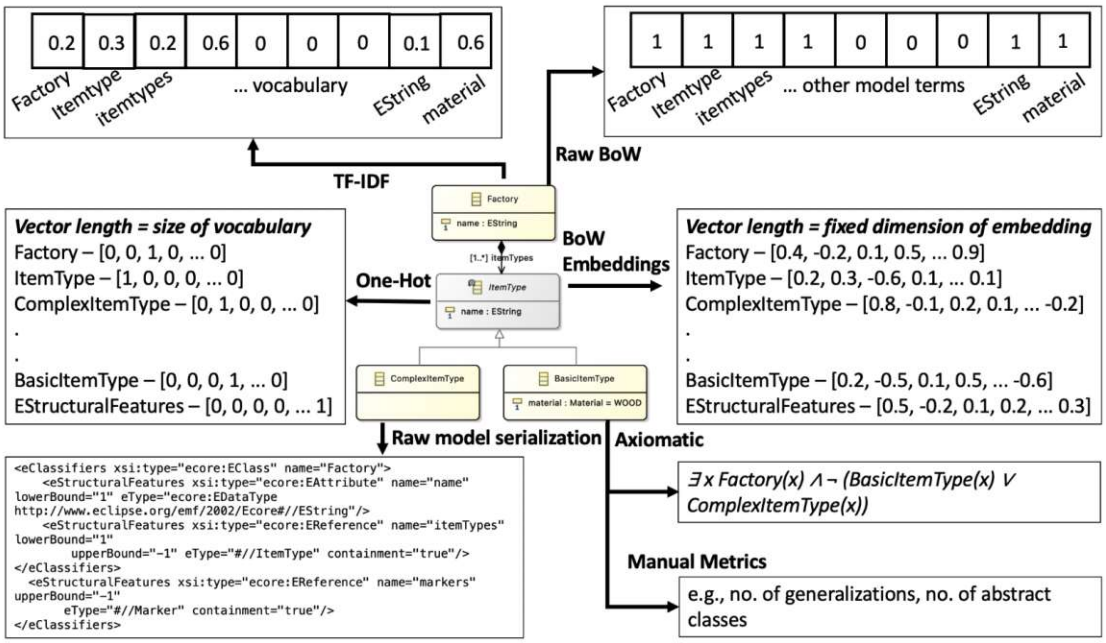


Figure 2.3: Semantic Encodings in ML4CM Applications (from [AGPB23])

Thus, a challenge in ML4CM is determining which sources of information from conceptual models should be included in the encodings and how this information should be encoded to support effective learning. The work in [AGPB23] addresses these challenges by conducting a systematic literature review of encoding strategies for conceptual models in ML-based applications. The authors distinguish between *structural* information (e.g., graph-based features) and *semantic* information (e.g., lexical terms, metamodel semantics, or ontological semantics) as main sources of information. They also highlight a lack of systematic comparison between different encodings and note that many ML4CM studies focus narrowly on specific tasks, using isolated, custom encoding variants. The different semantic encoding variants commonly used in ML4CM applications are shown in Figure 2.3.

In this section, we build on the previous work and provide a focused review of semantic encoding approaches as relevant to this thesis. This section structures the literature into three categories: term frequency encodings (Section 2.3.1), embedding-based encodings (Section 2.3.2), and other semantic encoding strategies (Section 2.3.3).

Complementary to task-specific works which are reviewed in the remaining sections, several recent studies assess ML4CM methods more holistically. For example, the authors in [LRCR22] built a framework for model classification that systematically evaluates different ML models in combination with multiple encoding variants. The work in [MIL+24] systematically reviews the use of ML to solve MDE problems, discussing trends, remaining research gaps, and open challenges. The authors in [ARR25] extend common ML-assisted modeling tasks with eXplainable Artificial Intelligence (XAI) to

aid in understanding and interpretation of results.

2.3.1 Term Frequency Encodings

A large body of prior work has relied on TF encodings to transform models into feature-based vector representations. These encodings typically follow a BoW approach, where models are represented as sets of terms, extracted from element names, types, or other attributes, which are encoded as sparse vectors containing the frequency of each term. When limited to individual terms, such representations are referred to as uni-grams. Extensions include bi-grams and n-grams, which typically capture short structural sequences by combining connected model elements. These extensions allow the encoding to include not just lexical, but also contextual and structural information.

To improve term quality and reduce noise, standard NLP techniques such as tokenization, stemming, lemmatization, stop-word removal, and synonym detection are commonly applied. In many cases, Term Frequency-Inverse Document Frequency (TF-IDF) weighting is used to scale term frequencies by their Inverse Document Frequency (IDF), emphasizing terms that rarely occur in models. TF-based encodings have the advantage of being interpretable, are often simple to compute, and are compatible with a variety of unsupervised and supervised ML algorithms.

The earliest application of TF encodings in CM, is the work in [BCVvdB16], which proposes bi-grams of connected elements combined with a type-based weighting scheme. These representations can be used for statistical comparison of models in large-scale respositories, e.g., clustering, outlier detection, or identification of representative models. The authors apply standard distance metrics (e.g., Manhattan) along with k-means and hierarchical clustering, offering improvements to traditional pairwise comparisons.

A closely related approach is presented in [BCvdB16], where uni-gram encodings of element names, enriched by basic NLP techniques, are used. The resulting vectors are used for Hierarchical Agglomerative Clustering (HAC) with cosine similarity and results are visualized using dendrograms to support comparative analysis of large metamodel repositories.

The use of n-gram encodings for models is further explored in [BC17], where uni-, bi-, and tri-grams of connected elements are extracted to better capture structural context. Token normalization and compound-word similarity further improve clustering performance. HAC with cosine similarity is used again for grouping similar models.

While the above works focus on unsupervised analysis, TF encodings have also been successfully applied in supervised learning contexts. The AURORA framework, introduced in [NRR+19], uses uni-, bi-, and n-grams enriched with properties such as typing and cardinality. These features serve as input to a FNN trained to classify metamodels into predefined categories. A more detailed evaluation of this approach is provided in a follow-up work [NDRI+21].

Clone detection has also benefited from frequency-based encodings. In [BCVDB19], n-grams are used in conjunction with subtree extraction and modified tree edit distance (APTED) to identify similar metamodel fragments. The combination of structural and lexical features together with NLP techniques such as synonym detection (e.g., via WordNet) and typo correction improve the precision of the method.

A lightweight and unsupervised alternative for automated classification and clustering of metamodels is explored in [RRR+21]. Here, Apache Lucene is used to index lexical terms and apply TF-IDF scoring. Despite its simplicity and no training being required, the method achieves competitive results in classification and clustering.

Another advancement is MemoCNN introduced in [NDRP+21]. It builds on traditional TF-IDF encoding of n-grams and reshapes the resulting feature vectors into two-dimensional matrices, allowing metamodels to be processed by Convolutional Neural Networks (CNNs). This deep learning-based model outperforms AURORA and demonstrates the potential of frequency-based encodings in combination with neural architectures.

Lastly, the work in [LC22] presents MAR, a search engine for models. Their approach focuses on model retrieval rather than classification. Models are encoded as Bag-of-Paths (BoP) of varying length and indexed using Apache HBase with frequency statistics. Although no learning algorithm is used, the frequency statistics enable efficient keyword-based and query-by-example search functionalities.

2.3.2 Embedding-based Encodings

Another common approach to encode semantic information in conceptual models is through embedding-based encodings. Embeddings can capture semantic similarity, contextual usage, or structural properties in a format compatible with ML algorithms. Single terms or larger model fragments are mapped into dense, real-valued vector spaces, typically using pre-trained language models or neural embedding techniques. Unlike sparse one-hot or frequency vectors, embeddings preserve semantic proximity by placing similar concepts closer together in the vector space.

Embedding approaches in CM typically fall into three categories. First, static word embeddings such as Word2Vec [MSC+13] or GloVe [PSM14] are used to represent tokens extracted from element names or other attributes. Second, contextual language models (e.g., BERT [DCLT19] or RoBERTa [LOG+19]) generate embeddings based on the surrounding context of model fragments. Third, structure-aware neural networks (e.g., Graph Neural Networks (GNNs) [WPC+20] or Long Short-Term Memory (LSTM) [HS97] NNs) derive embeddings by learning directly from model graphs. All three representations are well suited for similarity-based retrieval, recommendation, classification, and transformation tasks.

An example of embedding-based recommendation is the system introduced in [GG21], which supports semantic autocompletion of BPMN models. The method slices models

into process fragments, encodes them using the Universal Sentence Encoder [CYK+18], and retrieves similar model sequences based on cosine similarity for recommendation. The embeddings allow the system to identify semantically similar model fragments even when labels differ lexically.

Another example with a recommendation use case is the work in [BCG+21], which presents an NLP-based architecture for semantic autocompletion of partial domain models. Their system integrates both general-purpose (GloVe) and project-specific contextual embeddings to suggest candidate elements based on vector similarity. This allows for personalized context-aware model suggestions that improve over time.

A similar goal is pursued in [WSS22], where the RoBERTa language model is used for concept recommendation. Metamodels are represented as hierarchical tree structures and serialized into token sequences through model-to-text transformations. Then, a RoBERTa model is trained on these sequences using masked language modeling to predict masked tokens based on context, enabling context-sensitive recommendations during design time.

The application of state-of-the-art NLP techniques to improve partial model-to-model transformations is explored in [DS22]. BERT-based embeddings with Conditional Random Field (CRF) sequence tagging and dependency parsing are used to extract semantic relations (e.g., verb/noun phrases, conjunctive statements, acronyms) from UML and BPMN models. Embeddings guide the transformation process, enabling more accurate and context-aware mappings between source and target models. Additional tasks such as acronym detection are handled using ML classifiers with morphological and contextual features.

Another work focusing on model transformations is [BCLG22], which proposes a generic encoder–decoder architecture [CvMG+14] based on LSTM NNs to automate heterogeneous model transformations. Models are encoded as trees and embedded using Tree-Long Short-Term Memorys (LSTMs) into fixed-size vectors. An attention-based LSTM decoder then learns to generate the target models from examples.

Graph embeddings are used in [LC21] to evaluate the realism of synthetic models generated through model generators. Real and synthetic models are represented as labeled, directed graphs, and encoded into graph embeddings through GNN-based message passing and attention-based aggregation. A GNN classifier is trained to distinguish real from synthetic models.

Another GNN-based representation learning approach is used in [AGB23] to automate the ontological stereotyping of UML class diagrams using OntoUML [GWAG15] models. A conceptual model is transformed into a Conceptual Knowledge Graph (CKG), where textual features are embedded using a domain-specific GloVe model, which is combined with meta-properties and ontological stereotypes for node-level representations. The graphs are input to a GNN, which learns the embeddings for stereotype prediction. The approach has been further developed in [AB24], which uses the CKG to create BoP encodings that incorporate node-specific contextual information.

In the context of bug localization, [LOZ+21] proposes IdentiBug, a deep learning-based tool that predicts links between bug reports and model elements in UML class diagrams. They combine Word2Vec embeddings for textual encoding and GraphSAGE [HYL17] for structural embeddings, framing the task as link prediction between model and bug nodes.

Finally, there has also been work on addressing the limitations of general-purpose embeddings in [LDC23]. They present WordE4MDE, a set of word embeddings trained on a corpus of modeling texts using GloVe and skip-gram Word2Vec models. In a follow-up work [LDC24], they further extend the modeling-specific embeddings by introducing subword models (FastText), contextual embeddings (BERT), and corpus augmentation using community data from StackOverflow and StackExchange. Embeddings are evaluated directly by plugging them into simple ML tasks (e.g., SVM for classification, k-means for clustering) and they show that WordE4MDE outperfoms standard embeddings in modeling contexts.

2.3.3 Other Semantic Encodings

In addition to TF and embedding-based approaches, several alternative encoding strategies have been applied to support ML applications in CM. These approaches differ significantly in how they represent models and the types of information they include, ranging from logic-based axiomatic encodings and manually crafted metrics to similarity measures and specific reinforcement learning encodings. These encodings are less common than frequency- or embeddings-based encodings as they are often tailored to specific use cases, but still offer interesting perspectives.

For example, the work in [FSG20] proposes a semi-automated framework to assist in diagnosing and repairing faulty OntoUML models. Axiomatic encodings are used as models are transformed into logic-based Alloy specifications, which are used to generate simulations (model instances). These simulations are annotated by domain experts and propositionalized into feature matrices, which are used as training data. The main learning method is a decision tree trained using gain ratio for feature selection and subgroup discovery is used to extract interpretable repair rules from the annotated simulations.

Several studies use manually defined metrics as their encoding approach for conceptual models. Each metric is assigned a value (e.g., how many elements exist in a model) and combined they form a feature vector that can be used in ML applications. For example, the work in [OCvdP13] presents an automated method to identify key classes in reverse-engineered UML class diagrams using supervised ML. Each class is represented as a feature vector based on 11 class-level metrics (e.g., number of operations, coupling measures) that serve as predictors. Another work [BSF+18], encodes intermediate BPMN models using 10 pragmatic features (e.g., edge layout, gateway usage). A FNN is trained on these features to distinguish novices from experts with high accuracy. The authors in [SSS22] encode class diagrams as feature vectors based on 17 structural and quality-related metrics. A deep NN is trained to classify flawed models and guide refactoring.

Another frequently used encoding approach is using similarity as a central representation mechanism. The work in [EGB16] presents a system that recommends UML classes in the design phase, by computing similarities between class elements and using a clustering algorithm for recommendation. Another work [AAZZ22] trains an FNN to predict similarity scores between UML diagrams based on vectorized representations of class names, attributes, and relationships. Finally, the authors in [BRR+16] encode metamodels either textually (via serialization and bi-grams) or structurally (via EMFCompare), which are then transformed into proximity matrices (using different similarity measures) for hierarchical clustering.

CHAPTER 3

CM2ML Framework

This chapter introduces the CM2ML framework, which provides a modular infrastructure to transform conceptual models into different ML-compatible encodings. The framework was originally developed by Müller [Mül24], focusing on structural encodings and UML models. This thesis extends the previous work on the framework with additional ArchiMate support and several semantic encoders. CM2ML is open source and is available as a monorepo on GitHub [1]. In addition, the visualizer is deployed to a website that can be accessed in the browser [2].

This chapter is organized into three sections, structured as follows. Section 3.1 describes the usage modes of CM2ML, demonstrating how it can be used in different environments and use cases. Section 3.2 describes the technical architecture of the framework, including the key components, the underlying IR for processing models, and the technologies used. Section 3.3 focuses on the ArchiMate parser, which was developed as part of this thesis to extend CM2ML with support for ArchiMate models and enable their use in subsequent encoding steps.

3.1 Usage

This section provides an overview of the usage modes supported by the CM2ML framework. CM2ML is designed to be modular and accessible in different application contexts, ranging from interactive encoding exploration of a single model to automated large-scale model processing. To accommodate these needs, the framework offers multiple interfaces: a CLI for scripting and reproducibility, a REpresentational State Transfer (REST) API for integration into web-based environments, a programmatic library for direct use in Node.js applications, and a visual interface for interactive inspection. The following subsections describe each usage mode in detail.

[1] https://github.com/borkdominik/CM2ML
[2] https://cm2ml.vercel.app/

3.1.1 CLI

The CLI of CM2ML provides a scriptable way to process conceptual models and generate encodings. It is the recommended interface for headless execution, automation, and large-scale dataset processing scenarios, and forms the basis of all experiments conducted in this thesis (see Chapter 5). Users specify the input models, the desired encoding method, and configuration parameters directly via command-line arguments. Both, a single model and a batch of models, can be processed.

A typical CLI command consists of the `cm2ml` keyword followed by a specific command name and parameters. An example invocation of a CM2ML CLI command is shown in Listing 3.1. Here, a batch of ArchiMate models is encoded using the BoW encoder. The input is a directory containing models (line 1) and the output is written to a JavaScript Object Notation (JSON) file (line 2). The command also includes flags to control the general behavior (line 3), a parser-specific parameter (line 4), and an encoder-specific parameter (line 5). The help text for this command, displaying all supported parameters, is shown in Figure 3.1.

```
1 cm2ml batch-archimate-bag-of-words ./models/
2         --out output/encoding-output.json
3         --strict --debug --pretty
4         --relationships-as-nodes
5         --include-types
```

Listing 3.1: Example CM2ML CLI Command

All commands follow a consistent pattern in terms of input/output and parameter specification. Each supported modeling language is associated with a set of CLI commands, organized by encoder type and processing mode. Input models must be provided either as individual files (for single-mode commands) or as directories containing multiple models (for batch commands). For example, the command `archimate-embeddings` processes a single ArchiMate file, while `batch-archimate-embeddings` encodes an entire directory of ArchiMate models. With three modeling languages (ArchiMate, Ecore, and UML), five structural encoders, four semantic encoders, and two modes of execution (i.e., single or batch), CM2ML currently provides a total of 54 unique commands. A full list of available commands can be obtained using `cm2ml --help`.

The output is produced in JSON format, by default written to standard output, or optionally saved to a file via the `--out` flag. Each CLI command exposes parser and encoder-specific parameters, together with shared flags to control general behavior. These flags include:

- `--out <file>`: Path to output file (default: stdout).
- `--pretty`: Enable pretty-printed JSON output.
- `--strict`: Enforce strict validation and fail on invalid or unknown input.

- `--debug`: Log debug information and validate results (if `--strict` is also enabled).
- `--help`: Display help message for the selected command.

In batch mode, all files in a directory are processed by default. Thus, the batch commands additionally support parameters for selective processing:

- `--start <n>`: Index of the first model to encode.
- `--limit <n>`: Maximum number of models to encode.
- `--continue-on-error` If true, the execution will continue when encountering an error.

```
Usage:
  $ cm2ml archimate-bag-of-words <inputFile>

Options:
  --debug                                  Log debug information and validate results if strict is also enabled.
 (default: false)
  --strict                                 Fail when encountering unknown or invalid input. (default: false)
  --relationships-as-nodes                 Treat relationships as nodes (default: false)
  --views-as-nodes                         Include views and link their respective elements (default: false)
  --node-whitelist <nodeWhitelist>         Whitelist of ArchiMate element types to include in the model. Root no
des will never be removed. Ignored if empty.
  --node-blacklist <nodeBlacklist>         Blacklist of ArchiMate element types to exclude from the model. Root
nodes will never be removed.
  --edge-whitelist <edgeWhitelist>         Whitelist of edge types to include in the model. Ignored if empty.
  --edge-blacklist <edgeBlacklist>         Blacklist of edge types to exclude in the model.
  --no-include-names                       Do not encode names as terms (default: true)
  --include-types                          Encode types as terms (default: false)
  --included-attributes <includedAttributes>  Additional attributes to encode as terms
  --min-term-length <minTermLength>        Minimum term length (default: 1)
  --max-term-length <maxTermLength>        Maximum length of terms (default: 100)
  --remove-duplicates                      Remove duplicate terms so that each term appears only once per model
(default: false)
  --stop-words <stopWords>                 List of stop words to exclude
  --no-tokenize                            Do not split and clean terms into separate tokens (controlled by term
 delimiters) (default: true)
  --term-delimiters <termDelimiters>       Delimiters used to split tokens
  --no-lowercase                           Do not convert terms to lowercase (default: true)
  --stem                                   Apply stemming to terms (default: false)
  --no-include-node-ids                    Do not include node IDs in output terms (default: true)
  --separate-views                         Separate Views (default: false)
  --encode-as-sentence                     Encode nodes as sentences (default: false)
  --encode-relationships                   Include relationships in sentences (default: false)
  --deduplicate                            Remove duplicate results (default: false)
  --deduplication-data <deduplicationData>  Additional data to check for duplicates. Must be a serialization of d
eduplicated batch output.
  --out <file>                             Path to output file
  --pretty                                 Pretty print JSON output (default: false)
  -h, --help                               Display this message
```

Figure 3.1: Help Text of a CM2ML CLI Command

3.1.2 REST API

CM2ML exposes a REST API that enables remote programmatic access to its encoders. It is stateless, language-agnostic, and built to support dynamic discovery and invocation

of available encoders. This makes the REST interface particularly useful for integrating CM2ML into web applications, modeling tools, or external services that require on-demand encoding capabilities.

The API provides the following endpoints for interaction with plugins:

- `GET /health`: Returns the number of registered plugins in the framework. This endpoint is useful for checking the server status.
- `GET /plugins`: Lists all available encoding plugins, including their parameter metadata (i.e., parameter name, type, description, and default value).
- `POST /plugins/{name}`: Invokes the specified encoder plugin with a user-provided JSON request body. The plugin name corresponds to a parser–encoder combination (e.g., `archimate-term-frequency`).

To invoke an encoder, a `POST` request is sent to the corresponding plugin endpoint. The request body must be a valid JSON object containing at least the key `"input"`, which holds an array of serialized models (in plain-text format, e.g., ArchiMate XML). Plugin-specific parameters can optionally be included as additional top-level keys in the request body. Unlike the CLI, the REST API does not distinguish between single or batch input (a single model or more can simply be passed in the input array).

An example request is shown in Listing 3.2 where two serialized ArchiMate models are processed using the TF encoder with type inclusion enabled (encoder-specific parameter). For readability, the serialized model inputs are simplified as `<model1>` and `<model2>`.

```
curl http://localhost:8080/plugins/archimate-term-frequency
   --header "Content-Type: application/json"
   --request POST
   --data '{
      "input": ["<model1>", "<model2>"],
      "includeTypes": true
   }'
```

Listing 3.2: Example CM2ML REST API Request (using cURL)

The response contains the encoding result in JSON format, consistent with the output of the corresponding CLI command.

In addition to interacting with plugins, the API provides endpoints related to embeddings, as used, e.g., for visualization of the embeddings encoder (see Section 4.4):

- `GET /embedding/{model}/{term}`: Returns the word embedding vector for a specified term from the given model. If the model is not supported, the server responds with status code 400 (Bad Request), and if the term is not found in the model's vocabulary, status 404 (Not Found) is returned.

- `GET /embedding/{model}/{term}/similar`: Retrieves the term most similar to the given term, based on the specified embedding model. Similarly to the previous endpoint, unsupported models yield status codes 400.
- `POST /embedding/pooled/`: Accepts a request with a JSON body containing an array of embedding vectors and a pooling strategy (either `mean` or `max`). The request is validated, and a pooled vector is returned based on the provided strategy.

3.1.3 Library

CM2ML is distributed as a set of modular Node.js libraries, making it easy to integrate into custom JavaScript or TypeScript applications. The library provides direct programmatic access to parser and encoder functions, including additional utility functions. All components of the framework are published as standalone npm packages and are provided under the `cm2ml` namespace[3]. Two additional bundles are provided for convenience:

- `@cm2ml/builtin`: Exports all built-in parser and encoder plugins, along with utility functions to compose available plugins. It also includes a pre-configured list of plugin combinations covering all supported parser–encoder pairs. This package is browser-compatible and does not include the CLI or REST adapters.
- `@cm2ml/cm2ml`: Includes all internal modules of the framework, along with the two pre-configured CLI and REST adapters and their executables.

Developers can import any parser or encoder and invoke it directly in code. Listing 3.3 shows a simple example of using the library to parse and encode an ArchiMate model. After installing the library, the CM2ML components can be simply imported (line 2). The model is first loaded from disk (line 5) and parsed into the IR (line 7). The resulting `GraphModel` is then passed to the term frequency encoder, which computes the encoding with TF-IDF weighting enabled (line 9).

```
1 import fs from 'fs'
2 import { ArchimateParser, TermFrequencyEncoder } from "@cm2ml/builtin";
3
4 // read XML
5 const model = fs.readFileSync("example.archimate").toString();
6 // parse to GraphModel
7 const graphModel = ArchimateParser.invoke(model, { debug: true })
8 // encode model
9 const output = TermFrequencyEncoder.invoke(graphModel, { tfIdf: true })
```

Listing 3.3: Example CM2ML Library Usage

Advanced users of the library can also use lower-level utility packages, such as `@cm2ml/plugin` to define custom encoders or `@cm2ml/ir` to work directly with the IR.

[3] `https://www.npmjs.com/~cm2ml`

3.1.4 Visualization

CM2ML provides a browser-based interface to visualize the encoding output and parsed IR of a conceptual model. The visualization is useful to interactively explore parser behavior, observe effects of encoder parameters in real-time, and understand how model elements are transformed into ML-compatible formats. The visualizer operates on a single model at a time and does not support batch processing. However, its strength lies in the ability to inspect, debug, and understand individual encoding results and trace them back to their source elements.

Figure 3.2 shows the main view of the CM2ML visualizer. On the left, users can select a parser and configure its parameters. Below that, the raw input model (e.g., an ArchiMate XML file) is displayed, along with options to load a model from a local file or remote URL. The center panel visualizes the parsed IR as a graph and selecting a node highlights its details, including attributes and links to other elements, in the lower property panel.

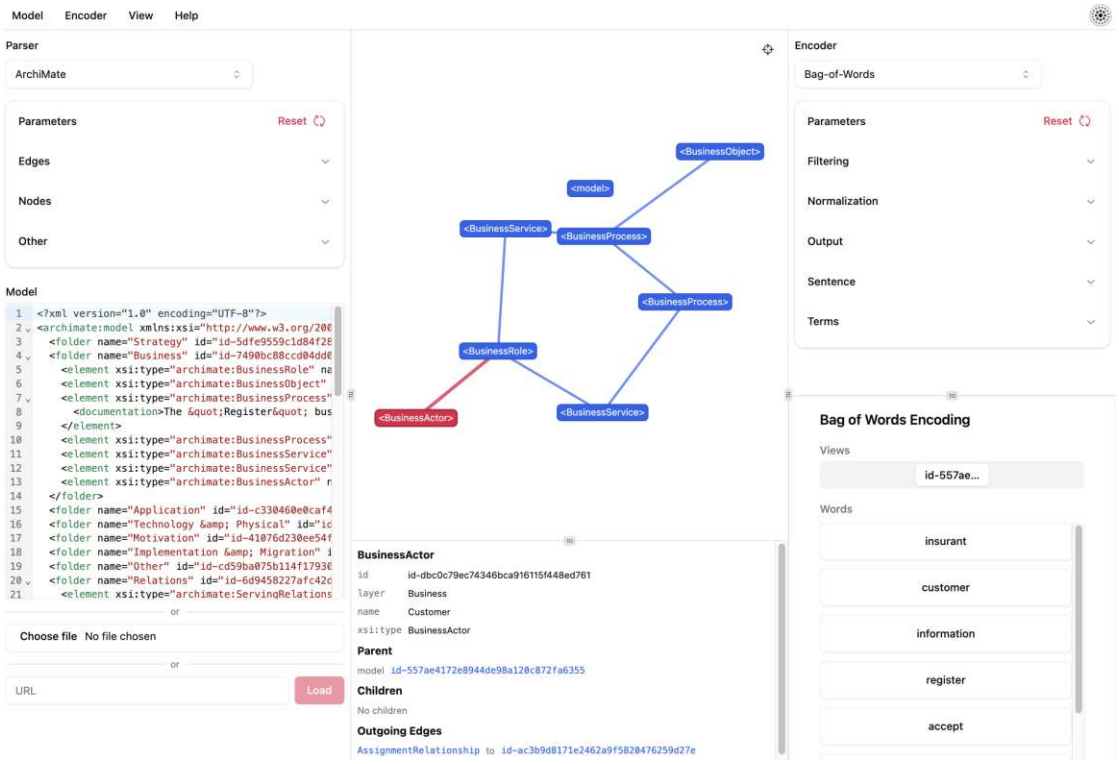


Figure 3.2: CM2ML Visualization Interface

On the right, users can choose an encoder and adjust its parameters. The encoding output is shown below in a panel that changes depending on the selected encoder (e.g., BoW list, term-document matrix, embedded vectors, etc.). This visualization allows users to inspect how specific parameters (e.g., term filters or normalization options) influence the resulting encoding in real-time. All parser and encoder settings can be reset

to defaults by using the corresponding reset buttons.

In addition to the graph-based view, the IR can also be rendered as a tree, which can be more helpful to inspect containment hierarchies and understand the overall model structure. Figure 3.3 shows the tree layout, where each child node represents a direct containment or relationship from its parent (i.e., the model).

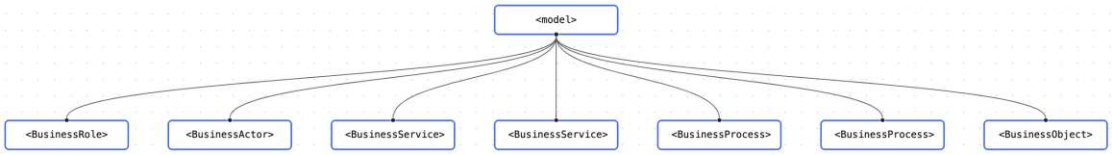


Figure 3.3: CM2ML Visualization: Tree View of the IR

3.2 Architecture

Figure 3.4 illustrates the overall architecture of the CM2ML framework. The architecture follows a modular, plugin-based design that separates concerns of parsing, encoding, and integration layers. At its core, CM2ML transforms conceptual models into encodings by parsing input data into an IR and applying configurable encoder plugins. Input and output interactions are handled by adapter components, which expose the framework through various interfaces such as the CLI, REST API, library, and visualization.

In the remainder of this section, we describe the key components (Section 3.2.1), the IR (Section 3.2.2), and the enabling technologies (Section 3.2.3) of the framework.

3.2.1 Components

The architecture is built around three central component types: parsers, encoders, and adapters. Each of these is implemented as a typed plugin that conforms to the interface provided by the `@cm2ml/plugin` package. A plugin in CM2ML is a reusable, typed execution unit with a defined input/output structure and parameter schema. This enables robust composition and integration within different execution environments.

Plugin composition is a central mechanism in CM2ML to allow flexible combination of functionality without duplicating logic. It allows two compatible plugins to be chained into a new plugin, where the output of the first becomes the input of the second. Parameter schemas are automatically merged during composition, and naming conflicts are resolved at runtime.

Parsers are responsible for transforming serialized conceptual models (e.g., string content of XML files) into the framework's internal graph-based IR. Each parser is specific to a modeling language (e.g., UML, ArchiMate, Ecore) and serves as the entry point for encoders. By composing a parser with an encoder, a complete pipeline is formed that supports the application of any available encoding to a given modeling language. This

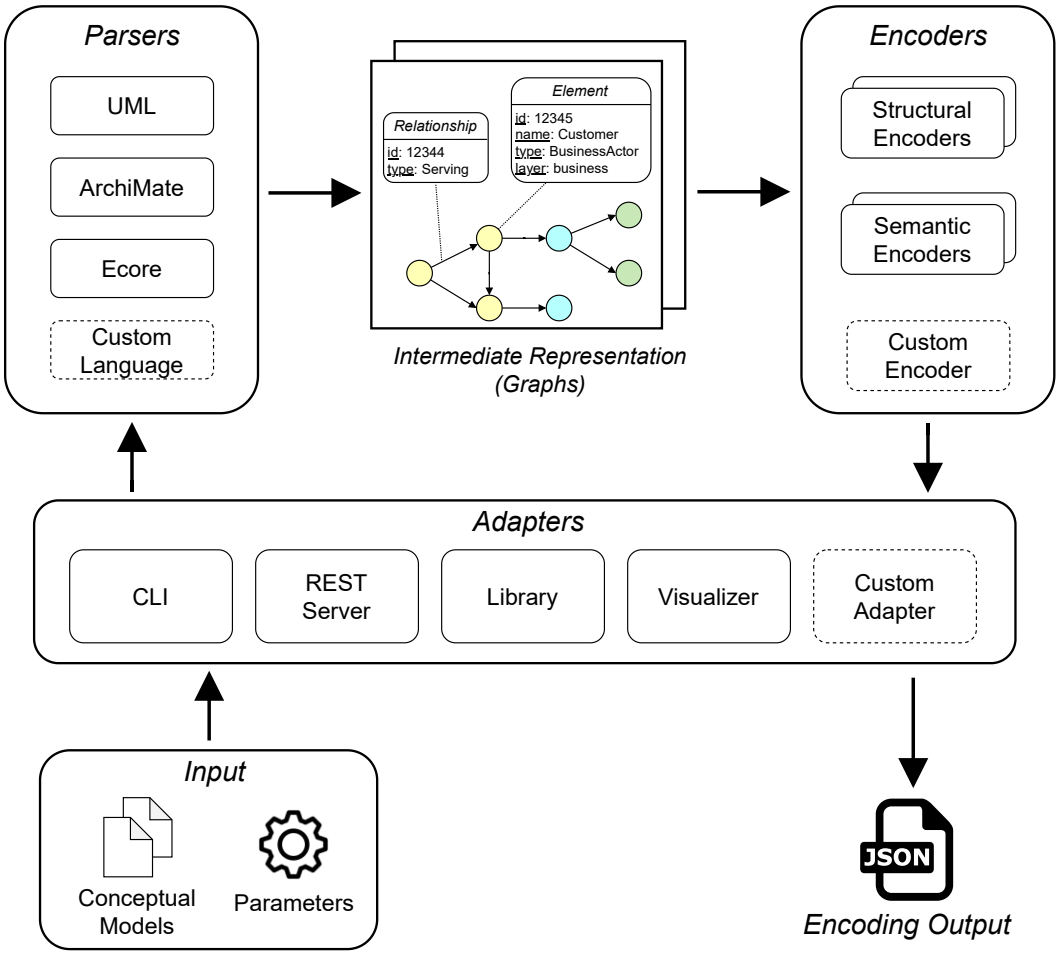


Figure 3.4: CM2ML Architecture

thesis contributes a parser for ArchiMate models, which is described in more detail in Section 3.3.

Encoders operate on the IR and transform it into structured JSON output. Structural and semantic encoders are implemented as plugins and support composition, customization, and batch execution. Batch support is provided by a utility wrapper that applies a given plugin to every model in an input collection.

Plugins expose typed configuration parameters that include metadata such as name, description, type, default value, and optional grouping. Supported parameter types include `boolean`, `number`, `string`, and `list<string>`. From these declarations, a validation schema is automatically derived and made available to adapters for validation and user interaction.

Adapters provide access to plugins in different runtime environments. The `@cm2ml/`

`plugin-adapter` package provides a generic interface for executing and serializing plugin outputs. Currently, the supported adapters include a CLI, a REST API server, a browser-based visualizer, and a programmatic library interface for Node.js applications. Custom adapters can also be defined. All adapters rely on the same plugin interface and produce encoding outputs in JSON format.

3.2.2 Intermediate Representation

A central design goal of the CM2ML framework is to decouple encoders from the specifics of modeling languages. This is achieved through a unified IR that provides a common abstraction layer for conceptual models. Encoders operate exclusively on this IR, making them agnostic to the input language and reusable across different model types. Thus, the IR acts as a bridge between language-specific parsers and encoder logic.

The IR is designed to be expressive enough to capture both structural and semantic information in diverse modeling languages, while remaining simple enough to be processed efficiently. While it is not possible to anticipate all future modeling scenarios, the IR aims to generalize over the most common constructs such as elements, relationships, attributes, and containment. To support extensibility, the IR allows parsers to embed language-specific metadata where needed.

Each IR instance includes a reference to a language-specific metamodel configuration. This configuration defines how key attributes, such as identifiers, types, and names, are accessed in models. It also specifies the list of valid element and relationship types, which is used for validation and type-sensitive encoding logic. Through this abstraction, encoders can interact with model elements generically, without hard-coding language-specific assumptions.

The main view of a model in the IR is a directed, attributed graph. Model elements are represented as nodes, and their relationships (e.g., associations or connectors) as edges. Both nodes and edges support arbitrary attributes, each with a declared type (e.g., string, number, boolean, category). Each node and edge also has a tag, which is typically initialized based on the source model's serialization (e.g., the XML tag name) for traceability. Nodes are uniquely identified and indexed internally to support efficient lookup. Edges are directed and reference their source and target nodes, with bidirectional references maintained for fast traversal.

In addition to the graph view, the IR exposes a tree structure over the nodes to represent containment hierarchies. Each model is required to have a single root node, and any model without a natural root must define a virtual one. Containment is explicitly tracked through parent and child references between nodes, and is automatically maintained when adding or removing nodes.

3.2.3 Technologies

CM2ML is implemented in the TypeScript programming language [Mic], a statically typed superset of JavaScript. TypeScript offers several advantages, including improved maintainability, early error detection, and stronger guarantees of correctness through static type checking. These benefits are relevant in complex software systems and have been shown to reduce code smells and improve long-term code quality [BM22]. Compatibility with Node.js [Foub] (or Bun [Ove] as an alternative runtime) enables the framework to be used in various application scenarios, and since TypeScript transpiles to JavaScript, the framework can run in both server-side and browser environments.

The project is organized as a monorepo [JJK+18] and managed by Turborepo [Ver], a high-performance build system optimized for JavaScript and TypeScript codebases. Turborepo simplifies the configuration and orchestration of development workflows such as building, linting, and testing. Vite [Inc] and Vitest [AFc] are used for front-end tooling and unit testing, while Zod [McD] is used to define and validate schemas, particularly for plugin parameters.

Model files are typically provided in XML format and parsing of these files is handled by the lightweight and fast htmlparser2 library [Bö]. The CLI is implemented using the CAC framework [EGO] and the REST API is built with Fastify [Foua], a web server optimized for speed and low overhead.

The browser-based visualizer is implemented using React [MP] and supported by the `shadcn/ui` component library [sha] for modern and accessible User Interface (UI) elements. It is deployed as a Progressive Web Application (PWA) [FKNW22], with all data processing and rendering handled client-side. No backend connectivity is required, and all application state is managed in-browser using the zustand state management library [Kat] and persisted via the `localStorage` API[4]. Graph-based visualizations of the IR are rendered using the vis-network library [vis] and for tree-based visualizations, React Flow [wG] is used.

3.3 ArchiMate Parser

The ArchiMate parser extends CM2ML with support for processing ArchiMate models and transforming them into the framework's IR. It is implemented as a plugin and through composition it can be combined with all available semantic and structural encoders.

This thesis implements a parser for ArchiMate models, supporting two XML formats: *(i)* Archi Tool Storage format (`*.archimate`), which is used as the main storage format in the Archi[5] tool, and *(ii)* The Open Group ArchiMate Model Exchange File Format (`*.xml`), which is a standard file format for the exchange of ArchiMate models between different tools.

[4] `https://developer.mozilla.org/docs/Web/API/Window/localStorage` (Accessed: 29.05.2025)

[5] `https://www.archimatetool.com/` (Accessed: 01.06.2025)

3.3.1 Implementation

The overall pipeline of the ArchiMate parser is defined through plugin composition, which combines a generic XML parser with a metamodel-specific refinement component. This modular design separates language-independent parsing logic from language-specific transformations. The processing steps of the ArchiMate parser are described below.

1) XML Parsing: The generic XML parser is a reusable plugin that converts raw XML input into an initial instance of the framework's IR. It is modeling language-agnostic and performs basic parsing tasks such as:

- For each XML element, an IR node is created with its tag name.
- XML attributes are extracted and converted into typed IR attributes (e.g., string, number, category)
- Parent-child relationships are established between elements to form a tree structure.
- XML nodes containing text are ignored by default, but a configurable handler allows parser authors to process these as needed. In the case of ArchiMate, this is only relevant for models that contain text within `purpose`, `documentation`, and `name` nodes.

The output of this stage is an unrefined IR object that reflects the document structure but lacks language-specific semantics such as relationships, types, or metamodel constraints. No edges are created in the initial IR and it consists only of nodes and their hierarchical structure.

2) Node Filtering & Cleanup: Irrelevant XML elements such as style (containing font and color information), layout bounds (i.e., x-and y-coordinates in the diagram), or other unused tags (e.g., `property` or `profile`) are removed. Diagram views may optionally be preserved or discarded based on the configuration.

3) Format-Specifc Preprocessing: The parser supports two ArchiMate XML serialization formats, each requiring specific preprocessing. A format-detection step dispatches the model to the appropriate restructuring function. For models in format *(i)*, child nodes stored in nested `<folder>` elements are lifted to the root. Text content from `<documentation>` or `<purpose>` children is added to attributes of their parent element. For models in format *(ii)*, ID attributes are renamed (`identifier` to `id`), container elements (e.g., `<elements>`, `<relationships>`) are flattened, and relationship tags are renamed to match a uniform format.

4) Type Normalization: To ensure compatibility with the current ArchiMate specification, the parser replaces deprecated type names with their updated equivalents. For example, `UsedByRelationship` is renamed to `ServingRelationship` or types with the outdated prefix `Infrastructure` are replaced with the prefix `Technology`. In addition, British is normalized to American spelling (e.g., `RealisationRelationship` to `RealizationRelationship`). The replacements are applied directly to the `xsi:type` attribute of relevant elements.

5) Refinement via Handlers: The refinement phase converts the initial IR tree into a semantically enriched model based on the ArchiMate metamodel. The main task of the refiner is to hierarchically process the IR and to create relevant edges between nodes. Each node is matched to a handler in a registry based on its `xsi:type` or tag. If no handler is found and strict mode is enabled, an error is thrown, otherwise, the node is discarded. Element handlers inject default attributes, infer derived properties (e.g., layer), and ensure metamodel compliance. Relationship handlers convert nodes into edges between source and target elements. Optionally, relationships can be represented as nodes, with explicit source and target edges connecting them to their endpoints. Diagram handlers optionally connect elements to their respective views.

6) Pruning and Validation: A final pruning step removes unwanted nodes and edges according to user-specified whitelists or blacklists. These filters support inclusion or exclusion based on ArchiMate type, allowing fine-grained control over model content. After pruning, the model is validated against basic constraints, depending on configuration.

Finally, the refined model is returned as a complete IR instance, which is ready for further processing by the encoders.

3.3.2 Parameters

The parameters of the ArchiMate parser are listed in Table 3.1.

Parameter	Description	Type	Default
`debug`	If enabled, log debugging information.	`boolean`	`false`
`strict`	If enabled, perform validations and do not accept invalid models.	`boolean`	`false`
`relsAsNodes`	Treat relationships as nodes.	`boolean`	`false`
`viewsAsNodes`	Include views and link their respective elements.	`boolean`	`false`
`nodeWhitelist`	Whitelist of ArchiMate element types to include in the model. Root nodes will never be removed. Ignored if empty.	`list<string>`	`[]`
`nodeBlacklist`	Blacklist of ArchiMate element types to exclude from the model. Root nodes will never be removed.	`list<string>`	`[]`
`edgeWhitelist`	Whitelist of edge types to include in the model. Ignored if empty.	`list<string>`	`[]`
`edgeBlacklist`	Blacklist of edge types to exclude in the model.	`list<string>`	`[]`

Table 3.1: ArchiMate Parser Parameters

CHAPTER 4

Encoders

This chapter presents the semantic encoding methods implemented in CM2ML to transform conceptual models into ML-compatible representations. The encoders extract, quantify, and structure semantic information from conceptual model artifacts in different ways, which can be controlled through parameters. Four different semantic encoders are implemented:

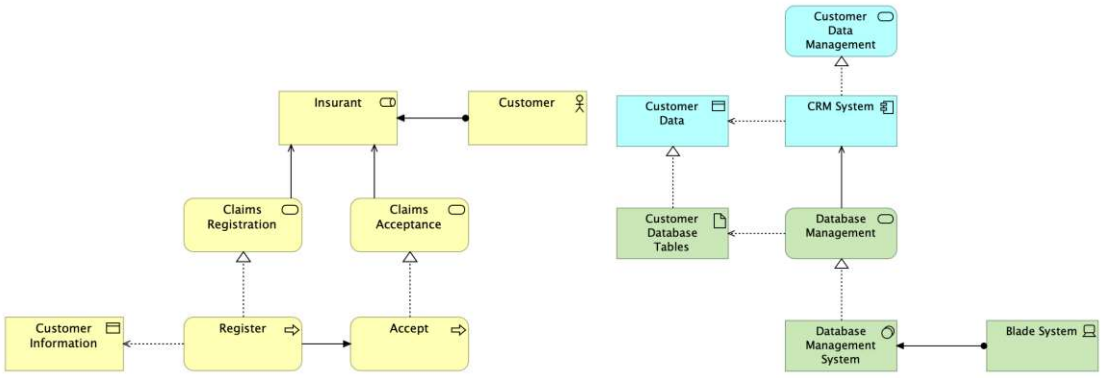
- The **BoW Encoder** (see Section 4.2) extracts raw terms from conceptual models based on element names, types, or other attributes. The output is a list of unweighted terms that might require additional preprocessing to be used in ML scenarios.
- The **TF Encoder** (see Section 4.3) extends the BoW encoder by quantifying term occurrences. It optionally computes TF-IDF weights and can produce more advanced representations such as bi-grams or n-grams that partially include structural information.
- The **Embeddings Encoder** (see Section 4.4) maps extracted terms to pre-trained word embeddings. This encoder supports several general and domain-specific embedding models and provides fallback strategies to deal with Out-Of-Vocabulary (OOV) terms.
- The **Triples Encoder** (see Section 4.5) extracts all element–relationship–element triples from models. These triples may optionally include element type information and vector representations, such as embeddings for names or one-hot vectors for types.

A concept shared by all encoders, except the triples encoder, is the use of a **Term Extractor** that serves as a common preprocessing layer. This layer extracts textual

information from parsed models and applies optional normalization techniques such as tokenization, stemming, or lowercasing. The term extractor is implemented as a reusable plugin and is configurable through a set of parameters that define which elements to include or filter out, how to normalize them, and how to format the output. These parameters are reused across the encoders. The first section of this chapter (Section 4.1) describes the term extractor in more detail.

In the remaining sections, the realized semantic encoders are described in more detail. For each encoder, the following aspects are described: *(i)* the processing steps of the transformation, *(ii)* a sample JSON output created via the CLI, *(iii)* an overview of configuration options, focusing on encoder-specific parameters (i.e., without the shared term extractor parameters), and *(iv)* a visualization example to demonstrate how encoding results can be explored interactively in the browser.

To illustrate the encodings, two small example ArchiMate models are used throughout this chapter, as depicted in Figure 4.1. To demonstrate the visualization examples, the model in Figure 4.1a is used. Both models are simplified extracts taken from the ArchiSurance case study [JBQ12] and are designed to contain a representative subset of elements, relationships, and naming conventions, as present in real-life models.



(a) Example Model A (b) Example Model B

Figure 4.1: ArchiMate example models used throughout this chapter

The final section of this chapter (Section 4.6) provides a comparison of the implemented semantic encoders and addresses the RQs of this thesis.

4.1 Common Term Extraction

The term extractor serves as a central component in the encoding process and is reused in several encoders. Its purpose is to extract, filter, normalize, and structure textual information from conceptual models to prepare it for further transformation. It is implemented as a reusable plugin in CM2ML and is configured through its own set

of parameters, which are shared among encoders, in addition to their encoder-specific configurations.

Figure 4.2 illustrates the role of the term extractor within the overall encoding pipeline. Initially, raw conceptual models are parsed into an IR that captures their structure as directed graphs. The term extractor operates directly on this graph-based representation to selectively extract lexical information and produce a list of terms. Its behavior can be customized to determine what information to include (e.g., names, types, or other attributes), how to filter it (e.g., based on term length or stop word lists), how to normalize it (e.g., tokenization, lowercasing, or stemming), and how to format the output (e.g., including node identifiers for traceability). The extracted and processed terms are provided in a standardized data structure that forms the basis for individual encoding steps. The BoW, TF, and embeddings encoders all internally invoke the term extractor to obtain terms as input and then apply their encoder-specific transformations (e.g., frequency counting in the TF encoder or mapping terms to embedding vectors in the embeddings encoder) to produce the final encoding output.

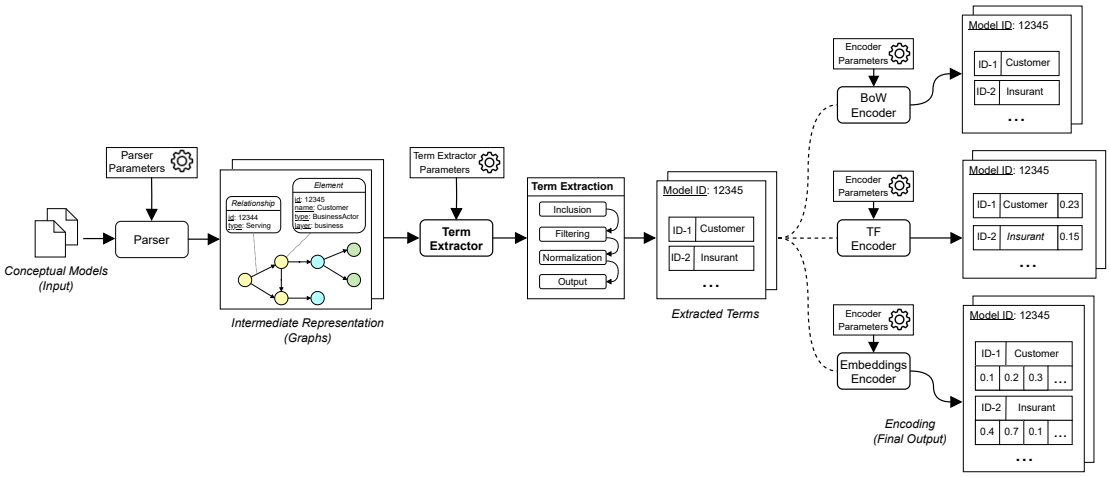


Figure 4.2: Term Extraction Process

The remainder of this section describes the term extractor component in more detail. First, the internal data structures for representing extracted terms are shown (Section 4.1.1). Then we describe the individual processing steps (Section 4.1.2), and finally, we describe the configurable parameters along with examples to illustrate their effect (Section 4.1.3).

4.1.1 Data Structures

To ensure a consistent and extensible representation of extracted lexical content, the term extractor and related encoders rely on a set of interfaces. These interfaces define a uniform format for representing extracted terms and their transformations, allowing reuse across different encoder components. Figure 4.3 shows the main interfaces involved.

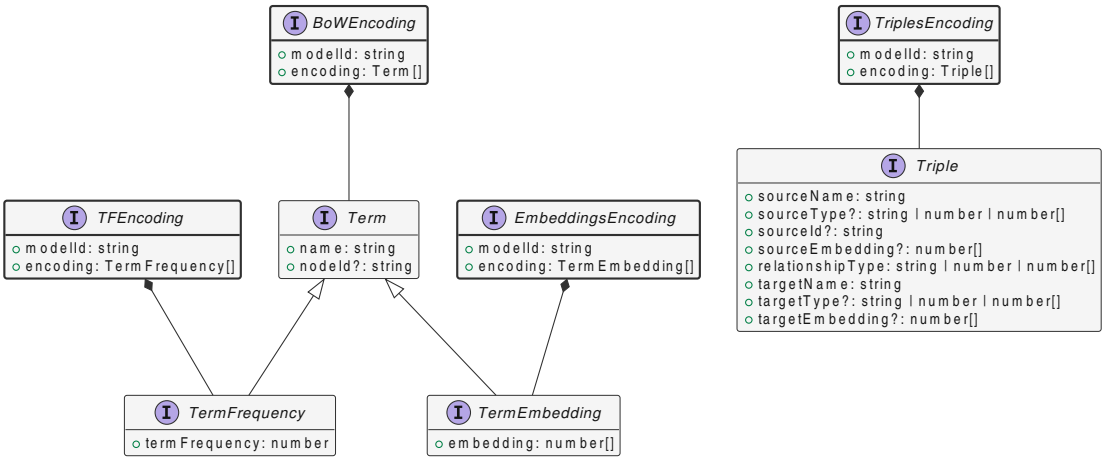


Figure 4.3: Data Structures to represent Terms and Encodings

The central data type used across encoders is the `Term` interface, which contains two fields: `name`, representing the extracted string (i.e., a token), and an optional `nodeId`, which preserves traceability by linking the term to its originating model element. This interface forms the basis for all term-based encodings (highlighted in bold in the figure), namely:

- `BoWEncoding`: A simple wrapper that associates a `modelId` with a list of extracted `Term` objects. No transformations are applied beyond the term extraction phase. This interface is used in the BoW encoder (see Section 4.2).
- `TFEncoding`: Represents a list of `TermFrequency` entries, each containing the term string and its corresponding frequency (which can additionally be weighted and normalized, e.g., through TF-IDF). This interface is used in the TF encoder (see Section 4.3).
- `EmbeddingsEncoding`: Extends the basic `Term` interface by adding an embedding vector to each term, represented by the `TermEmbedding` interface. Embeddings are fixed-length numeric arrays derived from pre-trained language models. This interface is used in the embeddings encoder (see Section 4.4).

In contrast to these term-based encodings, the `TriplesEncoding` interface is specific to the triples encoder (see Section 4.5) and is used to represent structural element-relationship-element triples. Each `Triple` object includes source and target names, a relationship type, and optionally, source/target types and embeddings. Types may be represented as strings, indices (`number`), or one-hot vectors (`number[]`), depending on the encoding configuration.

4.1.2 Processing Steps

The term extractor operates through four sequential processing steps, reflected in the four parameter groups (see Section 4.1.3), to prepare textual information for the encoders: inclusion, filtering, normalization, and output.

Inclusion: Determines which textual information from the IR of models is selected for extraction, including model element names, types, and additional user-specified attributes.

Filtering: Removes terms that are unlikely to contribute meaningfully to subsequent encoding tasks. The term extractor performs two main filtering operations: term length filtering and stop word removal. Terms shorter or longer than user-defined thresholds are discarded, and by default common English stop words (e.g., *a*, *an*, *the*, *are*, etc.) are excluded, as these terms typically carry minimal semantic content. Additional stop words can be specified, e.g., to include domain-specific words.

Normalization: Normalization harmonizes the extracted terms to improve consistency in subsequent processing steps. Three basic normalization steps are supported: tokenization, lowercasing, and stemming. Tokenization splits text into separate terms using predefined delimiters such as spaces, hyphens, and underscores. Lowercasing ensures uniformity by simply converting terms to lowercase. Stemming reduces variations of words to a common root form using the Porter-Stemmer algorithm [Por80]. Future enhancements could include more advanced NLP techniques, such as context-sensitive tokenization or lemmatization.

Output: The final processing step generates structured output, optionally including traceability via node identifiers. Traceability is relevant, for example, for visualization, to link encoded representations directly back to the original model elements. The current output format is kept minimal to reduce the overall data being processed and to provide sufficient flexibility for subsequent encoder processing steps.

4.1.3 Parameters

The behavior of the term extractor is controlled by a set of parameters that control which textual elements are included, how they are preprocessed, and how the output is structured. These parameters are shared between all encoders that internally invoke the term extractor (i.e., BoW, TF, and embeddings encoder) to ensure consistent term extraction. Table 4.1 summarizes all available parameters, grouped by functional category. Their effects are described in more detail below.

Inclusion: The inclusion parameters control which textual information is extracted from the models. When `includeNames` is enabled, the names of model elements (e.g., *Customer Database*) are extracted as terms. If `includeTypes` is enabled, the type of each element (e.g., *ApplicationComponent*) is also extracted and added to the set of terms. Additionally, `includedAttributes` allows specifying any custom element attributes to be treated as terms (e.g., *id* or *layer* attributes in ArchiMate models).

Parameter	Description	Type	Default
Inclusion			
`includeNames`	Extract element names as terms	`boolean`	`true`
`includeTypes`	Extract element types as terms	`boolean`	`false`
`includedAttributes`	Attributes to include as terms	`list<string>`	`[]`
Filtering			
`minTermLength`	Minimum length of terms	`number`	`1`
`maxTermLength`	Maximum length of terms	`number`	`100`
`stopWords`	Stop words to exclude from terms	`list<string>`	`DEFAULT`†
Normalization			
`tokenize`	Split and clean terms into tokens	`boolean`	`false`
`termDelimiters`	Delimiters used for splitting tokens	`list<string>`	`[ , -, _]`
`lowercase`	Convert terms to lowercase	`boolean`	`true`
`stem`	Apply stemming to terms	`boolean`	`false`
Output			
`includeNodeIds`	Include node IDs in output	`boolean`	`true`
`separateViews`	Treat each view separately	`boolean`	`false`

† `DEFAULT` includes common English stop words such as *the*, *an*, *of*, etc.

Table 4.1: Term Extractor Parameters

Filtering: Filtering parameters are used to exclude irrelevant or undesirable terms. `minTermLength` and `maxTermLength` define the allowed term length range to possibly remove overly short terms (e.g., single characters) or unusually long ones. The `stopWords` parameter specifies a list of words that should be ignored during extraction. If not explicitly provided, a default list of common English stop words (e.g., *the*, *and*, *of*, etc.) is used to eliminate high-frequency but low-information terms.

Normalization: Normalization parameters control how the extracted strings are processed before being returned as terms. When `tokenize` is enabled, names are split into smaller tokens based on defined `termDelimiters` (e.g., whitespace, hyphens, underscores), allowing multi-word names to be represented as multiple terms. `lowercase` ensures that all tokens are transformed to lowercase, avoiding mismatches caused by inconsistent casing. If `stem` is enabled, the tokens are reduced to their root forms (e.g., *processing* becomes *process*), helping to generalize the inflections of words.

Output: The output parameters influence how the extracted terms are structured in the

final output. If `includeNodeIds` is enabled, each term is linked to the corresponding node identifier in the model by adding an additional `sourceId` attribute to each term in the output. For ArchiMate models, setting `separateViews` to true causes the extraction process to treat each view within a model separately, producing distinct term lists per view instead of aggregating all terms at the model level.

4.2 Bag of Words Encoder

The BoW encoder transforms conceptual models into a list of textual terms that represent the semantic content of the model. Unlike more advanced encoding techniques, the BoW encoder does not apply any weighting or additional transformation to the terms. It simply extracts and outputs them, optionally removing duplicates or formatting them as sentence-like strings. This encoder is useful for simple ML applications such as model classification or clustering based on textual features, or it can be used with additional preprocessing in more advanced ML scenarios.

Technically, the BoW encoder acts as a thin wrapper around the term extractor plugin (see Section 4.1). The term extractor is invoked internally, providing reusable logic for term inclusion, filtering, normalization, and output formatting. The underlying parameters of the term extractor are reused, and additional encoder-specific parameters are added. This includes duplicate removal, where terms can be deduplicated within a model, and sentence encoding, where instead of emitting isolated terms, the encoder can represent elements and relationships as full sentences, similar to the work in [GG21].

4.2.1 Processing Steps

The BoW encoder transforms conceptual models into lists of extracted terms or sentences. The following steps summarize the internal processing logic:

Input Handling & Term Extraction: The encoder receives a batch of models, each already parsed into the graph-based IR with invalid inputs filtered out. If sentence encoding is disabled, the encoder delegates term extraction to the term extractor plugin, passing along all relevant parameters (e.g., which terms to include, how to normalize, etc.).

Duplicate Removal (Optional): If enabled, the encoder removes duplicate terms to ensure that each term appears only once per model. This can be useful for purely symbolic or presence-based representations.

Sentence Encoding (Optional): If enabled, the encoder bypasses the term extractor and instead constructs a list of sentences. Depending on the provided parameters, the following types of sentences are constructed:

- **Entity-based Sentences**: For each model element, a sentence is generated from its name, and optionally its type is prepended (e.g., *Application Component: CRM System.*).

- **Relationship-based Sentences**: If `encodeRels` is also enabled, the encoder traverses the model's relationships and maps them to predefined verb phrases (e.g., *Serving* relationship becomes *serves*). Sentences are then constructed by combining the connected source and target elements of the relationship with the verb phrase (e.g., *Customer Portal serves Order Processing.*). This parameter is currently only supported for ArchiMate relationship types via a hard-coded verb mapping, but can be easily extended to accommodate additional modeling languages.

Output Formatting: The final output consists of a list of entries, each containing a `modelId` and a list of associated `Term` objects. Each term includes a `name` (i.e., word or sentence) and, optionally, a `nodeId` for traceability. All extracted terms are stored as unordered lists, without frequency information or vector representations. This design makes it a lightweight and interpretable encoding method that can serve as a foundation for further transformation.

4.2.2 Example Output

Listing 4.1 shows a sample encoding result for the model from Figure 4.1b, using default parameters. For readability, the output and node identifiers have been simplified. The output consists of a JSON object for each processed model, which contains a model identifier (`modelId`) and a list of `terms`. Each term includes a `name` (the extracted word or token) and an optional `nodeId` referencing the originating element in the model.

```
{
    "modelId": "model-a",
    "terms": [
        { "nodeId": "a-1", "name": "example" },
        { "nodeId": "a-2", "name": "customer" },
        { "nodeId": "a-2", "name": "data" },
        { "nodeId": "a-3", "name": "customer" },
        { "nodeId": "a-3", "name": "data" },
        { "nodeId": "a-3", "name": "management" },
        { "nodeId": "a-4", "name": "crm" },
        { "nodeId": "a-4", "name": "system" },
        { "nodeId": "a-5", "name": "customer" },
        { "nodeId": "a-5", "name": "database" },
        { "nodeId": "a-5", "name": "tables" },
        { "nodeId": "a-6", "name": "database" },
        { "nodeId": "a-6", "name": "management" },
        { "nodeId": "a-7", "name": "blade" },
        { "nodeId": "a-7", "name": "system" },
        { "nodeId": "a-8", "name": "database" },
        { "nodeId": "a-8", "name": "management" },
        { "nodeId": "a-8", "name": "system" }
    ]
}
```

Listing 4.1: BoW Encoder Example Output

As shown, multiple terms can originate from the same node (e.g., *customer*, *data*, and *management* from the node with ID *a-3*), since tokenization is enabled by default.

In addition, identical terms may appear multiple times across different nodes, since deduplication is disabled by default.

4.2.3 Parameters

The BoW encoder builds on the parameters of the term extractor and introduces additional encoder-specific parameters to control deduplication and sentence-based output formatting. Table 4.2 lists the BoW-specific configuration options. The parameters of the term extractor are inherited and can also be configured (see Section 4.1.3). In the following, we describe each parameter, specific to the BoW encoder, in more detail.

Parameter	Description	Type	Default
`removeDuplicates`	Remove duplicate terms so that each term appears only once per model	`boolean`	`false`
`encodeAsSentence`	Encode entities as sentences	`boolean`	`false`
`includeRels`	Include relationships in sentences	`boolean`	`false`

Table 4.2: BoW Encoder Parameters

`removeDuplicates`: If enabled, ensures that each unique term appears only once per model. This is useful for producing a strict set-based representation rather than a multiset. For example, if a term like *database* occurs multiple times within a model, only one instance is retained.

`encodeAsSentence`: Switches the output format from individual terms to full sentences. Instead of raw terms (e.g., *[customer, data]*), the encoder emits sentences consisting of element types and names (e.g., *Business Object: Customer Data.*). This option is particularly useful when using the output as training input for language models (e.g., Universal Sentence Encoder [CYK+18]).

`encodeRels`: Used in combination with `encodeAsSentence`, this parameter enables the inclusion of inter-element relationships in the generated sentences. Relationships are mapped to predefined verbs (e.g., *serves*, *triggers*), creating sentences like *Application Service: Billing Service serves Business Process: Payment Processing.*. This parameter is currently only supported for ArchiMate relationship types.

4.2.4 Visualization

Figure 4.4 shows the visualization of the BoW encoder, using default parameters. On the left, the model is rendered as a graph (or tree) using the internal IR. On the right, the encoder panel displays the selected encoder, along with its parameters, grouped into categories. Below this, the visualization of the encoding output for the selected model or view (depending on the `separateViews` parameter) is shown. Each term appears as a single word, reflecting the default configuration in which only names are included, and

basic normalization (i.e., lowercasing and tokenization) is applied. Each item in the list supports traceability, that is, clicking on a term highlights the corresponding element in the model graph, as shown in the figure for the term *insurant*.

Figure 4.5 shows the output when the `encodeAsSentence` parameter is enabled together with `encodeRels`. Instead of isolated words, the encoder now produces full sentences derived from model elements and relationships, allowing structural connections in the model to be represented as natural language expressions. The generated sentences combine element types and names with a verb phrase using fixed templates (e.g., *BusinessService: Claims Registrations serves BusinessRole: Insurant.*).

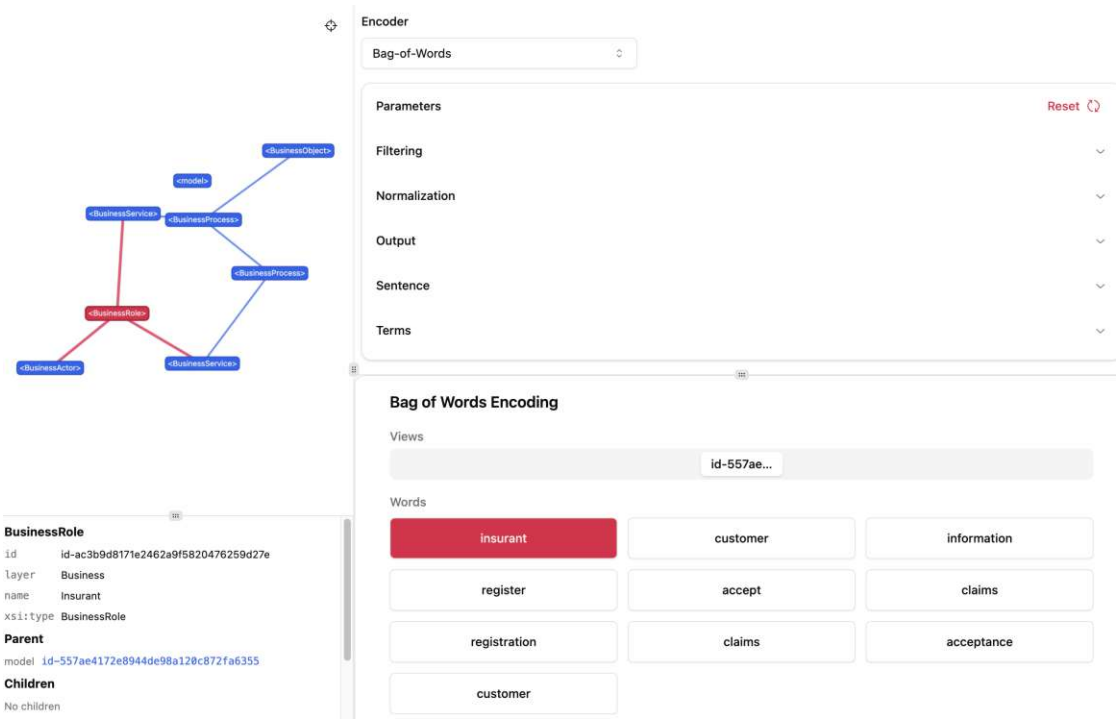


Figure 4.4: BoW Encoder Visualization (Default Parameters)

4.3 Term Frequency Encoder

The TF encoder transforms conceptual models into a representation that captures the frequency of extracted terms within models. More specifically, it produces a term-document matrix, where rows correspond to model identifiers, and columns to unique terms derived from the input corpus. The value at each position reflects the frequency of the corresponding term in the given model. Depending on the configuration, this frequency can be raw, normalized, or weighted using the TF-IDF scheme. This encoding format makes the TF encoder useful for ML scenarios that benefit from feature-based vector representations, such as classification, clustering, or similarity-based analysis.

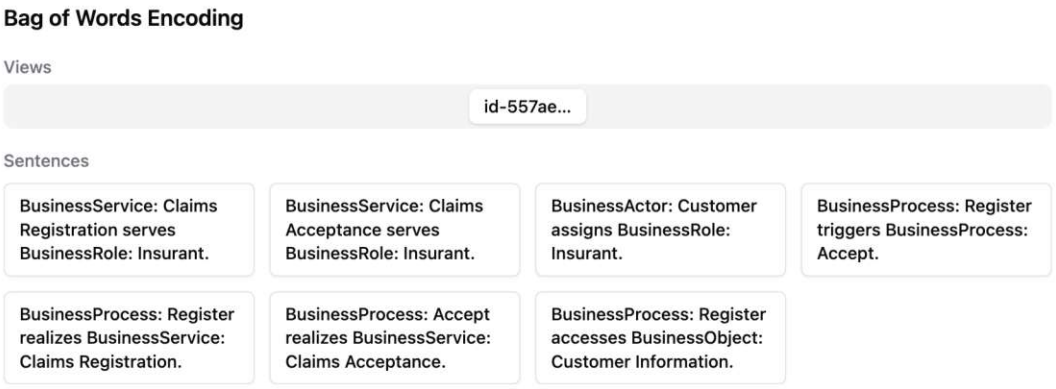


Figure 4.5: BoW Encoder Visualization with Sentence Encoding

The TF encoder builds on the common term extractor (see Section 4.1), reusing its capabilities and parameters for term extraction and preprocessing. Beyond standard term extraction, the encoder also supports bi-gram and n-gram representations to capture richer contextual semantics. Bi-grams are constructed by combining two attributes of model elements (e.g., name and type), joined with a configurable separator. N-grams, on the other hand, are derived from paths of a specified length within the model graph, optionally treating edges as undirected or allowing shorter paths to also be included in the output.

4.3.1 Processing Steps

The TF encoder receives a batch of conceptual models in the graph-based IR, with invalid models filtered out, as input. Depending on the configuration, one of three term extraction strategies is applied. If bi-gram encoding is enabled, the encoder generates composite terms by combining two element-level attributes (e.g., type and name) into a single string. If n-gram encoding is enabled, it constructs structural paths through the model graph (i.e., sequences of connected nodes of configurable length), which are then represented as terms. If neither strategy is selected, the encoder falls back to standard single-term extraction using the term extractor plugin and its configured parameters.

After term extraction, the encoder aggregates terms per model and computes their frequencies. These are stored as lists of term-frequency pairs, optionally linked to their source elements via node identifiers. A frequency cut-off can be applied at this stage to exclude infrequent terms from further processing. Once all term frequencies are collected, a global vocabulary is constructed from the union of terms across all models. This vocabulary defines the columns of the resulting term-document matrix, while models or views constitute the rows. The values in the matrix are determined by the configuration parameters `normalizeTf` and `tfIdf`, which control whether the frequencies are scaled or reweighted.

By default, if neither normalization nor TF-IDF is enabled, the matrix contains raw term counts (e.g., a value of *2* if a term occurs twice in a model). If normalization is enabled, these counts are scaled by the total number of terms in the model. The normalized term frequency $TF(t, d)$ for a term t in model d is computed as:

$$TF(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{4.1}$$

where $f_{t,d}$ is the number of occurrences of term t in model d, and the denominator sums over all term occurrences in that model. If normalization is disabled, the encoder uses the raw count $f_{t,d}$ directly.

If TF-IDF weighting is enabled, each normalized term frequency is further scaled by the IDF of the term. The IDF score downweights terms that are common across many models and highlights those that are locally frequent but globally rare. The IDF for a term t is calculated as:

$$IDF(t) = \log\left(\frac{N + 1}{DF(t) + 1}\right) + 1 \tag{4.2}$$

where N is the total number of models and $DF(t)$ is the number of models containing term t. The smoothing constants $(+1)$ are used to avoid division by zero and ensure numerical stability. The final TF-IDF weight is then given by:

$$TF - IDF(t, d) = TF(t, d) \cdot IDF(t) \tag{4.3}$$

This weighting scheme emphasizes the most informative terms, i.e., those that occur frequently within a model but not across all models. The final output includes the term-document matrix (mapping model IDs to their corresponding term vectors), the ordered list of terms (defining the matrix columns), and the list of processed model identifiers (corresponding to the matrix rows).

4.3.2 Example Output

Listing 4.2 shows an example output of the TF encoder, using the two models from Figure 4.1 as input. For readability, the output and node identifiers have been simplified.

The output of the TF encoder consists of three components: the list of model identifiers from the models that have been processed, the global vocabulary of extracted terms, and the corresponding term-document matrix. Each model is represented as a row vector aligned with the shared vocabulary, and each entry denotes the frequency of the corresponding term within the model. Depending on the parameters, the cell values can alternatively be normalized frequencies or computed TF-IDF scores. For a single model, the output includes a list of extracted `TermFrequency` objects (see Section 4.1.1) to

```
{
    "modelIds": ["model-a", "model-b"],
    "termDocumentMatrix": {
      "model-a": [2, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 1, 1],
      "model-b": [3, 2, 3, 1, 3, 3, 1, 1, 0, 0, 0, 0, 0, 0, 0]
    },
    "termList": [
      "customer", "data", "management", "crm", "system", "database",
      "tables", "blade", "insurant", "information", "register",
      "accept", "claims", "registration", "acceptance"
    ]
}
```

Listing 4.2: Encoder Example Output

provide traceability, e.g., for the visualization. Note that traceability is partially lost for terms that occur more than two times.

Table 4.3 displays the same term-document matrix in tabular format for better readability. Each cell indicates how often a term appears in the respective model. For instance, the term *customer* appears twice in `model-a` and three times in `model-b`. Conversely, *register*, *accept*, and *claims* are specific to `model-a`, while *crm* and *blade* occur only in `model-b`. This representation can be used for various vector-based ML techniques, such as classification or clustering.

Model	customer	data	management	crm	system	database	tables	blade	insurant	information	register	accept	claims	registration	acceptance
model-a	2	0	0	0	0	0	0	0	1	1	1	1	2	1	1
model-b	3	2	3	1	3	3	1	1	0	0	0	0	0	0	0

Table 4.3: Term-Document Matrix for the output in Listing 4.2

4.3.3 Parameters

Table 4.4 lists the available parameters of the TF encoder. The parameters listed here control the encoding behavior related to term frequency computation and term composition strategies, such as bi-grams and n-grams. The parameters of the term extractor are inherited and can also be configured (see Section 4.1.3). In the following, we describe each parameter, specific to the TF encoder, in more detail.

TF Parameters: The `normalizeTf` parameter controls whether raw term counts are normalized by the total number of terms in each model (see Equation (4.1)). When enabled, term frequencies are converted to relative values, reducing the impact of model size. If `tfIdf` is set to `true`, the encoder computes TF-IDF weights instead of raw or

Parameter	Description	Type	Default
TF Parameters			
`normalizeTf`	Normalize term frequency by total number of terms	`boolean`	`false`
`tfIdf`	Compute TF-IDF scores for terms	`boolean`	`false`
`frequencyCutoff`	Minimum frequency for a term to be included	`number`	`1`
Bi-gram Parameters			
`bigramEnabled`	Enable Bi-gram extraction	`boolean`	`false`
`bigramSeparator`	Separator for Bi-gram terms	`string`	`.`
`bigramFirstTerm`	First term of the Bi-gram	`string`	`name`
`bigramFirstTermAttr`	Attribute name for the first term if type is attribute	`string`	
`bigramSecondTerm`	Second term of the Bi-gram	`string`	`type`
`bigramSecondTermAttr`	Attribute name for the second term if type is attribute	`string`	
N-gram Parameters			
`nGramEnabled`	Enable N-gram extraction	`boolean`	`false`
`nGramLength`	Length of N-gram paths	`number`	`2`
`keepLowerPaths`	Keep lower length paths in N-gram extraction	`boolean`	`false`
`undirected`	Consider the graph as undirected when extracting paths	`boolean`	`false`

Table 4.4: TF Encoder Parameters

normalized frequencies (see Equation (4.3)). This emphasizes terms that are common within a model but rare across the entire batch. The parameter `frequencyCutoff` sets a threshold to exclude globally rare terms. Terms with total frequency across all models below this threshold are omitted from the vocabulary and corresponding term-document matrix.

Bi-gram Parameters: The TF encoder supports bi-gram construction, where two element-level attributes are combined into a single composite term. Enabling the parameter `bigramEnabled` activates this mode. The parameters `bigramFirstTerm` and `bigramSecondTerm` define which attributes to use as term components. Supported values are `name`, `type`, or `attribute`. If the value is `attribute`, the corresponding attribute name must be specified via `bigramFirstTermAttr` or `bigramSecondTermAttr`. The `bigramSeparator` determines how the two components are joined (e.g., a period or dash).

N-gram Parameters: Alternatively, the encoder can extract structural n-grams from the model graph. This is enabled via the `nGramEnabled` parameter. The `nGramLength` parameter specifies the length of the paths to extract, where each path corresponds to a sequence of connected nodes. If `keepLowerPaths` is enabled, the encoder also retains shorter paths encountered during traversal. The parameter `undirected` determines whether the graph is traversed in a bidirectional manner, leading to more paths, or follows the directed edges.

4.3.4 Visualization

Figure 4.6 shows the visualization of the TF encoder, using its default parameters. On the left, the conceptual model is displayed as a graph using the IR. On the right, the encoder interface displays the selected encoder, its configurable parameters, and the resulting output. The encoding result is visualized as a term-document matrix, containing the frequency of each extracted term within the current model.

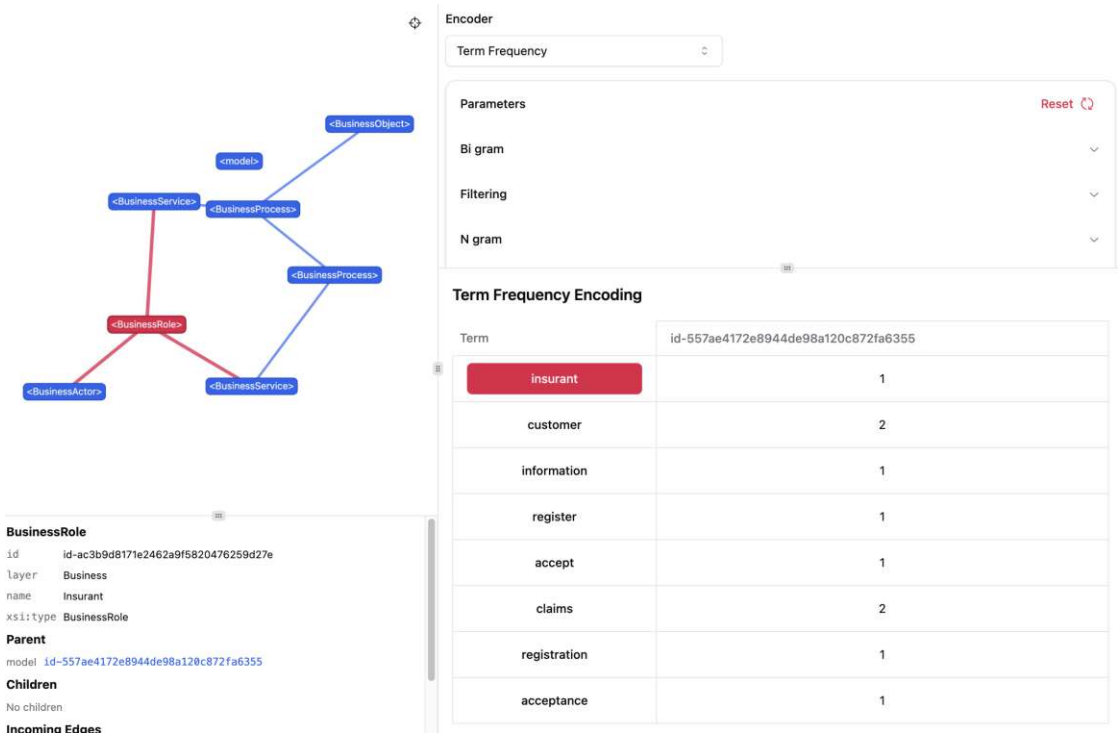


Figure 4.6: TF Encoder Visualization (Default Parameters)

In this matrix, rows correspond to terms and columns to documents (in this case, a single model). The cell values indicate the raw term frequencies. For example, the term *customer* appears with a frequency of 2, since it occurs twice in the model. Since no additional encoding is enabled, terms are extracted using the standard term extractor as uni-grams, i.e., plain lexical tokens without contextual composition.

Figure 4.7a illustrates the visualization with bi-gram encoding enabled. Here, each term is a composition of two element-level attributes, in this case, `name` and `type`, joined using the default separator (e.g., *Customer.BusinessActor*). This representation allows for a more fine-grained differentiation between elements with similar names but different types, or vice versa.

Figure 4.7b shows the visualization under n-gram encoding with a path length of two, together with `keepLowerPaths` and `undirected` disabled. In this mode, the encoder traverses the model graph and extracts linear paths of connected nodes, treating each path as a compound term. For example, the pair *(Register, Accept)* represents a sequence of two connected elements. This approach encodes not only lexical content but also structural information. If the path length were increased to three, only three terms would be included: *(Register, Accept, Claims Acceptance)*, *(Register, Claims Registration, Insurant)*, and *(Accept, Claims Acceptance, Insurant)*, as these represent the only directed paths of length three in the model graph.

Term Frequency Encoding

Term	id-557ae4172e8944de98a120c872fa6355
Insurant.BusinessRole	1
Customer Information.BusinessObject	1
Register.BusinessProcess	1
Accept.BusinessProcess	1
Claims Registration.BusinessService	1
Claims Acceptance.BusinessService	1
Customer.BusinessActor	1

(a) TF Encoder Bi-gram Visualization

Term Frequency Encoding

Term	id-557ae4172e8944de98a120c872fa6355
(Register, Accept)	1
(Register, Claims Registration)	1
(Register, Customer Information)	1
(Accept, Claims Acceptance)	1
(Claims Registration, Insurant)	1
(Claims Acceptance, Insurant)	1
(Customer, Insurant)	1

(b) TF Encoder N-gram Visualization

Figure 4.7: TF Encoder Bi-gram & N-gram Visualization

4.4 Embeddings Encoder

The embeddings encoder maps extracted model terms to vector representations using pre-trained word embeddings. This transformation enables conceptual models to be processed using vector-based ML approaches that rely on semantic similarity between terms. Each extracted term is extended with an embedding vector, resulting in a dense encoding format that captures latent semantics derived from large external corpora [AX19].

Unlike previous encoders, which rely entirely on internal computations, the embeddings encoder requires access to external resources, namely pre-trained word embeddings stored in large text files. These resources are not loaded in full during runtime, instead, a preprocessed index is created using an external Python script to allow efficient lookup of term vectors at runtime. While the batch transformation interface (e.g., CLI or REST

server) accesses these files via Node.js, the visualization interface in the browser operates differently. In this case, fallback vectors (e.g., zero vectors) are initially returned, and the actual embedding lookup is delegated to a REST backend.

CM2ML currently supports several pre-trained embedding models: general-purpose embeddings such as Word2Vec [MSC+13] and GloVe [PSM14], as well as domain-specific embeddings like WordE4MDE [LDC23], trained on modeling-related corpora. All standard term extraction parameters are reused, and embedding-specific configuration options allow users to select the model, set the embedding dimension, choose strategies for handling OOV terms, and optionally encode entire models as single vectors by pooling term embeddings.

4.4.1 Processing Steps

The embeddings encoder transforms conceptual models into dense vector representations using pre-trained word embeddings. Due to technical limitations related to file system access in the browser environment, the encoder distinguishes between two execution modes: **batch processing** and **single-model processing**. Figure 4.8 illustrates the workflow of both execution modes within the framework.

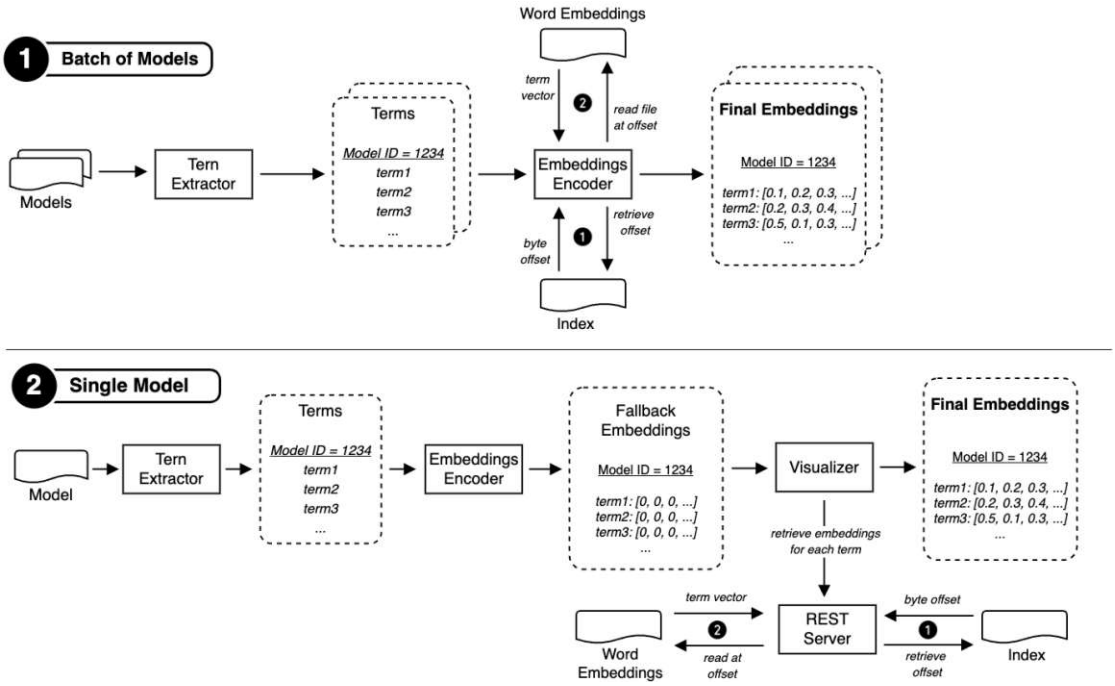


Figure 4.8: Embeddings Retrieval Process for a Batch of Models and a Single Model

Before using the encoder, the required embedding files must be downloaded and indexed. This can be achieved through an external Python script that leverages the `gensim` library [Řeh10] to download embeddings and generate an index file. The index maps

vocabulary terms to their byte offsets within the embedding files, allowing efficient retrieval during encoding.

In **batch mode** (top part of Figure 4.8), a set of models is first processed through the term extractor to obtain a list of relevant terms for each model. For each term, the encoder accesses the index to locate the corresponding byte offset with which the associated embedding vector can be retrieved directly from disk. These vectors are then attached to the terms, resulting in a list of term-embedding pairs per model as the final output.

In **single-model mode** (bottom part of Figure 4.8) direct file access via Node.js (e.g., `fs.readFileSync`) is not available due to browser environment restrictions. As a workaround, the encoder initially assigns fallback embeddings (e.g., zero vectors) to all terms. These placeholder vectors are passed to the visualization layer, which asynchronously retrieves the actual vectors by querying a REST server. The server uses the same index and offset-based retrieval as in batch mode to return the true embedding vectors to the browser for visualization.

The main operation of the encoder is the embedding lookup. If a term is present in the index, its corresponding vector is simply read and added to the output. Otherwise, if a term is OOV, the encoder applies an OOV strategy as configured. Supported strategies include substituting with a zero vector, generating a random vector in range $[-1, 1]$, or retrieving the vector of the most similar known word based on Levenshtein distance.

Additionally, the encoder optionally supports model-level vector pooling, whereby a single embedding is computed for an entire model by aggregating the individual term vectors. Supported pooling strategies include element-wise mean and maximum. The resulting pooled vector is included alongside the per-term embeddings in the output.

4.4.2 Example Output

Listing 4.3 shows an example output of the embeddings encoder applied to the two models shown in Figure 4.1. For readability, the output has been simplified: term vectors are truncated (actual dimensionality is 300), numerical values are rounded to four decimal places, and node identifiers of terms have been omitted.

Each model in the output is represented by its unique `modelId`. The corresponding object contains a list of `embeddings`, each associating a term extracted from the model with a pre-trained embedding vector. Furthermore, since the parameter `modelEncoding` is enabled in this example, a `pooledVector` is included for each model, representing the semantic content of the entire model. In this case, the pooled vector is the average of all individual term vectors, including duplicates and zero vectors if present, since the `poolingStrat` is set to `avg`.

The term *insurant* is not part of the embedding vocabulary and is therefore treated as OOV. Given that the `oovStrategy` is set to `zero`, its embedding is represented by a

```
{
  "modelIds": ["id-1", "id-2"],
  "modelEmbeddings": [
    {
      "modelId": "model-a",
      "pooledVector": [-0.1030, -0.0076, ...],
      "embeddings": [
        { "term": "insurant", "embedding": [0, 0, ...] },
        { "term": "customer", "embedding": [-0.1484, 0.1289, ...] },
        { "term": "information", "embedding": [-0.0718, -0.2656, ...] },
        { "term": "register", "embedding": [0.0040, -0.1934, ...] },
        { "term": "accept", "embedding": [-0.2314, -0.0928, ...] },
        { "term": "claims", "embedding": [-0.0918, 0.1690, ...] },
        { "term": "registration", "embedding": [0.0557, -0.0776, ...] },
        { "term": "claims", "embedding": [-0.0918, 0.1690, ...] },
        { "term": "acceptance", "embedding": [-0.3066, -0.0430, ...] },
        { "term": "customer", "embedding": [-0.1484, 0.1289, ...] }
      ]
    },
    {
      "modelId": "model-b",
      "pooledVector": [-0.0860, -0.0748, ...],
      "embeddings": [
        { "term": "customer", "embedding": [-0.1484, 0.1289, ...] },
        { "term": "data", "embedding": [-0.1729, -0.1426, ...] },
        /* Other terms with embeddings ... */
      ]
    }
  ]
}
```

Listing 4.3: Embeddings Encoder Example Output

zero vector. Also note that identical terms occurring more than once (e.g., *claims* or *customer*) are preserved in the output as repeated entries.

4.4.3 Parameters

Table 4.5 lists the encoder-specific parameters of the embeddings encoder. In addition to these, the shared parameters of the term extractor (see Section 4.1.3) can also be configured to influence which terms are extracted and how they are processed before retrieving their embeddings.

The `embeddingsModel` parameter specifies which pre-trained word embedding model to use for lookup. Supported values include `word2vec`, `glove`, as well as variants adapted to the modeling domain `word2vec-mde` and `glove-mde`. The files of these models must be downloaded in advance, including their index files, which enable efficient byte-level access to individual vectors.

The `dimension` parameter controls the dimensionality of the embedding vectors. If the selected embedding model provides vectors with higher dimensionality, they are simply truncated to the specified size. All currently supported models include vectors of 300 dimensions.

Parameter	Description	Type	Default
`embeddingsModel`	The pre-trained word embedding model to use (download of files required)	`string`	`word2vec`
`dimension`	The dimensionality of the embeddings	`number`	`300`
`oovStrategy`	The strategy to handle OOV terms	`string`	`zero`
`modelEncoding`	Encode the whole model as a single vector	`boolean`	`false`
`poolingStrat`	Strategy to pool the embeddings of the model terms if modelEncoding is enabled	`string`	`mean`

Table 4.5: Embeddings Encoder Parameters

OOV terms, i.e., words not found in the embedding index, are handled according to the `oovStrategy`. Available options include:

- `zero`: assign a vector of all zeros.
- `random`: generate a random vector in the range $[-1, 1]$.
- `discard`: skip the term entirely.
- `most-similar`: substitute the term with its nearest known word based on Levenshtein distance.

The `modelEncoding` flag enables generation of a single vector representation per model by aggregating all individual term embeddings. If activated, the `poolingStrat` parameter defines how this aggregation is performed. Supported pooling strategies include `mean` (element-wise average over all term vectors) and `max` (element-wise maximum value across all vectors).

4.4.4 Visualization

Figure 4.9 shows the visualization of the embeddings encoder in the browser, using default parameters. The left-hand side displays the input model (from Figure 4.1a) as a graph. On the right-hand side, the encoder parameters can be configured (not visible in the figure), and the output is presented as a matrix, where each row corresponds to a term and each column to a vector dimension.

The embedding values are displayed numerically, with color-coded background to reflect magnitude. Terms that are OOV and could not be mapped to any pre-trained embedding (e.g., *insurant*) are displayed with zero vectors and visually highlighted in red. Clicking on a term reveals its source element in the intermediate representation. When model-level encoding is enabled, a pooled vector is added above the term matrix.

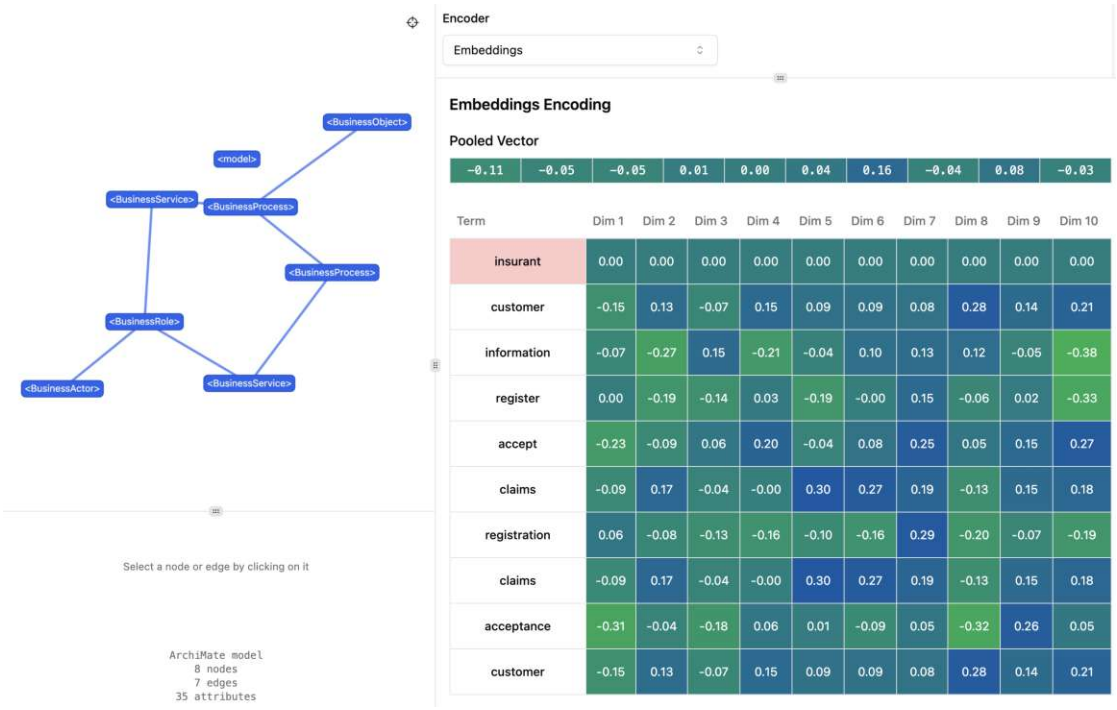


Figure 4.9: Embeddings Encoder Visualization

4.5 Triples Encoder

The triples encoder transforms conceptual models into sets of triples, each capturing a relationship between two model elements. Unlike the encoders described in previous sections, this encoder does not use the common term extractor (see Section 4.1) and instead directly encodes each edge in the model graphs as subject-relation-object triples. A triple consists of the source and target element names, the relationship type between these two elements, and optionally additional type encodings or embedding vectors.

The main motivation behind this encoder is to retain structural semantics that are lost in standard term-based encodings. By explicitly capturing relationship types and connected elements, the resulting representation is more suitable for ML tasks that require structural awareness and relational reasoning, such as link prediction, node classification, or other graph-based learning methods.

The triples encoder supports different encoding strategies for node and relationship types, including symbolic, numerical, and one-hot representations. Furthermore, if enabled, element names can be enriched with pre-trained word embeddings. In that case, multi-word names are handled using various strategies (e.g., averaging or concatenation of individual token embeddings), and OOV terms are handled via configurable fallback mechanisms.

4.5.1 Processing Steps

The triples encoder processes a batch of conceptual models by iterating over their relationships and converting them into subject-relation-object triples. It operates directly on the graph-based IR, bypassing the common term extractor used by other encoders, as it is designed to explicitly capture relationships between elements rather than individual lexical terms.

Each model is first filtered to ensure that it is valid. The encoder then iterates over the model's edges to extract the involved source and target nodes. If either node is missing or lacks a valid name attribute, the edge is skipped. For each valid edge, a triple is created consisting of the source and target element names and the type of the relationship connecting them. The relationship type is encoded either as a raw symbolic string, a numerical value, or a one-hot vector, depending on the configuration. Optionally, the encoder may include the source element's identifier for traceability, specified by the `includeSourceId` parameter. Furthermore, if `includeTypes` is enabled, the source and target types are also included in the triple. These types can be encoded again either as raw symbolic strings, numeric values, or one-hot vectors.

When `useEmbeddings` is enabled, triples are enriched with embedding vectors for source and target element names. Similarly to the embeddings encoder (see Section 4.4), this process begins by loading a pre-computed word embedding index file from disk. Each element name is tokenized and normalized into tokens, and for each token, the corresponding embedding vector is retrieved using byte offsets from the index.

The tokenizer applied before embedding lookup performs several normalization steps to ensure consistency with the token format of most embedding models: numeric characters are replaced with their textual representation (e.g., `123` becomes `one two three`), camelCase and acronyms are split into separate tokens, punctuation and delimiters are removed, and stop words and non-alphabetic tokens are filtered out.

If a word is not found in the index, an OOV strategy is applied, such as using a zero vector, sampling a random vector, or selecting the most similar word based on Levenshtein distance. For multi-word names, embeddings are combined using a configurable strategy (e.g., averaging, concatenation, or selecting the first token).

4.5.2 Example Output

An example output of the triples encoder is shown in Listing 4.5, encoding the two models shown in Figure 4.1. For readability, the list of triples is abbreviated, identifiers are simplified, and vector contents are truncated. Comments indicate the active positions in the one-hot vectors, e.g., `one-hot @ index 30` means that only the value at position 30 (starting at 0) is set to 1 and all other values are set to 0.

The output was created using the CLI command shown in Listing 4.4. The following parameters are applied: types are included and encoded as one-hot vectors, word embeddings are enabled using the GloVe model, multi-word names are combined by averaging

token embeddings, and the OOV strategy is set to `zero`, meaning unknown words are mapped to zero vectors.

```
cm2ml batch-archimate-triples ./models/testing/
    --include-types --types-as-one-hot
    --use-word-embeddings --embeddings-model glove
    --combine-words-strategy average --oov-strategy zero
```

Listing 4.4: CM2ML CLI command used to create the output in Listing 4.5

```
[
  {
    "modelId": "model-a",
    "triples": [
      {
        "sourceName": "Database Management",
        "relationshipType": [0,0,0,0,0,0,0,0,0,0,1],
        "targetName": "Customer Database Tables",
        "sourceEmbedding": [-0.493048, 0.231377, ... ],
        "targetEmbedding": [-0.617787, 0.331407, ...  ],
        "sourceType": [0, 0, ..., 1, 0, ...], // one-hot @ index 30
        "targetType": [0, 0, ..., 1, 0, ...], // one-hot @ index 32
        "sourceId": "a-1"
      },
      /* Other triples... */
    ]
  },
  {
    "modelId": "model-b",
    "triples": [
      {
        "sourceName": "Claims Registration",
        "relationshipType": [0,0,1,0,0,0,0,0,0,0,0],
        "targetName": "Insurant",
        "sourceEmbedding": [0.256355, 0.12046, ...],
        "targetEmbedding": [0, 0, ...],
        "sourceType": [0, 0, ..., 1, 0, ...], // one-hot @ index 7,
        "targetType": [0, 1, 0, ...], // one-hot @ index 1,
        "sourceId": "b-1"
      },
      /* Other triples... */
    ]
  }
]
```

Listing 4.5: Triples Encoder Example Output

In the output, each model contains a list of triple objects, consisting of various attributes. A minimal triple object includes the source and target names as strings together with the relationship type, in this case, encoded as a one-hot vector. Type inclusion is enabled, thus, the `sourceType` and `targetType` are also included and again, encoded as one-hot vectors. Since embeddings are enabled, the object also contains embedding vectors (300 dimensions) of the source and target names. For the name *Insurant*, no embeddings were found (i.e., OOV) and, since we use zero as an OOV strategy, the embedding vector is filled with 0s. Multi-word names (e.g., *Database Management*) are combined into single vectors by averaging the embedding vectors of individual tokens (e.g., *Database* and

Management). Finally, since the parameter `includeSourceId` is enabled by default, `sourceId` is also included, linking each triple to the corresponding source element node.

4.5.3 Parameters

The triples encoder defines its own set of parameters and does not reuse any term extraction parameters. All available parameters of the triples encoder are summarized in Table 4.6. In the remainder, we discuss each of the parameters and their effects.

Parameter	Description	Type	Default
`includeSourceId`	Include source IDs of elements in the encoding output for traceability	`boolean`	`true`
`includeTypes`	Include type encodings for source and target	`boolean`	`false`
`typesAsNumber`	Encode types as a numerical value	`boolean`	`false`
`typesAsOneHot`	Encode types in form of one-hot vectors	`boolean`	`false`
`useEmbeddings`	Use pre-trained word embeddings for element names	`boolean`	`false`
`embeddingsModel`	The pre-trained word embedding model to use (download of files required)	`string`	`word2vec`
`combineStrategy`	Strategy to combine embeddings for multi-word names	`string`	`avg`
`oovStrategy`	Strategy to handle OOV terms	`string`	`zero`

Table 4.6: Triples Encoder Parameters

Similarly to the other encoders, the `includeSourceId` parameter, enabled by default, controls whether the identifier of the source node is included in each triple for traceability.

The `includeTypes` parameter determines whether the element types of the source and target nodes are included in the triples. When enabled, these types are represented either as symbolic strings, numeric values, or one-hot encoded vectors. The encoding format is controlled by the parameters `typesAsNumber` and `typesAsOneHot`. If both are disabled, the type remains a plain string. If `typesAsNumber` is enabled, each type is mapped to a unique integer index. If `typesAsOneHot` is enabled, the type is represented as a one-hot vector whose length corresponds to the full type vocabulary. These options are mutually exclusive, and enabling both results in a one-hot encoding.

The `useEmbeddings` parameter enables embedding lookup for the source and target element names. If enabled, each name is tokenized and mapped to word vectors from a pre-

trained embedding model specified via the `embeddingsModel` parameter. Supported models include `glove`, `word2vec`, `glove-mde`, and `word2vec-mde`. All embeddings are stored locally and must be pre-downloaded.

Multi-word names are tokenized into words, and their embeddings are then combined using the strategy specified by the `combineWordsStrategy` parameter. Supported strategies include:

- `average`: Compute the mean of all token embeddings.
- `first`: Use only the first token's embedding.
- `concat`: Concatenate all token embeddings (note that this increases vector dimensionality).
- `skip`: Skip all multi-word terms and only embed single-word names.

If a token cannot be found in the embedding index, the encoder applies an OOV strategy, configured by the `oovStrategy` parameter with the following options:

- `zero`: Use a vector of zeros.
- `random`: Sample a random vector in the range $[-1, 1]$.
- `discard`: Skip the token entirely.
- `most-similar`: Replace the unknown word with the most similar one found in the vocabulary, using Levenshtein distance.

4.5.4 Visualization

Figure 4.10 shows the visualization of the triples encoder in the browser, using default parameters. The left-hand side displays the input model (from Figure 4.1a) as a graph. On the right-hand side, the encoder parameters can be configured, and the output is shown in a table with three columns: source name, relationship type, and target name.

As a more complex example, Figure 4.11 shows the result when type encodings and word embeddings are enabled. Here, the tabular view is extended with additional columns, including source and target types (numerically encoded), as well as source and target embedding vectors (truncated for readability).

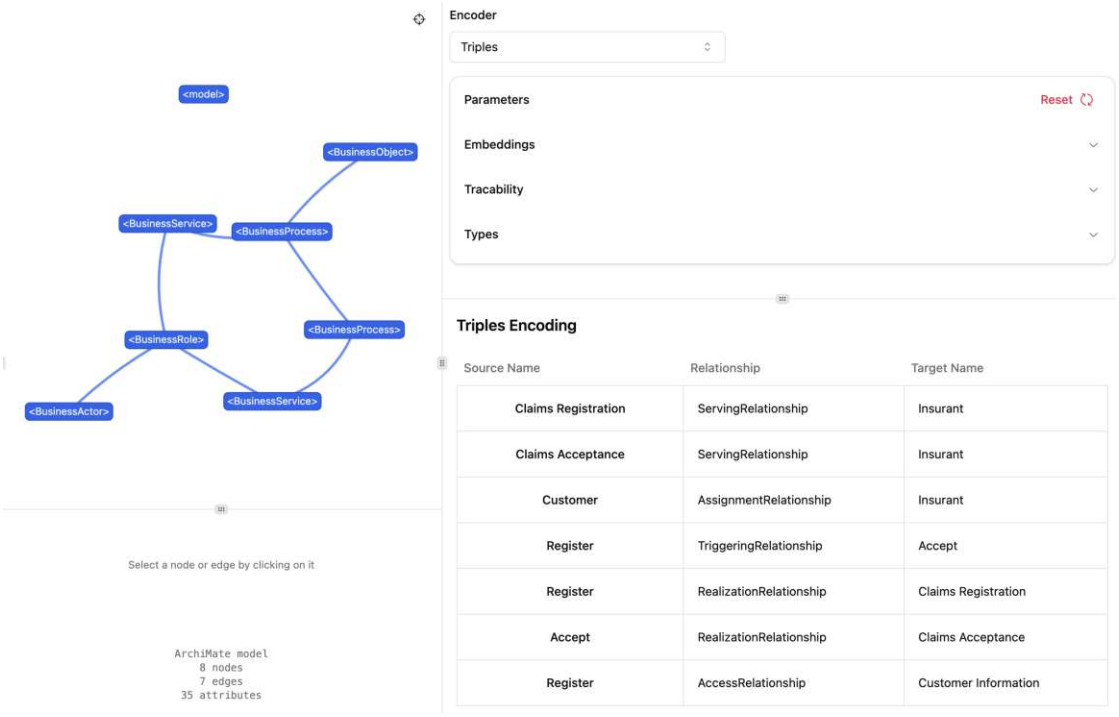


Figure 4.10: Triples Encoder Visualization

Triples Encoding

Source Name	Source Type	Relationship	Target Type	Target Name	Source Embedding	Target Embedding
Claims Registration	8	3	2	Insurant	−0.39, −0.17, −0.14, 0.07, 0.13, ...	−0.21, −0.52, −0.29, −0.00, 0.07, ...
Claims Acceptance	8	3	2	Insurant	−0.25, −0.29, −0.07, 0.22, −0.19, ...	−0.21, −0.52, −0.29, −0.00, 0.07, ...
Customer	1	9	2	Insurant	0.06, −0.27, −1.11, 0.69, −0.06, ...	−0.21, −0.52, −0.29, −0.00, 0.07, ...
Register	5	8	5	Accept	−0.63, 0.57, −0.47, 0.14, 0.16, ...	0.13, −0.19, 0.36, 0.62, 0.03, ...
Register	5	2	8	Claims Registration	−0.63, 0.57, −0.47, 0.14, 0.16, ...	−0.39, −0.17, −0.14, 0.07, 0.13, ...
Accept	5	2	8	Claims Acceptance	0.13, −0.19, 0.36, 0.62, 0.03, ...	−0.25, −0.29, −0.07, 0.22, −0.19, ...
Register	5	11	10	Customer Information	−0.63, 0.57, −0.47, 0.14, 0.16, ...	−0.25, −0.11, −0.72, 0.58, −0.49, ...

Figure 4.11: Triples Encoder Visualization including Types (Numeric Encoding) and Embeddings

4.6 Comparison of Encoders

This section compares the different encoder types introduced in the previous sections with the goal of evaluating their ability to extract and represent the semantic content of conceptual models in ML-compatible formats. The comparison addresses the two RQs of this thesis:

> **RQ1:** *To what extent can semantic information in ArchiMate models be extracted and encoded into suitable representations for machine learning applications?*
>
> **RQ2**: *How do semantic encoding strategies compare in terms of their ability to preserve information and support machine learning tasks?*

To answer these questions, we compare the following encoder variants:

- **BoW Encoder**: Extracts plain lexical terms or sentences without any weighting or numerical representation. Two variants are considered: *(i)* plain BoW using individual lexical tokens, and *(ii)* sentence-based encoding where sentences are generated from elements and relationships.
- **TF Encoder**: Produces frequency-based vectors for terms through term-document matrices. We distinguish between *(i)* standard uni-grams, *(ii)* bi-grams composed of multiple element attributes, and *(iii)* n-grams based on structural paths in the model graph.
- **Embeddings Encoder**: Maps terms to pre-trained word embeddings. We consider: *(i)* encoding each individual term with embedding vectors, and *(ii)* pooled representations that aggregate individual term vectors into a single model-level embedding.
- **Triples Encoder**: Captures structural relationships in the form of subject-relation-object triples. We consider the variant where triples are enriched with both, type encodings and word embeddings.

First, we conduct a comparative analysis of these encoders to characterize their strengths and weaknesses, and their suitability for ML tasks (Section 4.6.1). The results of the comparison serve as a basis for answering the RQs (Section 4.6.2).

4.6.1 Comparative Analysis

The purpose of this comparison is to characterize how different encoders represent the underlying model semantics and to assess their readiness and expressiveness for ML applications. For this, we define a set of comparative criteria and analyze each encoder across those dimensions.

Comparative Criteria

To compare the different encoder variants introduced in this chapter, we define a set of comparative criteria that capture both the semantic depth of the encodings and their properties relevant to ML applications. Table 4.7 lists the criteria, along with descriptions, used for the comparative analysis.

Criteria	Description
Granularity	The level at which information is encoded, e.g., individual terms, complete sentences, structural paths, or triples.
Lexical Coverage	Whether and to what extent the encoder extracts and uses lexical properties such as element names, types, and attributes.
Structure	The degree to which the encoder captures structural relationships (e.g., edges, paths, dependencies).
Context	Whether the encoder captures meaning from context, e.g., co-occurrence, term frequency, embedding similarity.
Dimensionality	The size and density of the output vectors (None / Low-dimensional sparse / High-dimensional dense).
Traceability	Can individual elements in the output be traced back to specific elements in the original model?
Interpretability	Can a human understand the meaning or origin of the encoded features?
ML Suitability	Is the output readily usable as input for standard ML methods (e.g., classification, clustering)?

Table 4.7: Criteria used for Encoder Comparison

Encoder Comparison

Table 4.8 provides a comparative overview of the encoders, evaluated against the defined criteria. Overall, the comparison reveals that no single encoder captures all semantic aspects. Instead, each encoder variant emphasizes different model semantics and has trade-offs in terms of dimensionality, interpretability, and ML suitability. This highlights the need for careful encoder selection based on task-specific requirements.

The **BoW encoders** operate at the surface level. The plain variant extracts individual terms from element names, types, or other attributes, without structural or contextual information. It excels in simplicity and interpretability, but lacks any notion of meaning or quantification beyond isolated words. The sentence-based variant adds limited structural and contextual cues by including relationships in the generated natural language sentences, however, only names and types are extracted. Both BoW variants produce purely symbolic

Encoding	Granularity	Lexical Coverage	Structure	Context	Dimensionality	Tracability	Interpretability	ML Suitability
BoW (Plain)	Term	✓	✗	✗	None	✓	✓	∼
BoW (Sentence)	Sentence	∼	∼	∼	None	✓	✓	∼
TF (Uni-gram)	Term	✓	✗	∼	Low	∼	✓	✓
TF (Bi-gram)	Element	✓	✗	✓	Low	∼	✓	✓
TF (N-gram)	Path	∼	✓	✓	Low	∼	✓	✓
Embeddings (Plain)	Term	✓	✗	✓	High	✓	∼	✓
Embeddings (Pooled)	Model	✓	✗	✓	High	✗	✗	✓
Triples	Triple	∼	✓	✓	High	✓	∼	✓

Table 4.8: Comparison of Encoders

encodings (no dimensionality) and require additional transformation (e.g., vectorization) for ML applications.

The **TF encoders** expand on BoW by incorporating frequency-based information and term co-occurrence patterns. The uni-gram variant captures term counts and thus supports basic contextual reasoning, though structural information remains absent. Bi-grams enrich the representation by combining element-level attributes (e.g., type and name), enabling finer-grained distinctions and shallow context modeling. N-grams go a step further by capturing structural paths of configurable length, embedding structural semantics into the representation, however, only element names are used. All TF variants produce sparse, interpretable vectors that are suitable for standard ML models. Traceability is partially lost due to aggregation.

The **embeddings encoder** transforms lexical terms into dense, fixed-size vectors based on pre-trained embeddings. The plain variant maintains term-level granularity and supports traceability but sacrifices interpretability due to large embedding dimensions. It is suitable for similarity-based ML tasks and benefits from distributional context captured by the underlying embedding model. If model-level pooling is enabled, the encoder outputs a single dense vector per model, which is efficient for classification or clustering tasks, but eliminates traceability and interpretability.

The **triples encoder** provides a structured representation by directly encoding relationships with their source and target elements as subject-predicate-object triples. It partially captures lexical information (element names and optionally types) and explicitly encodes the model structure. If embeddings are included, contextual information is also preserved. Triples strike a balance between structure and context but introduce complexity due

to their multi-field output format. The triples are useful, for example, for graph-based learning tasks or tasks requiring relational reasoning.

Suitability for ML Applications

The different encoder variants exhibit varying degrees of suitability for specific ML tasks, depending on the type of information they capture and the form of their output. This subsection outlines typical ML tasks and discusses which encoders are most appropriate for each.

For **classification tasks**, such as assigning categories to models, numerical vector representations are essential. The TF encoders (uni-gram, bi-gram, n-gram) and the embeddings encoder (with pooled embeddings) are best suited in this context. They provide compact, fixed-size vectors per model, which are compatible with traditional classifiers (e.g., LR, SVMs, or NNs). Pooled embeddings, in particular, offer a dense representation that is suitable for gradient-based optimization, though interpretability is limited.

For **similarity-based tasks**, such as model retrieval or clustering, the embeddings encoder is most appropriate. The use of pre-trained word embeddings enables semantic similarity to be measured in a continuous space, even for terms that differ lexically but are semantically related. Plain embeddings maintain term-level granularity, allowing fine-grained similarity comparisons, while pooled vectors support similarity measures at the model level.

Relational reasoning tasks (e.g., link prediction or node classification based on relations) benefit most from the triples encoder, which explicitly encodes relations between elements within the model.

4.6.2 Addressing the Research Questions

The comparative analysis in Section 4.6.1 shows that different types of semantic information present in conceptual models can indeed be captured and encoded into representations suitable for ML applications. The encoders developed and evaluated in this work are not limited to ArchiMate, but are designed to handle conceptual models in general, as long as they follow a graph-based structure with named elements and typed relationships. ArchiMate models constitute a subset of such conceptual models, and thus benefit directly from the same encoding mechanisms. The only encoder behavior specific to ArchiMate is the optional separation of views, where each view is treated as a distinct model instance during encoding.

Each encoder contributes to capturing a different aspect of model semantics, namely:

- **Lexical semantics**, such as names and types of model elements, are extracted by all encoders, with varying degrees of normalization and coverage (e.g., attribute inclusion).

- **Distributional semantics**, represent meaning through co-occurrence patterns and statistical frequency. This is captured by the TF and embeddings encoders.
- **Structural semantics**, including the relational topology of the model graph, are encoded by the triples encoder and, to a lesser extent, by the n-gram variant of the TF encoder or the sentence-based BoW encoder.
- **Contextual semantics**, such as the usage and ordering of terms, are incorporated through sentence encoding, path-based n-gram sequences, and embedding models trained on large textual corpora.

Although each encoder captures only a subset of the overall model semantics, the results show that various aspects can be systematically extracted and transformed into ML-compatible formats. However, certain limitations remain. For example, ArchiMate semantics, such as metamodel- or ontological semantics, are not yet fully captured by current encoders. Furthermore, trade-offs exist between interpretability, dimensionality, and representational richness. A subset of the encoders and parameters is quantitatively evaluated in Chapter 5.

In conclusion, semantic information from ArchiMate and related conceptual models can be effectively encoded for ML purposes, but the encoding strategy must be chosen with the desired type of semantic content and the target ML task in mind. Hybrid approaches that combine complementary encoders (including existing structural encoders) may offer a promising direction to capture richer representations.

CHAPTER 5

Evaluation

This chapter presents the experimental evaluation conducted to assess the effectiveness of selected semantic encoders in practical ML scenarios. The goal is to show how the encoders can be used for a specific task and how different parameter configurations can be systematically evaluated to assess their influence on ML performance.

Section 5.1 outlines the dataset and experimental setup used for the two ML tasks in this evaluation, including model selection, annotation process, evaluation metrics, and tooling. The two classification tasks are then introduced and described in detail. Section 5.2 focuses on the dummy classification task, where TF-based encodings are used to distinguish dummy from non-dummy views. Section 5.3 evaluates a node classification task, using the triples encoder to predict the type of an element given its partial local context. Both tasks evaluate encodings under varying parameter configurations and using different ML models.

5.1 Dataset & Experimental Setup

This section describes the dataset and experimental setup used for the two evaluation tasks. The aim is to ensure reproducibility and provide context for interpreting the results. Section 5.1.1 presents the dataset used in the experiments, including how models were selected and annotated to provide ground truth labels for the two classification tasks. Section 5.1.2 details the experimental setup of both tasks, including the evaluation metrics and the tools and hardware used during execution.

5.1.1 Dataset

Data-driven approaches in CM, particularly those involving ML, rely heavily on the availability of curated and labeled datasets. While general-purpose ML has benefited from large-scale labeled datasets across domains [GLX+23], conceptual modeling lacks

openly available domain-specific resources that support empirical evaluation and reproducibility [BCPP20]. Recent efforts in the modeling community have begun closing this gap by providing curated datasets of models. Examples include, MAR [LC20, LC22], a search engine for models in various languages (e.g., Ecore, BPMN, UML, or Petri nets), ModelSet [LCIC22, LIC22, LIC24], a labeled dataset of Ecore and UML models, or the Findable, Accessible, Interoperable, Reusable (FAIR) OntoUML/UFO Catalog [SBF+23].

Recently, for the enterprise modeling domain, the EA ModelSet was introduced as a curated dataset of ArchiMate models [GSB23, GSB25]. It aligns with FAIR principles [WDA+16] and was designed to support reproducible research, empirical and statistical analysis, and training of ML models in the EA domain. In this thesis, a filtered subset of the EA ModelSet is used to evaluate the semantic encoders and different parameter configurations on concrete ML tasks.

In the following, we describe the dataset used for both evaluation tasks in more detail, including how models were selected to obtain a representative subset and the annotation process to derive suitable labels for the two classification tasks.

Model Selection & Characteristics

The evaluation in this thesis is based on a curated subset of models extracted from the EA ModelSet dataset (version 0.0.3). The dataset in its original form contains 977 models of varying sizes and languages. To ensure compatibility with the embedding models used in the encoders, which are trained on English corpora, only models tagged as English were retained. This also aligns with the NLP components of the semantic encoders (e.g., tokenization), which are optimized for English. In addition, English-language models simplify the subsequent annotation process, as the involved annotators are fluent in this language. All selected models conform to the Archi XML serialization format and are valid input for the ArchiMate parser implemented as part of this thesis.

After filtering, 564 models remained, which form the basis for all evaluation experiments presented in this thesis. The selected models vary significantly in size, making them a representative sample for evaluating semantic encodings. The number of elements per model ranges from 10 to 4003, with an average of 102.13. The number of relationships spans from 0 to 5641, averaging 126.92 per model. The number of diagram views ranges from 1 to 328, with a mean of 6.97.

Annotation Process & Label Availability

Two types of classification tasks are addressed in this thesis, each requiring different forms of label acquisition. For the dummy view classification task (see Section 5.2), labels were created manually through an annotation process. In contrast, the node classification task (see Section 5.3) relies on automatically derived labels based on the model structure, without requiring human annotation.

The annotation process for dummy view classification involved six annotators with experience in CM. Each model view was annotated by two independent annotators to ensure label reliability. The annotation workload was split into primary and secondary assignments. The primary assignment evenly distributed the set of models among annotators, while the secondary assignment selected a subset from other annotators' primary sets to guarantee overlap and that each model is reviewed by more than one annotator.

Each diagram to be annotated corresponds to a view in a model and is uniquely identified by the filename pattern `<modelId>.png__-__<viewId>.png`. Up to five views per model were exported and included in the annotation set. Given that some models contain more than 100 views, this selection strategy prioritized coverage while ensuring feasibility. Views were exported as PNG images (using the Archi CLI), compressed using pngquant[1], and integrated into a custom LabelStudio[2] setup deployed via Docker. Each annotator worked on an isolated branch of the annotation project, with individual labeling environments containing the assigned diagrams.

The labeling interface supported three categories: *Yes* (dummy view), *No* (valid view), and *Pattern* (view containing generic patterns or templates). In total, 1450 views were annotated across 564 models. 1737 labels of type *No*, 706 of type *Yes*, and 216 of type *Pattern* were created. Of the annotated views, 1178 (81.2%) showed agreement between annotators, while 272 (18.8%) contained conflicting labels. Furthermore, 241 views were labeled by only one annotator.

To derive the final binary labels, all *Pattern* annotations were reclassified as *Yes*. Models with inconsistent annotations across views were manually reviewed. In cases of ambiguity or irreconcilable disagreement, the affected models were excluded from the dummy classification dataset. After this cleanup step, 573 views were labeled as *Yes* and 839 as *No*. This set of binary labels exhibits moderate class imbalance and forms the basis for the dummy classification task.

For the node classification task, labels were automatically derived from the encoded models. Specifically, each triple extracted from a model encodes a source node (with name and type), a relationship type, and a target node (with name and type). The type of the target node serves as the classification label, eliminating the need for manual annotation. The resulting corpus contains 69,890 triples, with 57,592 unique triples. The number of triples per model varies considerably, ranging from 7 to 5641, with a mean of 127.30 and a standard deviation of 321.98.

The class distribution of target types is shown in Figure 5.1 with classes on the y-axis (colored by ArchiMate layer) and the number of instances on the x-axis. The distribution is highly imbalanced, with a few dominant classes and several infrequent ones. For instance, *ApplicationComponent* and *BusinessProcess* account for nearly 8% each of the labeled instances, whereas some types occur fewer than 20 times (e.g.,

[1] `https://pngquant.org/` (Accessed: 04.06.2025)
[2] `https://labelstud.io/` (Accessed: 04.06.2025)

TechnologyInteraction, *AndJunction*, and *OrJunction*). This imbalance poses challenges for ML training, particularly for generalization and performance on underrepresented classes. Strategies to mitigate this are considered in the evaluation, such as sampling or class weighting.

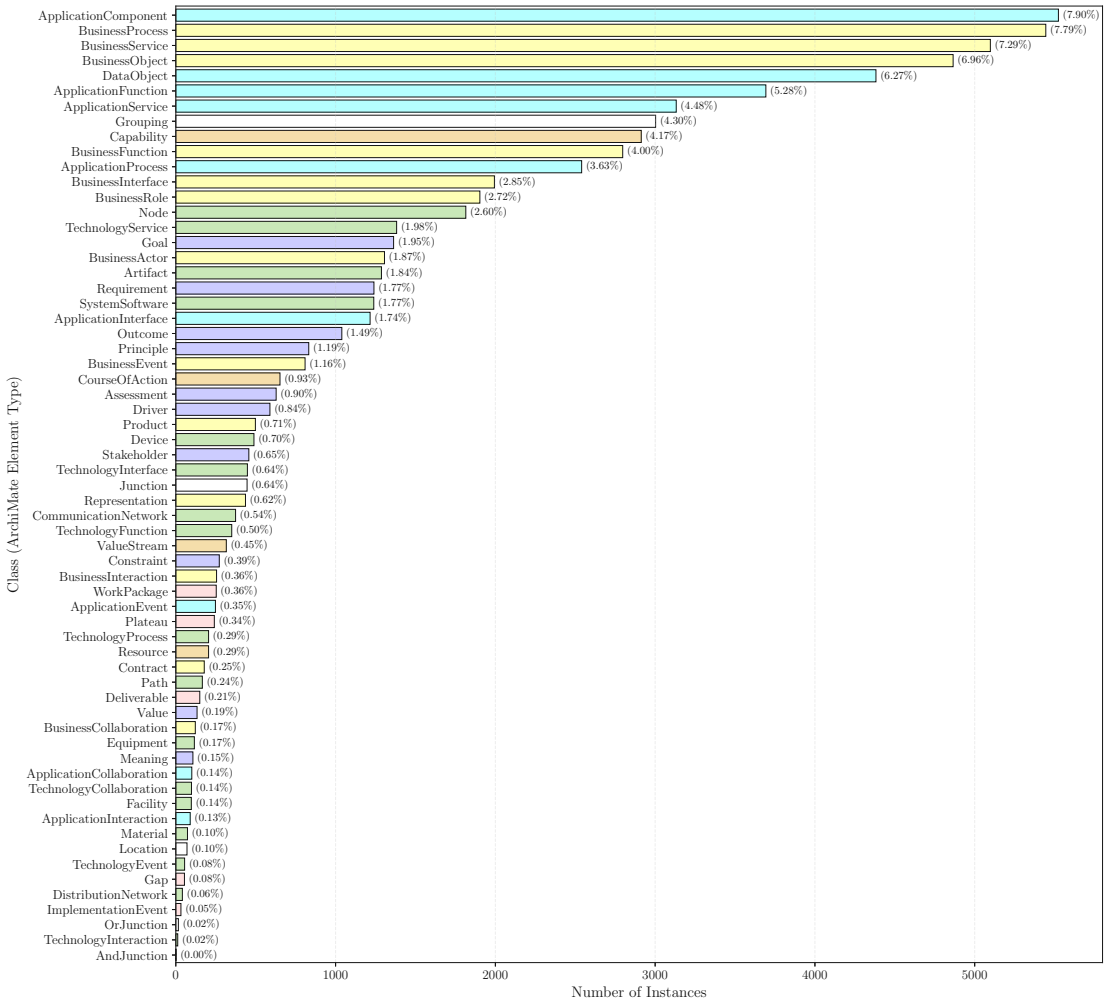


Figure 5.1: Class Distribution of Target Types (for Evaluation 2)

5.1.2 Experimental Setup

This subsection describes the experimental setup used for both evaluation tasks. First, we provide a high-level overview of the two evaluation scenarios with a diagram that illustrates the overall workflow. Then, we describe the evaluation metrics used to assess model performance, tailored to the binary and multi-class nature of the respective tasks. Finally, we detail the hardware environment and tools used to conduct the experiments for reproducibility.

Setup Overview

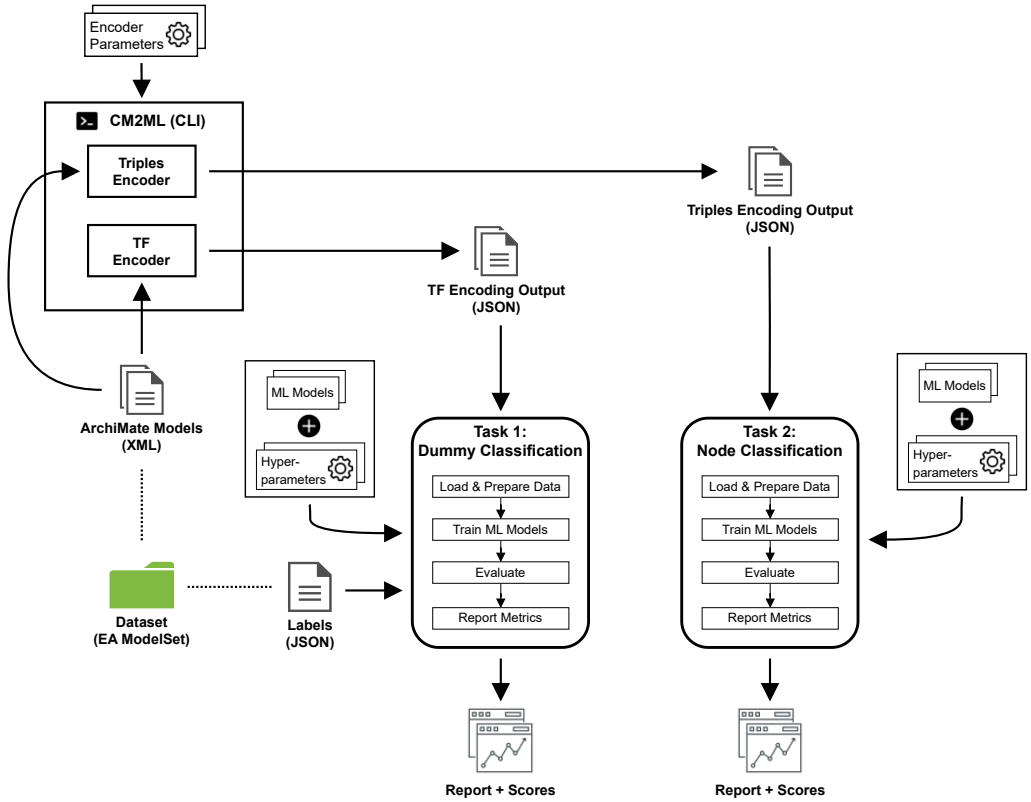


Figure 5.2: Experimental Setup Overview

Figure 5.2 illustrates the overall workflow used to evaluate the semantic encodings developed in this thesis. The process is structured around two ML tasks: dummy classification and node classification. Both serve to empirically assess the usefulness of the encodings under different parameter settings. The tasks are described in more detail in the corresponding subsections (see Section 5.2 and Section 5.3)

The workflow begins with a subset of ArchiMate models from the EA ModelSet (see Section 5.1.1). These models are encoded using the CM2ML CLI with different parameter configurations. For the dummy classification task, the TF encoder is used to generate a term-based numerical representation of each model view. For node classification, the triples encoder extracts element–relationship–element structures enriched with semantic features such as word embeddings and type encodings.

The encoded representations are stored as JSON files and serve as input to each task-specific evaluation pipeline. In both tasks, a set of ML models is trained using the encoded data, and grid search with cross validation is used to find the best hyperparameters for each ML model.

For dummy classification, manually annotated binary labels are used to train classifiers

that distinguish between dummy and non-dummy views. In the node classification task, the goal is to predict the element type of the target node in each triple, formulated as a multi-class classification problem. The target labels are directly derived from the model structure.

Each evaluation pipeline consists of four stages: loading and preprocessing the encoded data, training and tuning classifiers, evaluating predictions using standard metrics, and reporting results. The final report provides a comparison of ML models with the best hyperparameters and performance metrics, from which insights can be derived on how different encoder configurations affect predictive accuracy.

Evaluation Metrics

This section describes the set of metrics used to evaluate both ML tasks in this thesis. All metrics are standard in classification tasks and provide complementary perspectives on model performance. Their values vary in the interval $[0, 1]$, where higher values indicate better performance.

The first evaluation task, dummy classification (see Section 5.2), is a binary classification problem, where each model view is classified as either valid (negative class) or dummy (positive class). The following metrics are used:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.1}$$

Precision measures the proportion of predicted dummy views that are indeed dummy. TP (true positives) denotes the number of correctly predicted dummy views, and FP (false positives) denotes the number of valid views incorrectly predicted as dummy. High precision indicates that valid views are rarely misclassified as dummy.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.2}$$

Recall measures the proportion of actual dummy views that are correctly identified. FN (false negatives) refers to dummy views that were incorrectly predicted as valid. High recall indicates that the model successfully identifies most dummy views and avoids false negatives.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.3}$$

Accuracy measures the overall correctness of the predictions. It is the proportion of correctly classified instances (both positive and negative) over all instances. TN (true negatives) represents valid views correctly classified as such. In the presence of class imbalance (i.e., one class has more instances than the other), accuracy can be misleading, as it generally favors the larger class.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.4}$$

The F1 score is the harmonic mean of precision and recall. It provides a more balanced metric than accuracy when classes are imbalanced and it accounts for both false positives and false negatives.

The second evaluation task, node classification (see Section 5.3), is a multi-class classification problem. The objective is to predict the correct type of a target node based on its context. Due to the multi-class setting and class imbalance in the dataset, additional metrics are used to assess per-class performance and account for label skew.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{\sum_{i=1}^{N} 1(y_i = \hat{y}_i)}{N} \tag{5.5}$$

Accuracy in the multi-class setting measures the proportion of correctly predicted labels over all N instances. y_i is the true label and $\hat{y}_i$ is the predicted label for instance i. However, as in the binary case, accuracy is sensitive to class imbalance and may be biased toward dominant classes.

$$\text{Balanced Accuracy} = \frac{1}{C} \sum_{i=1}^{C} \frac{TP_i}{TP_i + FN_i} \tag{5.6}$$

Balanced accuracy computes the average recall (true positive rate) of all C classes. TP_i and FN_i denote the number of true positives and false negatives for each class i, respectively. This metric compensates for imbalances by giving equal weight to all classes.

$$\text{F1-Macro} = \frac{1}{C} \sum_{i=1}^{C} \text{F1}_i \tag{5.7}$$

Macro-averaged F1 computes the unweighted mean of all per-class F1 scores, treating all classes equally regardless of their frequency.

$$\text{F1-Weighted} = \sum_{i=1}^{C} w_i \cdot \text{F1}_i \tag{5.8}$$

The weighted F1 adjusts each class-specific F1 score by its support w_i (i.e., the proportion of instances belonging to class i). This metric balances class performance while accounting for the true class distribution.

$$\text{Precision}_{\text{macro}} = \frac{1}{C} \sum_{i=1}^{C} \frac{TP_i}{TP_i + FP_i} \tag{5.9}$$

Macro-averaged precision calculates the average precision over all classes, where FP_i denotes the false positives for class i. It reflects how well each class is identified independently of frequency.

$$\text{Recall}_{\text{macro}} = \frac{1}{C} \sum_{i=1}^{C} \frac{TP_i}{TP_i + FN_i} \tag{5.10}$$

Macro-averaged recall measures the average sensitivity across classes, showing how thoroughly each class is captured by the model.

Hardware & Tooling

All experiments were conducted on standard consumer-grade hardware to demonstrate practicality. The used machine has an Apple M1 Pro processor and 16 GB of RAM, running macOS Sequoia (version 15.3.2). GPU acceleration was not used during model training.

For encoding the ArchiMate models, the CM2ML CLI was used with Node.js version 20.11.0. All encodings were created with the corresponding encoding command and respective parameters, with each configuration of encoder parameters resulting in a dedicated JSON output file. If the encoded output exceeded practical file size limits, it was split across multiple files. More details on the exact CLI commands used are provided in the respective evaluation sections. In the second evaluation scenario (node classification), the triples encoder was used with word embeddings, which requires downloading pre-trained embedding files. This is supported via an auxiliary script included in the repository.

For model training and evaluation, Python 3.12.9 was used together with libraries from the Python data science ecosystem: `scikit-learn` for model implementation and evaluation, `NumPy` and `pandas` for data handling, and `matplotlib` and `seaborn` for generating visualizations and plots. To ensure reproducibility, all experiments were executed with fixed random seeds, where applicable. The entire evaluation workflow is implemented via standalone Python scripts, which are available in the public GitHub repository accompanying this thesis.

5.2 Evaluation 1: Dummy Classification

The first evaluation investigates how well TF-based semantic encodings can be used to automatically identify dummy views in ArchiMate models. Dummy views are diagrams that do not convey meaningful information but instead serve as placeholders, incomplete drafts, or artifacts created for testing purposes. Automatic detection of such views can be beneficial for large-scale model repository management, quality assurance, and data preprocessing.

The task is formulated as a binary classification problem, where each view is classified as either valid or dummy, based on its encoding. The evaluation explores how different encoder configurations and ML models perform on this task. In particular, it analyzes the effect of various parameter settings on classification accuracy, using a manually annotated subset of the EA ModelSet dataset as ground truth.

The following subsections detail the task setup, parameter configurations, and evaluation results.

5.2.1 Task Definition & Evaluation Setup

The goal of this evaluation is to assess how well term-frequency-based encodings of model views can be used to automatically distinguish between valid and dummy views. Dummy views are placeholders that do not reflect meaningful enterprise architecture content and are commonly introduced for quick sketches or testing purposes. Automatically detecting such views enables improved filtering and quality assurance in large-scale model repositories.

The task is formulated as a binary classification problem with supervised learning, where the input is a frequency-based term representation of a model view, and the output is a binary label indicating whether the view is considered a dummy (1) or valid (0). The labels used for this evaluation were manually annotated by modeling experts as part of the dataset construction process (see Section 5.1.1).

The input representations are generated using the TF encoder (see Section 4.3), which transforms each model view into a vector of term frequency values. Only uni-gram terms are considered in this evaluation with different parameter configurations of the encoder to examine the impact of various preprocessing and feature selection steps, such as stemming, tokenization, normalization, or the inclusion of type terms from elements and relationships.

Each encoding configuration is applied on the same set of models, and the output encoding is stored as a JSON file that contains the term-document matrices. In a classification pipeline, the output JSON files are loaded and used as input to various ML models. In particular, the classification pipeline involves the following steps:

1. **Loading and preprocessing**: Encoded term-document matrices and label files are loaded into `pandas` data frames. Views without labels are excluded and missing term values are replaced with zeros.
2. **Data splitting**: Each configuration is split into training and test sets using stratified sampling (80/20 split) to preserve class distribution.
3. **Hyperparameter tuning**: For each ML algorithm, relevant hyperparameters are optimized via 5-fold cross-validated grid search using F1-score as the primary metric.

4. **Model training and evaluation**: The best model for each algorithm from the previous step is evaluated again, using 5-fold cross-validation on the training set. Reported metrics are stored in a CSV file, including F1-score, accuracy, precision, and recall.

Four basic classifiers with different hyperparameters are evaluated for this task. The evaluated ML models and hyperparameter combinations are listed in Table 5.1.

ML Model	Hyperparameter	Values
Logistic Regression	`C`	{ 0.01, 0.1, 1, 10, 100 }
	`solver`	{ liblinear, lbfgs }
Support Vector Machine	`C`	{ 0.01, 0.1, 1, 10, 100 }
	`kernel`	{ linear, rbf }
Random Forest	`n_estimators`	{ 50, 100, 200 }
	`max_depth`	{ None, 10, 20, 30 }
K-Nearest Neighbors	`n_neighbors`	{ 3, 5, 7, 9 }
	`weights`	{ uniform, distance }

Table 5.1: ML Models and Hyperparameters used for Evaluation 1

5.2.2 Parameter Configurations

In this task, different configurations of the TF encoder were evaluated to explore how the parameters affect classification performance. The tested configurations are organized into two levels: *(i)* configuration groups, which define the types of semantic information included in the term representation, and *(ii)* parameter variants, which define how this information is processed and weighted. Each configuration group defines a different set of encoded concepts (e.g., element names, element types, and relationship types), while the parameter variants apply different normalization, preprocessing, and weighting techniques. This two-dimensional organization enables an isolated analysis of how higher-level content and lower-level transformations each contribute to model performance.

Three configuration groups were defined, as summarized in Table 5.2. For each group, 16 parameter variants were applied, which are listed in Table 5.3. This combination results in 48 total configurations (3 groups × 16 variants). For example, the configuration ID `C_n_1` only includes element names and does not apply any additional parameters, while `C_ntr_16` includes element names, types, and relationships with all four parameters applied. Also note that each configuration is evaluated on all four ML models with different hyperparameter configurations, resulting in even more combinations and longer execution time.

Group Prefix	Names as Terms	Types as Terms	Relationships as Nodes
C_**n**_<*ID*>	✓	✗	✗
C_**nt**_<*ID*>	✓	✓	✗
C_**ntr**_<*ID*>	✓	✓	✓

Table 5.2: Configuration Groups of Evaluation 1

Config ID	Normalize TF	Stem	Tokenize	TF-IDF
C_<*group*>_1	✗	✗	✗	✗
C_<*group*>_2	✓	✗	✗	✗
C_<*group*>_3	✗	✓	✗	✗
C_<*group*>_4	✗	✗	✓	✗
C_<*group*>_5	✗	✗	✗	✓
C_<*group*>_6	✓	✓	✗	✗
C_<*group*>_7	✓	✗	✓	✗
C_<*group*>_8	✓	✗	✗	✓
C_<*group*>_9	✗	✓	✓	✗
C_<*group*>_10	✗	✓	✗	✓
C_<*group*>_11	✗	✗	✓	✓
C_<*group*>_12	✓	✓	✓	✗
C_<*group*>_13	✓	✓	✗	✓
C_<*group*>_14	✗	✓	✓	✓
C_<*group*>_15	✓	✗	✓	✓
C_<*group*>_16	✓	✓	✓	✓

Table 5.3: Parameter Variants of Evaluation 1

5.2.3 Results & Discussion

The results of the dummy view classification evaluation are summarized in Figure 5.3. The figure shows the F1 scores for all encoder configurations and ML models. Each subplot corresponds to one ML model, and each line represents the F1 scores (y-axis) achieved by one of the three encoder configuration groups (`C_n`, `C_nt`, and `C_ntr`), plotted over the 16 parameter variants (x-axis). In total, 192 combinations (48 encoder configurations × 4 ML models) were evaluated.

The evaluation confirms that the choice of encoding configuration has a substantial impact on classification performance. Across all ML models, configurations that include semantic type information (`C_nt` and `C_ntr`) consistently outperform the baseline group

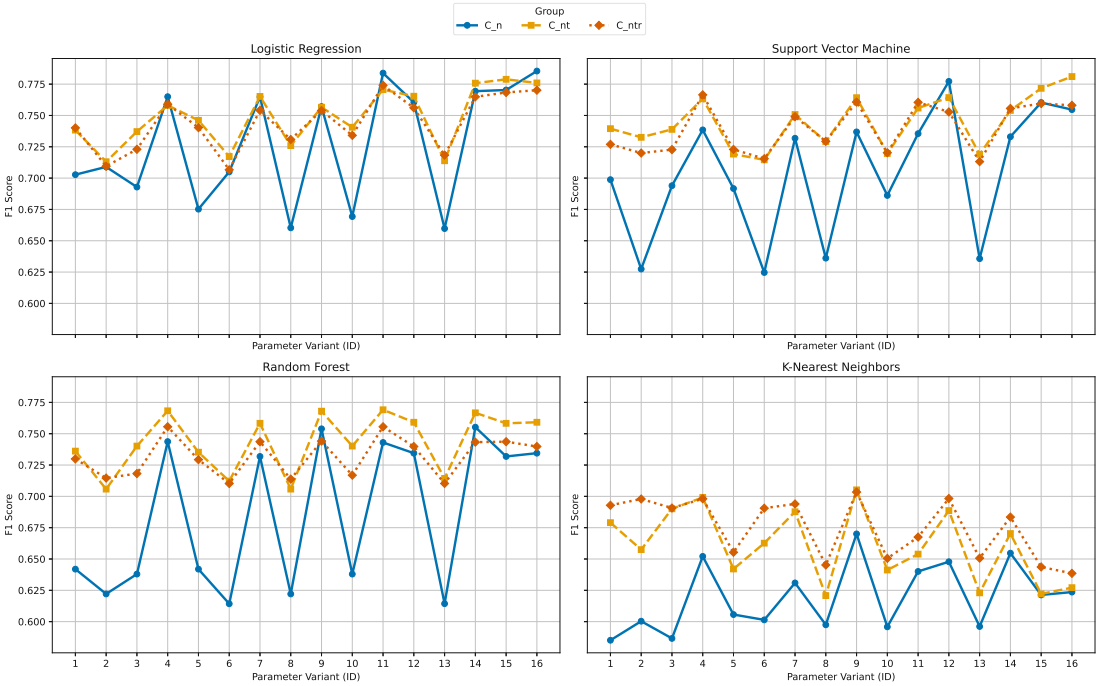


Figure 5.3: Evaluation 1 Results

C_n, which encodes only element names. This trend holds for nearly all parameter variants, suggesting that the inclusion of element and relationship types is beneficial for distinguishing dummy views.

Parameter Influence: Among the evaluated encoder parameters, tokenization appears to be the most influential. Configurations where tokenization is disabled (i.e., parameter variants 1-3, 5-6, 8, 10, and 13) consistently performed worse, particularly within the C_n group, as can be seen through the lower spikes in the plots. This suggests that breaking down element names into smaller lexical units significantly improves model performance, likely by reducing sparsity and improving generalization. TF-IDF weighting also proved beneficial, particularly in combination with tokenization and type information as evidenced by consistently high performance in later parameter variants (especially IDs 11–16). In contrast, normalization and stemming showed less consistent impact, contributing marginal improvements in some configurations but not significantly affecting ranking in the top results.

Best Performing Configurations: Table 5.4 lists the top 10 configurations sorted by F1 score. The best result was achieved using LR with configuration C_n_16 (F1 = 0.7854), followed closely by C_n_11 and C_nt_16. Notably, 10 out of the top 10 configurations apply tokenization and 9 apply TF-IDF weighting, confirming the positive effect of these parameters. Although one might expect richer configurations like C_ntr

to dominate, several top results come from `C_n` and `C_nt`, suggesting that optimal parameter tuning can sometimes compensate for less semantic input.

Config ID	ML Model	F1	Accuracy	Precision	Recall
C_n_16	LR	0.7854	0.8305	0.8031	0.7693
C_n_11	LR	0.7838	0.8197	0.7591	0.8114
C_nt_16	SVM	0.7811	0.8251	0.7948	0.7693
C_nt_15	LR	0.7788	0.8323	0.8345	0.7315
C_n_12	SVM	0.7772	0.8242	0.7952	0.7604
C_nt_14	LR	0.7758	0.8143	0.7580	0.7958
C_ntr_11	LR	0.7741	0.8135	0.7578	0.7915
C_nt_11	LR	0.7706	0.8117	0.7594	0.7826
C_n_15	LR	0.7703	0.8072	0.7408	0.8047
C_ntr_16	LR	0.7702	0.8135	0.7670	0.7737

Table 5.4: Top 10 Configurations by F1 Score of Evaluation 1

ML Model Comparison: LR was the overall best-performing model, achieving the highest F1 scores in 7 of the top 10 configurations and in 32 out of 48 configurations overall. SVMs also performed well and frequently matched or approached the performance of LR (highest F1 score in 13 out of 48 configurations). RFs produced more variable results, achieving competitive scores in a few configurations but rarely outperforming the top linear models (highest F1 score in 3 out of 48 configurations). The KNN algorithm consistently underperformed, which is expected given its sensitivity to feature dimensionality and sparsity of the data.

In summary, the results indicate that both semantic content and text processing parameters substantially influence classification performance. The inclusion of element and relationship types generally improves accuracy in all models, and tokenization and TF-IDF weighting are critical parameters for effective feature construction. Although LR and SVMs are the most effective models, the encoder configuration itself is the main determinant of success.

5.3 Evaluation 2: Node Classification

The second evaluation investigates the effectiveness of semantic encoding strategies for supporting multi-class classification tasks in conceptual models. Specifically, the task focuses on predicting the correct type of a target node within an ArchiMate model, given the structural and semantic context provided by a triple (source element, relationship, target element). This type of prediction has practical applications in modeling tools, where it could assist users during model construction by recommending element types based on partial input.

The evaluation is conducted using the triples encoder, which transforms each triple into a fixed-length feature vector that includes word embeddings and one-hot representations of structural metadata. The resulting dataset is used to train and evaluate different ML classifiers. Eight encoder configurations are tested, varying in the choice of word embeddings and OOV handling strategies. Three ML models are considered: LR, RF, and MLP.

The following subsections detail the task setup, parameter configurations, and evaluation results.

5.3.1 Task Definition & Evaluation Setup

The second evaluation task focuses on node classification in ArchiMate models with the goal to predict the type of a target node based on its structural and semantic context. This task is framed as a multi-class classification problem and is relevant both from a modeling and encoding perspective. From a modeling perspective, the classification can be integrated as a lightweight approach into practical applications such as model auto-completion or intelligent modeling assistants. From an encoding perspective, the task evaluates whether the semantics and context of a model element can be effectively captured and what impact different embedding models and OOV strategies have.

Each prediction instance is derived from a triple consisting of a source node, a relationship, and a target node. The triples encoder (see Section 4.5) is used to transform each triple into a fixed-length feature vector. The resulting vector encodes: *(i)* the embedding of the source node's name, *(ii)* a one-hot vector for the source node's type, *(iii)* a one-hot vector for the relationship type, and *(iv)* the embedding of the target node's name. These components are concatenated into a single input vector. The target label is the type of the target node, which can be one of 63 ArchiMate types and is also represented as a one-hot vector.

In the evaluation procedure, the dataset is prepared by parsing all triples from the encoded models. Only triples with complete information are retained and classes with fewer than five samples are excluded to ensure sufficient quantity for supervised learning. The resulting dataset is then stratified and split into training and test sets (80/20). All ML models are evaluated using 5-fold stratified cross-validation on the training set, with grid search for hyperparameter optimization. Macro-averaged F1 score is used as the primary selection metric.

Three classifiers are considered for this task: LR, RF, and MLP. Table 5.5 lists the models and hyperparameters used in the grid search for each model. Standardization is applied where appropriate, and early stopping is enabled with an adaptive learning rate for MLP training.

ML Model	Hyperparameter	Values
Logistic Regression	C	{ 0.1, 1.0 }
	penalty	{ l2 }
	solver	{ lbfgs }
	class_weight	{ None, balanced }
Random Forest	n_estimators	{ 100 }
	max_depth	{ None, 20 }
	class_weight	{ None, balanced }
MLP Classifier	hidden_layer_sizes	{ (256,), (512, 256) }
	alpha	{ 1e-4 }

Table 5.5: ML Models and Hyperparameters used for Evaluation 2

5.3.2 Parameter Configurations

All models in this evaluation are encoded using the triples encoder (see Section 4.5). The evaluation specifically focuses on the influence of word embedding parameters, which are expected to impact how well semantic information is captured in the resulting feature vectors.

Table 5.6 lists the evaluated configurations and their corresponding parameter combinations. Two parameters were varied to examine their influence on model performance. The first parameter, `embeddingsModel`, determines which pre-trained word embedding model is used to encode element names. Four models, as currently supported in the encoder, were evaluated: `glove` and `word2vec`, both trained on general-domain corpora (Wikipedia + Gigaword and Google News, respectively), and two domain-specific variants, `glove-mde` and `word2vec-mde`, trained on modeling-specific texts (i.e., papers from well-known modeling conferences, e.g., MODELS, SoSyM, ER). The second parameter, `oovStrategy`, controls how OOV terms are handled during encoding. Two strategies were tested: the `zero` strategy, which replaces unknown terms with a zero vector, and the `most-similar` strategy, which substitutes an OOV token with the embedding of the most similar known word based on Levenshtein distance.

In addition to the varied parameters, a set of base parameters was used in all configurations. In particular, type information for both source and target elements is included through `includeTypes`. This type information is encoded using one-hot vectors over the complete set of 63 ArchiMate types, through `typesAsOneHot`. Word embeddings are enabled for both source and target element names through `useEmbeddings`. For multi-word names, `average` is used as the `combineWordsStrategy`, where individual word embeddings are combined by averaging to produce a single fixed-length vector. The

Configuration	Embeddings Model	OOV Strategy
C1	glove	zero
C2	glove	most-similar
C3	word2vec	zero
C4	word2vec	most-similar
C5	glove-mde	zero
C6	glove-mde	most-similar
C7	word2vec-mde	zero
C8	word2vec-mde	most-similar

Table 5.6: Encoding Configurations of Evaluation 2

option `includeSourceId` was disabled in all configurations, as traceability was not required and its exclusion reduced file sizes.

Each configuration was applied to the full dataset using the CM2ML CLI. To manage memory and file size constraints, each run was split into three segments using the `--start` and `--limit` flags. An example command to encode the first segment of configuration C1 is shown in Listing 5.1.

```
cm2ml batch-archimate-triples ./models/
   --start 0 --limit 200
   --out ml/.output/triples/C1/C1_1.json
   --strict --continue-on-error
   --no-include-source-id
   --include-types
   --types-as-one-hot
   --use-word-embeddings
   --embeddings-model glove
   --combine-words-strategy average
   --oov-strategy zero
```

Listing 5.1: CLI Command to create the first segment of Configuration C1

Since word embeddings rely on vocabulary lookups, vocabulary coverage of tokens from the dataset plays a critical role in the quality of the resulting vector representations. Table 5.7 shows the coverage of each embedding model on the dataset vocabulary. The *Tokens* column refers to the total number of tokens in the corpus used for training, whereas *Vocab. Size* denotes the final vocabulary, containing only unique tokens. The *Coverage* column shows how many tokens of the dataset vocabulary are covered, which includes 405,825 tokens in total. All models achieve coverage above 95%, with general-domain models (`glove` and `word2vec`) performing slightly better due to their larger vocabulary size. The domain-specific models (`glove-mde` and `word2vec-mde`), however, may offer improved semantic alignment with modeling-specific terminology.

Model	Corpus	Tokens	Vocab. Size	Coverage
glove	Wikipedia+Gigaword	~6B	400,000	396,188 (97.63%)
word2vec	Google News	~100B	3,000,000	391,294 (96.42%)
glove-mde	Modeling Texts	~40M	55,519	388,416 (95.71%)
word2vec-mde	Modeling Texts	~40M	55,519	388,416 (95.71%)

Table 5.7: Vocabulary Coverage of Word Embedding Models

5.3.3 Results & Discussion

The results of the node classification task are summarized in Table 5.8, which presents the performance metrics for all eight encoder configurations and three ML models. Overall, the evaluation shows that the choice of ML model has a significantly larger influence on performance than the specific encoder parameters. RF classifiers consistently outperform the other models in all configurations, achieving the highest scores for accuracy, macro F1, and weighted F1. MLP classifiers perform slightly worse, but still deliver competitive results. LR, while simpler and faster, consistently ranks lowest, due to its limited ability to capture the complex relationships within the input representations.

The best-performing result is achieved with configuration C4 using a RF classifier. This configuration reaches a weighted F1 score of 0.884 and an accuracy of 0.884, making it the most effective overall. However, the differences between configurations are generally small. In fact, most configurations using RF achieve a weighted F1 score greater than 0.87 with variations rarely exceed 0.02. These findings suggest that the models are robust to small changes in encoder parameters, and that the inclusion of semantic and structural features via the triples encoder provides a sufficiently rich signal for classification.

When looking at the impact of encoder parameters, no clear pattern emerges. Configurations using general-purpose embeddings (glove and word2vec) often slightly outperform those using domain-specific embeddings (glove-mde and word2vec-mde). This may be attributed to the larger vocabulary coverage of general-purpose models, which can better encode rare or compound terms. Although domain-specific embeddings might capture semantic similarity better, their smaller vocabulary may lead to higher rates of OOV tokens, reducing their practical advantage in this setting.

Similarly, the choice of `oovStrategy` (i.e., zero vs. most-similar) shows no consistent effect on performance. In some cases, the `most-similar` strategy slightly decreases performance, likely due to introducing noise when replacements with similar terms are semantically not aligned. Nevertheless, the differences are minimal, often within a 0.001–0.003 range in weighted F1. This suggests that the classifiers are relatively robust to the handling of unknown tokens, provided that a sufficient amount of contextual information is available through the remaining input features.

To better understand model behavior, Figure 5.4 shows per-class F1 scores and support values for the "best configuration" (C4 with RF). Note the use of two different x-axis for *Support* (top x-axis) and *F1 Score* (bottom x-axis). The plot shows a wide

Configuration	Model	Acc	Bal Acc	F1-Ma	F1-W	Prec	Rec
C1 (glove, zero)	LR	0.778	0.715	0.732	0.777	0.753	0.715
	RF	0.881	0.801	0.841	0.880	0.904	0.801
	MLP	0.860	0.772	0.784	0.860	0.812	0.772
C2 (glove, most-similar)	LR	0.770	0.702	0.724	0.769	0.759	0.702
	RF	0.881	0.790	0.829	0.881	0.892	0.790
	MLP	0.853	0.753	0.776	0.852	0.812	0.753
C3 (word2vec, zero)	LR	0.780	0.718	0.737	0.779	0.764	0.718
	RF	0.883	0.794	0.832	0.882	0.894	0.794
	MLP	0.860	0.770	0.786	0.860	0.813	0.770
C4 (word2vec, most-similar)	LR	0.771	0.684	0.705	0.770	0.734	0.684
	RF	0.884	0.788	0.830	0.884	0.889	0.788
	MLP	0.855	0.724	0.755	0.854	0.809	0.724
C5 (glove-mde, zero)	LR	0.779	0.717	0.740	0.778	0.774	0.717
	RF	0.882	0.789	0.830	0.882	0.892	0.789
	MLP	0.850	0.759	0.778	0.849	0.809	0.759
C6 (glove-mde, most-similar)	LR	0.769	0.712	0.733	0.768	0.774	0.712
	RF	0.881	0.799	0.843	0.881	0.906	0.799
	MLP	0.852	0.771	0.786	0.852	0.811	0.771
C7 (word2vec-mde, zero)	LR	0.778	0.723	0.738	0.778	0.758	0.723
	RF	0.878	0.800	0.840	0.877	0.903	0.800
	MLP	0.858	0.770	0.780	0.858	0.799	0.770
C8 (word2vec-mde, most-similar)	LR	0.770	0.671	0.695	0.769	0.732	0.671
	RF	0.878	0.771	0.818	0.877	0.886	0.771
	MLP	0.850	0.736	0.753	0.849	0.787	0.736

Table 5.8: Evaluation 2 Results

variation in prediction performance per classes. While some classes with higher support also achieve high F1 scores, this is not always the case. For example, classes such as *ApplicationComponent* or *BusinessProcess* have support values greater than 1000, but still exhibit relatively low F1 scores. Conversely, some classes with moderate or low support show comparatively strong performance. This indicates that the model's ability to correctly classify a type is influenced by more than just class frequency, including factors such as feature distinctiveness and semantic overlap with other types.

In summary, this evaluation demonstrates that the triples encoder provides effective input representations for multi-class node classification, without requiring manually labeled data. RFs are the most robust classifier in this setting, with minor sensitivity to embedding parameters. Domain-specific embeddings and OOV handling strategies show limited influence on overall results, and class-level prediction remains difficult in the presence of overlapping semantics and imbalanced label distributions.

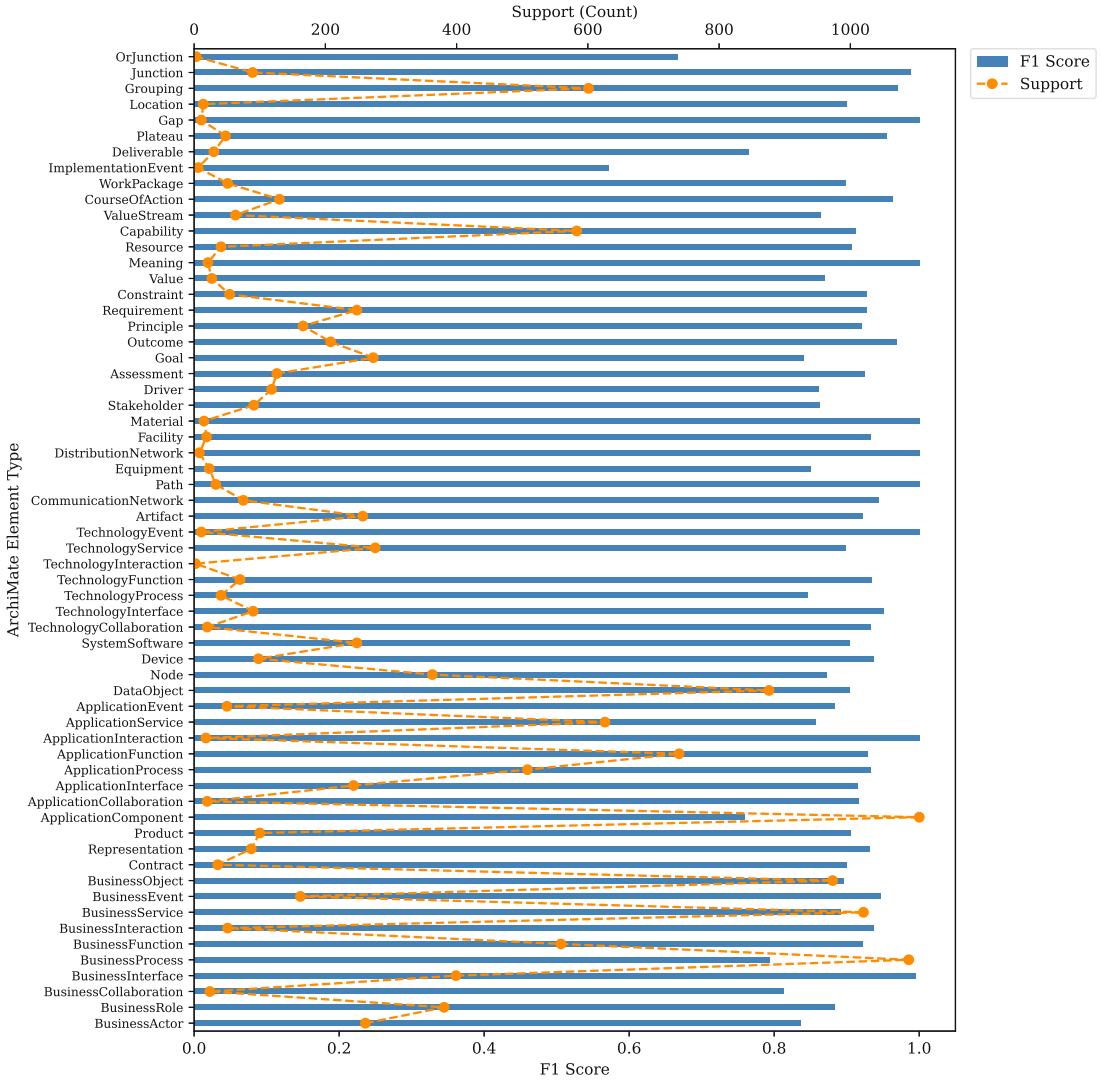


Figure 5.4: Evaluation 2: Per-class F1 Scores and Support for Configuration C4

CHAPTER 6

Conclusion

This chapter concludes the thesis by summarizing the main findings and reflecting on its contributions, limitations, and implications for future research. Section 6.1 revisits the motivation, technical developments, and key results, providing a consolidated overview of how the thesis addressed the problem of encoding semantic information in conceptual models. Section 6.2 discusses current limitations and proposes directions for future work.

6.1 Summary

This thesis addressed the problem of how to systematically encode semantic information in conceptual models for use in ML applications. Although conceptual models contain diverse information sources (e.g., names, types, context, metamodel semantics, etc.), existing ML4CM approaches often focus on narrow use cases that ignore different types of information in their evaluation. This work aims to close this gap by introducing configurable semantic encoding strategies that can be systematically evaluated to ensure optimal performance during training.

To support this goal, the CM2ML framework was extended in several ways. An ArchiMate parser was developed to process two widely used XML formats with support for configurable parameters, such as view separation, element filtering, and relationship handling. The framework was also enhanced with reusable NLP utilities, including a term extractor and helper modules for tokenization, embedding retrieval, and similarity matching.

Building on this foundation, four semantic encoders were implemented: *(i)* a BoW encoder (supporting tokens and sentences), *(ii)* a TF encoder (supporting uni-, bi-, and n-grams), *(iii)* an embeddings encoder (mapping terms to pre-trained word vectors, with optional pooling), and *(iv)* a triples encoder (extracting relational triples with optional

type encodings and embedding vectors). These encoders were compared in Section 4.6 using a set of qualitative criteria that reveal their respective strengths and trade-offs.

To complement the qualitative analysis, two quantitative experiments were conducted (Chapter 5), showing the feasibility of the implemented encoders for practical ML tasks and how parameter choices influence performance. The first task, dummy view classification, evaluated TF-based encodings and showed that configurations including tokenization and TF-IDF weighting achieved the best results. The second task, node classification, used the triples encoder to predict element types based on their relational context, combining one-hot type vectors and word embeddings. While performance differences across configurations were minor, the experiment demonstrated how the triples encoder supports multi-class classification without requiring manually labeled data.

6.2 Limitations & Future Work

While this thesis establishes a foundation for semantic encoding of conceptual models for ML, several limitations remain that offer promising directions for future work.

One limitation concerns the **coverage of semantic aspects**. The realized encoders target selected types of semantics (e.g., lexical, structural, distributional, contextual) but do not exhaustively capture all possible semantic dimensions present in conceptual models. For instance, metamodel semantics, domain ontologies, or behavioral/temporal aspects are not currently addressed. However, the relevance of specific semantic features is highly dependent on the target ML task, making it difficult to define a universally complete encoding. Future work should explore additional semantic dimensions and assess their usefulness in more diverse tasks.

Similarly, the **parameter space** of the implemented encoders is not comprehensive. Although key parameters were supported (e.g., basic tokenization, embedding model, n-gram configuration, etc.), many other options, such as zone-based term weighting, advanced OOV strategies, or alternative pooling techniques remain unexplored. Expanding the parameter space would allow for more fine-grained control and performance tuning.

Another conceptual limitation is the **separation of structural and semantic encodings**. In practice, many ML tasks benefit from hybrid representations that combine structural and semantic information. While this thesis focused on isolated encoder types to allow clearer comparison, future research should investigate multi-encoder or hybrid encoding strategies that combine, for example, graph-based representations with semantic vectors to leverage complementary strengths.

The **CM2ML framework** itself poses certain limitations in terms of integration and extensibility. It is implemented in TypeScript within a Node.js environment, which restricts compatibility with the broader ML ecosystem that is predominantly Python-based. Moreover, on-demand encoding is often necessary in ML workflows, and many advanced NLP or embedding libraries are not available in the JavaScript ecosystem. As a result, the implementation of components such as the embeddings encoder required

workarounds. A future reimplementation in Python could alleviate these limitations and better support experimentation with state-of-the-art ML tools.

In terms of **evaluation scope**, this thesis quantitatively assessed only two encoders (TF and triples) and explored a limited set of parameter configurations. A broader evaluation involving all encoder types, a more comprehensive parameter grid, and additional datasets would provide deeper insight into the generalizability and robustness of each encoding strategy. Also, the **ML models** used in the experiments were limited to traditional approaches (e.g., LR, SVM, RF, KNN, MLP). Future work could include more advanced models, such as transformer-based classifiers or GNNs, particularly for encoders that capture sequential or relational information.

Furthermore, the evaluation included only **two ML tasks**, dummy view classification and node classification, both based on supervised learning methods. Other relevant tasks such as clustering, link prediction, or anomaly detection remain untested. Extending the evaluation to these tasks could help uncover additional strengths or weaknesses of the encoders.

Finally, there are **limitations related to the dataset**. The first classification task relied on a subset of labeled models, and label coverage was incomplete. Moreover, since the semantic encodings depend heavily on element names, the quality and consistency of these names can directly influence performance. Inconsistent or poorly named elements may lead to degraded results. Future work should involve curating and annotating richer datasets with high-quality naming conventions and labels, and testing the encoders on additional modeling languages such as UML or Ecore to validate their generalizability beyond ArchiMate.

Overview of Generative AI Tools Used

ChatGPT[1] (GPT-4o model) was used during the preparation of this thesis to improve writing quality, specifically in terms of language clarity and stylistic refinement.

The use of ChatGPT did not affect the originality or academic integrity of the work, since it functioned solely as a writing assistant, comparable to advanced language editing software. All conceptual work, critical analysis, and final formulations were carried out independently by the author.

[1] `https://chatgpt.com/` (Accessed: 30.05.2025)

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 1, 10

API Application Programming Interface. 21, 23, 24, 27, 29, 30

ATL ATLAS Transformation Language. 8

BoP Bag-of-Paths. 16, 17

BoW Bag-of-Words. ix, xi, 3, 5, 15, 22, 26, 35, 37–39, 41–45, 61–63, 65, 87, 93, 95

BPMN Business Process Model and Notation. 8, 16–18, 68

CKG Conceptual Knowledge Graph. 17

CLI Command Line Interface. 21–25, 27, 29, 30, 36, 50, 56, 57, 69, 71, 74, 82, 93

CM Conceptual Modeling. ix, xi, 1, 4–7, 10, 15, 16, 18, 67, 69

CM2ML Conceptual Models to Machine Learning. ix, xi, 2–6, 21–30, 35, 36, 51, 71, 82, 87, 88, 93

CM4ML Conceptual Modeling for Machine Learning. 1

CNN Convolutional Neural Network. 16

CRF Conditional Random Field. 17

DSR Design Science Research. ix, 2

EA Enterprise Architecture. 8, 9, 68, 71, 75

EMF Eclipse Modeling Framework. 8

ER Entity-Relationship. 8

FAIR Findable, Accessible, Interoperable, Reusable. 68

FNN Feedforward Neural Network. 13, 15, 18, 19

GNN Graph Neural Network. 16, 17, 89

HAC Hierarchical Agglomerative Clustering. 15

IDF Inverse Document Frequency. 15, 46

IR Intermediate Representation. 3, 21, 25–32, 37, 39, 41, 43, 45, 49, 56, 93

JSON JavaScript Object Notation. 22, 24, 25, 28, 29, 36, 42, 71, 74, 75

KNN K-Nearest Neighbors. 12, 13, 79, 89

LR Logistic Regression. 11, 12, 64, 78–80, 83, 89

LSTM Long Short-Term Memory. 16, 17

MDE Model-Driven Engineering. 1, 7, 14

ML Machine Learning. ix, xi, 1–7, 10, 11, 13–18, 21, 26, 35, 41, 44, 47, 50, 55, 61–65, 67, 68, 70–72, 75–77, 79–81, 83, 87–89, 95

ML4CM Machine Learning for Conceptual Modeling. ix, xi, 1, 2, 4, 7, 13, 14, 87, 93

MLP Multi-Layer Perceptron. 13, 80, 81, 83, 89

NLP Natural Language Processing. ix, xi, 1, 5, 15–17, 39, 68, 87, 88

NN Neural Network. 11, 16–18, 64

OCL Object Constraint Language. 8

OOV Out-Of-Vocabulary. 35, 51, 52, 54–59, 80–84, 88

PWA Progressive Web Application. 30

REST REpresentational State Transfer. 21, 23–25, 27, 29, 30, 50–52

RF Random Forest. 12, 79, 80, 83, 84, 89

RQ Research Question. 4, 36, 61

SVM Support Vector Machine. 12, 18, 64, 79, 89

TF Term Frequency. ix, xi, 3, 5, 6, 15, 18, 24, 35, 37–39, 44–50, 61, 63–65, 67, 71, 74–77, 87–89, 93, 95

TF-IDF Term Frequency-Inverse Document Frequency. 15, 16, 25, 35, 38, 44, 46–48, 77–79, 88

UI User Interface. 30

UML Unified Modeling Language. 1, 4, 8, 17–19, 21, 22, 27, 68, 89

URL Uniform Resource Locator. 26

XAI eXplainable Artificial Intelligence. 14

XML Extensible Markup Language. 3, 5, 9, 24, 26, 27, 29–31, 68, 87

Bibliography

[AAZZ22] Alhassan Adamu, Salisu Mamman Abdulrahman, Wan Mohd Nazmee Wan Zainoon, and Abubakar Zakari. Model matching: Prediction of the influence of uml class diagram parameters during similarity assessment using artificial neural network. In *Deep Learning Approaches for Spoken and Natural Language Processing*, pages 97–109. Springer, 2022.

[AB24] Syed Juned Ali and Dominik Bork. A graph language modeling framework for the ontological enrichment of conceptual models. In Giancarlo Guizzardi, Flávia Maria Santoro, Haralambos Mouratidis, and Pnina Soffer, editors, *Advanced Information Systems Engineering - 36th International Conference, CAiSE 2024, Limassol, Cyprus, June 3-7, 2024, Proceedings*, volume 14663 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2024.

[AFc] Matías Capeletto Anthony Fu and Vitest contributors. Vitest. `https://vitest.dev/`. Accessed: 28-05-2025.

[AGB23] Syed Juned Ali, Giancarlo Guizzardi, and Dominik Bork. Enabling representation learning in ontology-driven conceptual modeling using graph neural networks. In Marta Indulska, Iris Reinhartz-Berger, Carlos Cetina, and Oscar Pastor, editors, *Advanced Information Systems Engineering - 35th International Conference, CAiSE 2023, Zaragoza, Spain, June 12-16, 2023, Proceedings*, volume 13901 of *Lecture Notes in Computer Science*, pages 278–294. Springer, 2023.

[AGPB23] Syed Juned Ali, Aleksandar Gavric, Henderik A. Proper, and Dominik Bork. Encoding conceptual models for machine learning: A systematic review. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1-6, 2023*, pages 562–570. IEEE, 2023.

[ARR25] Francisco Javier Alcaide, José Raúl Romero, and Aurora Ramírez. Can explainable artificial intelligence support software modelers in model comprehension? *Software and Systems Modeling*, pages 1–26, 2025.

[AX19] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019.

[BAR23] Dominik Bork, Syed Juned Ali, and Ben Roelens. Conceptual modeling and artificial intelligence: A systematic mapping study. *arXiv preprint arXiv:2303.06758*, 2023.

[BC17] Önder Babur and Loek Cleophas. Using n-grams for the automated clustering of structural models. In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, *SOFSEM 2017: Theory and Practice of Computer Science - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings*, volume 10139 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2017.

[BCG+21] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. An nlp-based architecture for the autocompletion of partial domain models. In Marcello La Rosa, Shazia Sadiq, and Ernest Teniente, editors, *Advanced Information Systems Engineering - 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28 - July 2, 2021, Proceedings*, volume 12751 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2021.

[BCLG22] Loli Burgueno, Jordi Cabot, Shuai Li, and Sébastien Gérard. A generic lstm neural network architecture to infer heterogeneous model transformations. *Software and Systems Modeling*, 21(1):139–156, 2022.

[BCPP20] 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(1):5–13, 2020.

[BCvdB16] Önder Babur, Loek Cleophas, and Mark van den Brand. Hierarchical clustering of metamodels for comparative analysis and visualization. In Andrzej Wasowski and Henrik Lönn, editors, *Modelling Foundations and Applications - 12th European Conference, ECMFA@STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*, volume 9764 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2016.

[BCVDB19] Önder Babur, Loek Cleophas, and Mark Van Den Brand. Metamodel clone detection with samos. *Journal of Computer Languages*, 51:57–74, 2019.

[BCVvdB16] Önder Babur, Loek Cleophas, Tom Verhoeff, and Mark van den Brand. Towards statistical comparison and analysis of models. In Slimane Hammoudi, Luís Ferreira Pires, Bran Selic, and Philippe Desfray, editors, *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016*, pages 361–367. SciTePress, 2016.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[BHI+20] Angela Barriga, Rogardt Heldal, Ludovico Iovino, Magnus Marthinsen, and Adrian Rutle. An extensible framework for customizable model repair. In *Proceedings of the 23rd ACM/IEEE International conference on model driven engineering languages and systems*, pages 24–34, 2020.

[BM22] Justus Bogner and Manuel Merkel. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 658–669, 2022.

[BRR+16] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Automated clustering of metamodel repositories. In Selmin Nurcan, Pnina Soffer, Marko Bajec, and Johann Eder, editors, *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*, volume 9694 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 2016.

[BSF+18] Andrea Burattin, Pnina Soffer, Dirk Fahland, Jan Mendling, Hajo A. Reijers, Irene Vanderfeesten, Matthias Weidlich, and Barbara Weber. Who is behind the model? classifying modelers based on pragmatic model features. In Mathias Weske, Marco Montali, Ingo Weber, and Jan vom Brocke, editors, *Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings*, volume 11080 of *Lecture Notes in Computer Science*, pages 322–338. Springer, 2018.

[Bö] Felix Böhm. htmlparser2. `https://www.npmjs.com/package/htmlparser2`. Accessed: 28-05-2025.

[CG12] Jordi Cabot and Martin Gogolla. Object constraint language (OCL): A definitive guide. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, volume 7320 of *Lecture Notes in Computer Science*, pages 58–90. Springer, 2012.

[Che76] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1):9–36, 1976.

[CT12] Michele Chinosi and Alberto Trombetta. BPMN: an introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.

[CvMG+14] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine

translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL, 2014.

[CYK+18] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[DLPS18] Lois M. L. Delcambre, Stephen W. Liddle, Oscar Pastor, and Veda C. Storey. A reference framework for conceptual modeling. In Juan Trujillo, Karen C. Davis, Xiaoyong Du, Zhanhuai Li, Tok Wang Ling, Guoliang Li, and Mong-Li Lee, editors, *Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings*, volume 11157 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2018.

[DRMR13] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.

[DS22] Paulius Danenas and Tomas Skersys. Exploring natural language processing in model-to-model transformations. *IEEE Access*, 10:116942–116958, 2022.

[EGB16] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. An UML class recommender system for software design. In *13th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2016, Agadir, Morocco, November 29 - December 2, 2016*, pages 1–8. IEEE Computer Society, 2016.

[EGO] EGOIST. Cac. `https://github.com/cacjs/cac`. Accessed: 28-05-2025.

[FKNW22] Reza Fauzan, Ice Krisnahati, Bima Dinda Nurwibawa, and Della Aulia Wibowo. A systematic literature review on progressive web application practice and challenges. *IPTEK The Journal for Technology and Science*, 33(1):43–58, 2022.

[Foua] OpenJS Foundation. Fastify. https://fastify.dev/. Accessed: 28-05-2025.

[Foub] OpenJS Foundation. Node.js. https://nodejs.org/. Accessed: 28-05-2025.

[FRK12] Hans-Georg Fill, Timothy Redmond, and Dimitris Karagiannis. FDMM: A formalism for describing adoxx meta models and models. In Leszek A. Maciaszek, Alfredo Cuzzocrea, and José Cordeiro, editors, *ICEIS 2012 - Proceedings of the 14th International Conference on Enterprise Information Systems, Volume 3, Wroclaw, Poland, 28 June - 1 July, 2012*, pages 133–144. SciTePress, 2012.

[FSG20] Mattia Fumagalli, Tiago Prince Sales, and Giancarlo Guizzardi. Towards automated support for conceptual model diagnosis and repair. In Georg Grossmann and Sudha Ram, editors, *Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings*, volume 12584 of *Lecture Notes in Computer Science*, pages 15–25. Springer, 2020.

[GG21] Maayan Goldstein and Cecilia González-Alvarez. Augmenting modelers with semantic autocompletion of processes. In Artem Polyvyanyy, Moe Thandar Wynn, Amy Van Looy, and Manfred Reichert, editors, *Business Process Management Forum - BPM Forum 2021, Rome, Italy, September 06-10, 2021, Proceedings*, volume 427 of *Lecture Notes in Business Information Processing*, pages 20–36. Springer, 2021.

[GLX+23] Youdi Gong, Guangzhen Liu, Yunzhi Xue, Rui Li, and Lingzhong Meng. A survey on dataset quality in machine learning. *Information and Software Technology*, 162:107268, 2023.

[Gro] The Open Group. Archimate 3.2 specification. https://pubs.opengroup.org/architecture/archimate32-doc/. Accessed: 28-05-2025.

[GSB23] Philipp-Lorenz Glaser, Emanuel Sallinger, and Dominik Bork. EA modelset - A FAIR dataset for machine learning in enterprise modeling. In João Paulo A. Almeida, Monika Kaczmarek-Heß, Agnes Koschmider, and Henderik A. Proper, editors, *The Practice of Enterprise Modeling - 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28 - December 1, 2023, Proceedings*, volume 497 of *Lecture Notes in Business Information Processing*, pages 19–36. Springer, 2023.

[GSB25] Philipp-Lorenz Glaser, Emanuel Sallinger, and Dominik Bork. The extended EA modelset—a FAIR dataset for researching and reasoning enterprise architecture modeling practices. *Software and Systems Modeling*, pages 1–19, 2025.

[GWAG15] Giancarlo Guizzardi, Gerd Wagner, João Paulo Andrade Almeida, and Renata SS Guizzardi. Towards ontological foundations for conceptual modeling: The unified foundational ontology (UFO) story. *Applied Ontology*, 10(3-4):259–271, 2015.

[HMPR04] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HYL17] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 1024–1034, 2017.

[Inc] VoidZero Inc. Vite. `https://vite.dev/`. Accessed: 28-05-2025.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: A QVT-like transformation language. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 719–720. ACM, 2006.

[JBQ12] Henk Jonkers, Iver Band, and Dick Quartel. The archisurance case study. *The Open Group*, pages 1–32, 2012.

[JJK+18] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K. Smith, Collin Winter, and Emerson R. Murphy-Hill. Advantages and disadvantages of a monolithic repository: a case study at google. In Frances Paulisch and Jan Bosch, editors, *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 225–234. ACM, 2018.

[JM15] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[Kat] Daishi Kato. zustand. `https://github.com/pmndrs/zustand`. Accessed: 28-05-2025.

[Lan17] Marc M. Lankhorst, editor. *Enterprise Architecture at Work - Modelling, Communication and Analysis, Fourth Edition*. Springer, 2017.

[LC20] José Antonio Hernández López and Jesús Sánchez Cuadrado. MAR: a structure-based search engine for models. In Eugene Syriani, Houari A. Sahraoui, Juan de Lara, and Silvia Abrahão, editors, *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*, pages 57–67. ACM, 2020.

[LC21] José Antonio Hernández López and Jesús Sánchez Cuadrado. Towards the characterization of realistic model generators using graph neural networks. In *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10-15, 2021*, pages 58–69. IEEE, 2021.

[LC22] José Antonio Hernández López and Jesús Sánchez Cuadrado. An efficient and scalable search engine for models. *Software and Systems Modeling*, 21(5):1715–1737, 2022.

[LCIC22] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. Modelset: a dataset for machine learning in model-driven engineering. *Software and Systems Modeling*, pages 1–20, 2022.

[LDC23] José Antonio Hernández López, Carlos Durá, and Jesús Sánchez Cuadrado. Word embeddings for model-driven engineering. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*, pages 151–161. IEEE, 2023.

[LDC24] José Antonio Hernández López, Carlos Durá, and Jesús Sánchez Cuadrado. Experimenting with modeling-specific word embeddings. *Software and Systems Modeling*, pages 1–23, 2024.

[LIC22] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. Using the modelset dataset to support machine learning in model-driven engineering. In Thomas Kühn and Vasco Sousa, editors, *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*, pages 66–70. ACM, 2022.

[LIC24] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. Modelset: A labelled dataset of software models for machine learning. *Science of Computer Programming*, 231:103009, 2024.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov.

RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[LOZ+21] Gelareh Meidanipour Lahijany, Manuel Ohrndorf, Johannes Zenkert, Madjid Fathi, and Udo Kelter. Identibug: Model-driven visualization of bug reports by extracting class diagram excerpts. In *2021 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2021, Melbourne, Australia, October 17-20, 2021*, pages 3317–3323. IEEE, 2021.

[LPJ10] Marc Lankhorst, Henderik Proper, and Henk Jonkers. The anatomy of the archimate language. *IJISMD*, 1:1–32, 01 2010.

[LRCR22] José Antonio Hernández López, Riccardo Rubei, Jesús Sánchez Cuadrado, and Davide Di Ruscio. Machine learning methods for model classification: a comparative study. In Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer, editors, *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*, pages 165–175. ACM, 2022.

[MBWM24] Judith Michael, Dominik Bork, Manuel Wimmer, and Heinrich C. Mayr. Quo vadis modeling? *Software and Systems Modeling*, 23(1):7–28, 2024.

[McD] Colin McDonnell. Zod. `https://zod.dev/`. Accessed: 28-05-2025.

[Mic] Microsoft. Typescript. `https://www.typescriptlang.org/`. Accessed: 28-05-2025.

[MIL+24] Ana C Marcén, Antonio Iglesias, Raúl Lapeña, Francisca Pérez, and Carlos Cetina. A systematic literature review of model-driven engineering using machine learning. *IEEE Transactions on Software Engineering*, 2024.

[Mit97] Tom M Mitchell. Does machine learning really work? *AI magazine*, 18(3):11–11, 1997.

[MP] Inc Meta Platforms. React. `https://react.dev/`. Accessed: 28-05-2025.

[MRRR02] Nenad Medvidovic, David S Rosenblum, David F Redmiles, and Jason E Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):2–57, 2002.

[MS21] Wolfgang Maass and Veda C Storey. Pairing conceptual modeling with machine learning. *Data & Knowledge Engineering*, 134:101909, 2021.

[MSC+13] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119, 2013.

[Mül24] Jan Müller. Cm2ml: A generic, portable, and extensible framework for encoding the structure of conceptual models. Master's thesis, Technische Universität Wien, 2024.

[NDRI+21] Phuong T Nguyen, Juri Di Rocco, Ludovico Iovino, Davide Di Ruscio, and Alfonso Pierantonio. Evaluation of a machine learning classifier for metamodels. *Software and Systems Modeling*, 20(6):1797–1821, 2021.

[NDRP+21] Phuong T Nguyen, Davide Di Ruscio, Alfonso Pierantonio, Juri Di Rocco, and Ludovico Iovino. Convolutional neural networks for enhanced classification mechanisms of metamodels. *Journal of Systems and Software*, 172:110860, 2021.

[NRR+19] Phuong Thanh Nguyen, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. Automated classification of metamodel repositories: A machine learning approach. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019*, pages 272–282. IEEE, 2019.

[OCvdP13] Mohd Hafeez Osman, Michel R. V. Chaudron, and Peter van der Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 140–149. IEEE Computer Society, 2013.

[OE20] Mert Ozkaya and Ferhat Erata. A survey on the practical use of UML for different software architecture viewpoints. *Information and Software Technology*, 121:106275, 2020.

[Oli07] Antoni Olivé. *Conceptual modeling of information systems*. Springer Science & Business Media, 2007.

[OMK20] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, 32(2):604–624, 2020.

[Ove] Oven. Bun. `https://bun.sh/`. Accessed: 28-05-2025.

[PK17] Myeongsuk Pak and Sanghoon Kim. A review of deep learning in image recognition. In *2017 4th international conference on computer applications and information processing technology (CAIPT)*, pages 1–3. IEEE, 2017.

[Por80] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[PTRC07] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[RAB+15] Stewart Robinson, Gilbert Arbez, Louis G. Birta, Andreas Tolk, and Gerd Wagner. Conceptual modeling: definition, purpose and benefits. In *Proceedings of the 2015 Winter Simulation Conference, Huntington Beach, CA, USA, December 6-9, 2015*, pages 2812–2826. IEEE/ACM, 2015.

[Řeh10] R Řehřek. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, page 45, 2010.

[RRIG09] Jan Recker, Michael Rosemann, Marta Indulska, and Peter Green. Business process modeling-a comparative analysis. *Journal of the association for information systems*, 10(4):1, 2009.

[RRR+21] Riccardo Rubei, Juri Di Rocco, Davide Di Ruscio, Phuong T. Nguyen, and Alfonso Pierantonio. A lightweight approach for the automated classification and clustering of metamodels. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021*, pages 477–482. IEEE, 2021.

[SBF+23] Tiago Prince Sales, Pedro Paulo F Barcelos, Claudenir M Fonseca, Isadora Valle Souza, Elena Romanenko, César Henrique Bernabé, Luiz Olavo Bonino da Silva Santos, Mattia Fumagalli, Joshua Kritz, João Paulo A Almeida, et al. A fair catalog of ontology-driven conceptual models. *Data & Knowledge Engineering*, 147:102210, 2023.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[sha] shadcn. shadcn/ui. https://ui.shadcn.com/. Accessed: 28-05-2025.

[SSS22] Brahmaleen Kaur Sidhu, Kawaljeet Singh, and Neeraj Sharma. A machine learning approach to software model refactoring. *International Journal of Computers and Applications*, 44(2):166–177, 2022.

[Tha13] Bernhard Thalheim. *Entity-relationship modeling: foundations of database technology*. Springer Science & Business Media, 2013.

[Ver] Vercel. Turborepo. https://turborepo.com/. Accessed: 28-05-2025.

[vis] vis.js. vis-network. https://github.com/visjs/vis-network. Accessed: 28-05-2025.

[WDA+16] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016.

[wG] webkid GmbH. React flow. https://reactflow.dev/. Accessed: 28-05-2025.

[WPC+20] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[WSS22] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. Recommending metamodel concepts during modeling activities with pre-trained language models. *Software and Systems Modeling*, 21(3):1071–1089, 2022.