IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 25 August 2025, accepted 18 September 2025, date of publication 29 September 2025, date of current version 6 October 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3615828

RESEARCH ARTICLE

LLM-Driven MDA Pipeline for Generating UML Class Diagrams and Code

ZAKARIA BABAALLA, (Member, IEEE), ABDESLAM JAKIMI, (Member, IEEE), AND MOHAMED OUALLA

Software Engineering and Information Systems Engineering (GL-ISI) Team, Faculty of Sciences and Techniques of Errachidia (FSTE), University of Moulay Ismail (UMI), Meknes 50050, Morocco

Corresponding author: Zakaria Babaalla (za.babaalla@edu.umi.ac.ma)

ABSTRACT The transformation of textual specifications into formal software models is a major challenge in software design automation. This study presents an integrated approach that combines the natural language interpretation capabilities of transformer models with explicit concept structuring using a domain specific language (DSL). This DSL, designed as a pivotal intermediate layer, ensures continuity between semantic extraction, UML modeling, and automatic Python code generation. By following the model-driven architecture (MDA) paradigm, the proposed pipeline follows a structured progression from text to model to code while maintaining a high level of traceability and controllability. The experimental evaluation conducted on a dedicated annotated corpus demonstrates the accuracy of the models on UML entities and highlights the importance of DSL for validation, editing, and exploitation of results. This approach paves the way for the development of intelligent modeling tools and structuring of automated transformation chains.

INDEX TERMS Class diagram, UML, MDA, DSL, LLMs, NLP, IOB annotation, source code.

I. INTRODUCTION

Modeling plays a central role in modern software development processes. It allows the formalization of functional requirements, abstractly representing the structure and behavior of a system, and guides the subsequent phases of design, documentation, and code generation [1]. The Unified Modeling Language (UML) is a widely used modeling standard [2], particularly through class diagrams. Class diagrams play an essential role in representing business entities, their attributes, methods, and the relationships between them [3]. They constitute a basis for communication between stakeholders, support for object-oriented design, and a gateway for implementation [4].

Despite their importance, UML diagrams are still mostly constructed manually from textual specifications [5]. This process relies on analysts'expertise and ability to interpret documents that are often written in natural language. The latter, although accessible and expressive, remains fundamentally ambiguous, contextual, and unstructured [6]. The same requirement can be formulated in multiple ways depending on the writer, making reliable and consistent automated extraction difficult. Consequently, the manual creation of UML diagrams from text is time-consuming, subject to interpretation errors, and difficult to trace.

The automation of this transformation from text to the UML model is now a strategic research axis. It is located at the intersection of natural language processing (NLP) [7], model-driven engineering (MDE) [8], and formal approaches to information structuring. The emergence of deep learning models, notably large language models (LLMs) [9], has made it possible to achieve remarkable performance levels in named entity recognition (NER) tasks [10]. These models, trained on large corpora, capture linguistic and semantic regularities, and are capable of automatically identifying the structuring components of a system from natural language specifications [11].

However, existing approaches based on NLP models often lead to results that are difficult to explain or verify [12], [13], [14], which suffer from a lack of transparency and do not

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet.

always allow validation or editing of the extracted models before the actual generation of diagrams or code. To fill this gap, we propose introducing an explicit and structured intermediate representation, in the form of a domain specific language (DSL) [15], designed to formalize the extracted UML entities. This textual DSL acts as a pivotal interface between the NLP model outputs and the UML artifacts. It allows validation of the syntactic structure, makes manual modifications if necessary, and guarantees clear traceability between the source text, extracted entities, and generated objects.

From this perspective, we present a complete approach aligned with the model-driven architecture (MDA) paradigm [8], which allows the structural transformation from text to code through three levels. The specifications expressed in natural language are first analyzed by an NER model based on a transformer [9]; the identified entities are then structured in a DSL playing the role of the PIM model, and the latter is automatically transformed into UML diagrams, then into executable source code. Therefore, the entire process is modular, explainable, and adaptable.

This study contributes to the literature in several ways. It introduces a manually annotated corpus according to an enriched Inside, Outside, Beginning (IOB) schema [16] that is suitable for extracting UML entities from technical texts. It proposed an original pipeline combining entity recognition [10], DSL structuring [15], UML modeling [2], and Python code generation [17]. Finally, it offers a thorough experimental evaluation, comparing several NER models on real specifications and analyzing the performances obtained on the entire chain, from text to code.

The remainder of this paper is organized as follows. Section II presents the theoretical foundations of our approach, successively addressing textual specifications, UML modeling, the MDA paradigm, NLP techniques, and the role of DSL. Section III presents related works and situates our contribution in the state-of-the-art. Section IV describes the construction of an annotated corpus. Section V describes the processing methodology from text-to-code generation. Section VI presents experimental results. Finally, Sections VII and VIII discuss the contributions, limitations, and open perspectives for evolution towards more automated, traceable, and explainable software modeling.

II. BACKGROUND

This section presents the conceptual foundation for the proposed approach. First, it explores the nature of textual specifications in natural language and their limitations in a software context. It then examines the fundamental principles of UML modeling, particularly class diagrams, and the model-driven architecture (MDA) paradigm, which structures the progressive transformation of a requirement expressed in natural language into a formal software artifact. Recent advances in NLP, notably UML entity recognition using transformer models, are also presented. Finally, this section introduces the central role played by a domain-specific language (DSL), which ensures explicit structuring of the extracted elements, facilitating their validation and transformation in a model-driven chain.

A. NATURAL LANGUAGE TEXT SPECIFICATIONS

In most software projects, functional requirements are described in natural language, in the form of specifications, usage scenarios, or technical documents. These texts are generally written by business experts or analysts, and are intended to be read and interpreted by software engineers. Among the most common formats, we can distinguish Software Requirements Specifications (SRS) [18], which adopt a relatively formal structure, and user stories [19], typical of agile methods, which describe functionality from the point of view of an end user.

Natural language writing has several advantages: it is accessible, flexible, and easily understood by all project stakeholders, regardless of their technical specialty. However, the same flexibility results in significant syntactic and semantic variability [20]. The same requirement can be formulated in multiple ways depending on the style, level of detail, or precision of the writer. This linguistic heterogeneity makes any attempt at direct automatic processing difficult because it introduces ambiguities, redundancies, and implicit meanings [6]. For example, the relationship between two entities can be expressed explicitly (`A student is enrolled in a course`) or implicitly (`Each course includes students`), which requires fine interpretation skills to deduce an association relationship.

These difficulties justify the use of advanced natural language processing (NLP) techniques that can analyze deep discourse structures, identify key entities, and model underlying relationships [21]. Therefore, the first step towards automatic modeling is to treat these texts as sources of information to be structured, and not as ready-to-use descriptions. This fundamental need for formalization is addressed by approaches that integrate NLP into model-driven modeling pipelines.

B. UML CLASS DIAGRAMS

The UML class diagram is one of the most widely used artifacts in object-oriented modeling [2]. It statically represents the internal structure of a system by identifying its main entities (classes), the data they contain (attributes), the operations they perform (methods), and the relationships between them (associations, aggregations, compositions, and generalizations) [22], [23]. Each class generally corresponds to a domain concept, encapsulating both data and behaviors, whereas relationships capture the structural and hierarchical links that define the architecture of the system.

Beyond their descriptive function, UML class diagrams play a central role in the software development cycle [24]. They serve as a shared reference for developers, analysts, and clients, helping to validate requirements, explore design alternatives, and formally document a system's structure.

They are also a foundation for automatic code generation, where classes and their relations are transformed into equivalent constructs in programming languages, such as Java, C#, or Python.

In the context of model-driven architecture (MDA), the class diagram acts as a bridge between high-level, platform-independent models (PIM) and concrete implementations (PSM) [25]. In this study, we aimed to automatically generate these diagrams from textual specifications using a domain-specific language (DSL) serving as an intermediate representation. This structured approach enhances traceability, reduces manual effort, and improves the consistency and reliability of generated software models.

C. MDA PARADIGM

Automating software production from specifications requires rigorous structuring of different transformation steps. The model-driven architecture (MDA) paradigm, proposed by the Object Management Group (OMG) [26], provides a powerful theoretical framework for organizing this progressive transformation through well-defined modeling levels. This subsection presents the three key MDA levels and highlights their benefits in terms of traceability, automatic generation, and separation of concerns.

The MDA paradigm is based on model transformation logic, in which abstract representations of the system are progressively refined towards more concrete forms until reaching the level of executable code [8]. This approach is based on the following three hierarchical modeling levels:

- The Computation Independent Model (CIM) corresponds to the conceptual level: it describes the system by focusing on business needs, without any technical considerations. A CIM is often expressed in natural language or in the form of semi-formal representations, such as process diagrams or user scenarios.
- The Platform Independent Model (PIM) is the logical modeling level. It represents the functional structure of the system independent of any implementation technology. This is where UML models, such as class diagrams, are used to formalize software entities and their attributes, behaviors, and relationships. PIM provides a stable abstraction that can be reused to generate different implementations that target various platforms.
- The Platform Specific Model (PSM) is the technical projection of the PIM onto a given platform or language. It considers the constraints specific to a technology (database, programming language, etc.) and allows the automatic generation of source code from the elements of the logical model.

One of the major interests of MDA lies in its ability to separate business, logical, and technical concerns while ensuring vertical traceability between different levels [27]. This makes it possible to precisely track the origin of a software artifact, return it to the initial specification, and check its consistency. This structuring also facilitates the reuse of models, portability between platforms, and automated code generation, while reducing the risks of human errors.

In our approach, the MDA paradigm guides the entire pipeline: natural language specifications constitute the CIM, extracted entities are formalized in a DSL that acts as the PIM, which is then automatically transformed into UML class diagrams (PSM) and then into Python code. This slicing makes the system more robust, explainable, and better suited to modern software engineering demands.

D. AUTOMATIC EXTRACTION VIA NLP

The analysis of textual specifications to extract usable UML entities relies on natural language processing (NLP) techniques. This subsection examines the role of named entity recognition (NER) [10] in this context and highlights the contribution of transformer-type models to the reliable identification of the structuring elements of the system. Thus, it prepares the transition towards the use of a DSL as a means of explicitly structuring these entities.

Natural language processing encompasses all techniques aimed at enabling machines to understand, analyze, or generate human language [7]. In the context of extracting UML models from textual specifications, the objective is to identify relevant entities in the text, such as classes, attributes, methods, and relationships, and then to structure them according to modeling logic. The task of named entity recognition (NER) is central. It consists of automatically detecting and classifying text segments (tokens or groups of words) by associating them with a predefined category such as `B_CLASS`, `I_ATTRIBUTE`, or `B_RELATION`.

Initially developed to detect generic entities (`people`, `places`, and `organizations`), NER was adapted to recognize domain-specific concepts, such as UML elements. This adaptation requires having an annotated corpus in a suitable format (e.g., IOB), and a model capable of generalizing from this annotation.

Transformer models, which have emerged in recent years, have revolutionized the field of NLP. They rely on multi-head attention mechanisms [9] to capture contextual relationships between words in a sentence, even at long distances. Variants, such as BERT [28], RoBERTa [29], SpanBERT [30], XLNet [31], Electra [32] or MiniLM [33] are currently the most efficient for NER tasks. By fine-tuning them on a corpus annotated specifically for UML entity extraction, these models can detect, with high accuracy, the structural concepts of a system from natural texts, including imperfect or ambiguous ones.

However, the raw extraction is insufficient. The extracted entities must be organized and validated for use in a formal modeling framework. Therefore, introducing an intermediate representation via a DSL completes the process by providing a syntactically controlled space in which NLP results are reinterpreted, structured, and prepared for UML transformation. This is described in detail in the following subsection.

E. STRUCTURING ROLE OF THE DSL

Once UML entities have been extracted from textual specifications, the question arises of their structuring, validation, and transformation into formal software artifacts. This subsection introduces the notion of a domain-specific language (DSL), which plays a pivotal role in our approach. It provides a definition and example of syntax, and describes its central role as a PIM level in the MDA architecture. Here, the DSL is presented as a structuring, controllable, and modifiable mechanism that enhances the traceability and transparency of the pipeline.

A domain-specific language (DSL) is designed to satisfy specific needs in a given application domain [15]. Unlike general-purpose languages such as Java or Python, DSL focuses on a restricted vocabulary that is specific to the problem to be solved. In our case, DSL is used to formalize the extracted UML entities (classes, attributes, methods, and relationships) in a simple, coherent textual syntax, which can be manipulated by automatic tools. Thus, it represents a structured intermediate layer between the natural language and the generated UML diagrams.

The syntax of this DSL is intentionally minimized, to facilitate readability and editing. For example, a specification like `'A user places an order'` can be automatically transformed into a DSL block as follows:

```
Class User:
  Methods: placeOrder
Class Order:

Relation: User association Order
```

Each entity is formalized according to a clear hierarchical structure, with explicit keywords (`Class`, `Methods`, `Relation`) and indentations inspired by modern languages.

In our pipeline, the DSL acts as a PIM (Platform Independent Model) level. This allows the results of linguistic analysis to be formalized independently of any technical platform or programming language. This intermediate representation has several advantages: It facilitates syntactic validation thanks to a defined grammar, allowing inconsistencies or omissions to be automatically detected. It improves traceability by maintaining a clear link between each detected textual entity and its equivalents in the formal model. This allows manual editing: the user can correct, complete, or adjust the model before moving on to UML generation or code, thus providing fine-grained control over the resulting output.

As a pivotal step in the pipeline, DSL provides an anchor for transforming textual specifications into usable software artifacts. This contributes to the separation of concerns valued by the MDA paradigm while enhancing the robustness, explainability, and flexibility of the entire system.

III. RELATED WORKS

The automatic extraction of UML class diagrams from natural language has been the subject of numerous studies based on linguistic approaches and machine learning techniques [34], notably LLMs. While these studies have enabled significant advances, they still present limitations, such as insufficient quality of the corpora, lack of explainability of the models, low adaptability to real contexts, and the absence of user control. This section reviews the main existing approaches, identifies their weaknesses, and positions our contribution as a structured and integrated response to the MDA paradigm.

A. TRADITIONAL LINGUISTIC APPROACHES

More and Phalnikar designed RAPID [35], a tool that automatically generates UML diagrams from textual specifications by combining NLP techniques, domain ontology, and heuristic rules. It identifies classes, attributes, and relationships by leveraging OpenNLP [36] and WordNet [37] for semantic analysis. This approach helps structure software requirements, reduce natural language ambiguity, and facilitate transformation into usable UML models.

Shinde et al. [38] proposed a method using natural language processing (NLP) to automatically extract object-oriented elements (classes, attributes, methods, and relationships) from textual specifications. The system applies grammatical analysis and heuristic rules to construct UML diagrams (classes and sequences) and generates Java code. By combining POS tagging [39], WordNet, and parsing models, it allows moving from textual description to partially automated software modeling, thus facilitating the design phase in software development.

Bashir et al. implemented READ [40], an automatic tool for generating UML class diagrams from English textual specifications, combining NLP techniques (tokenization and POS tagging) and heuristic rules. It extracts classes, attributes, methods, and relationships based on a domain ontology with a threshold system to filter relevant entities. Evaluated on several use cases, READ achieved a recall of 94% and a precision of 69.5%, outperforming several existing tools. This offers an effective solution for automating object modeling from text.

Sanyal and Ghoshal [41] proposed a hybrid method for automatically generating UML class diagrams from textual specifications. It combines a system based on linguistic rules to extract classes, attributes, and relationships with a fusion mechanism to unify the elements extracted from different use cases. This approach allows us to obtain consistent, accurate, and redundancy-free diagrams while remaining more flexible than purely machine learning-based methods.

Aunsa et al. [42] explored a heuristic method to extract classes and semantic relationships from user scenarios. This approach aims to facilitate UML modeling by allowing users to define conceptual structures without requiring extensive technical expertise. This study highlights the effectiveness of this method in end-user development contexts, where understanding business needs takes precedence over programming skills.

B. MACHINE LEARNING AND LLM-BASED APPROACHES

Yang and Sahraoui [43] developed a method for automatically extracting UML class diagrams from natural language specifications. Their approach was based on a structured processing chain: text preprocessing, identification of classes and relationships, generation of UML fragments according to rules, and their assembly. They constructed an annotated dataset from the AtlanMod Zoo[1] repository. The results show that, despite its effectiveness, this method remains limited by linguistic ambiguities and performance of NLP tools. However, this study constitutes an important contribution to the automation of UML modeling.

Saini et al. presented DoMoBOT [44], which is an intelligent system capable of automatically extracting UML class diagrams from textual descriptions. By combining rule-based NLP techniques and supervised learning models, the system improves the extraction accuracy. It integrates human-machine interaction by proposing suggestions to the modeler and adjusting the model according to feedback. DoMoBOT also ensures traceability through dedicated structures. The evaluations showed high F1-scores and real-time operation, validated by a real-world user study.

Rigou et al. [45] introduced an innovative method for the automatic extraction of UML class diagrams from textual specifications, relying on deep learning techniques including BiLSTM networks [10], integrating Named Entity Recognition (NER), coreference resolution [10], and relationship classification [10]. This approach aims to overcome the limitations of traditional heuristic methods by efficiently dealing with linguistic ambiguities and implicit structures often present in specification texts. Through experiments, The approach demonstrates that this combination improves the accuracy of UML entity identification (classes, attributes, and relationships), thus enhancing the automation potential in the software-modeling chain.

The approach proposed by Calamo et al. [11] evaluates the extent to which large language models (LLMs), such as ChatGPT [46], are capable of automatically generating UML class diagrams from textual descriptions. This method examines the syntactic validity of the diagrams produced, their semantic fidelity to the expressed requirements, and their usefulness as conceptual models for model-driven engineering. The results show that, although LLMs generally produce compliant and well-structured diagrams, inaccuracies persist, particularly in the relationships between classes and completeness of the models. The study concludes that these tools offer good assistance potential but still require human validation to ensure the quality of the generated models.

Rouabhia and Hadjadj [47] studied in their approach how large language models can automatically enrich UML class diagrams by generating methods using use cases. By testing nine LLMs on a set of 21 use cases, the study shows that all produce syntactically valid diagrams, with varying performances depending on the models. The results indicate that LLMs can accelerate modeling; however, errors in annotations and signatures indicate that human supervision is still necessary.

Al Ahmad et al. [48] analyzed the effectiveness of the GPT-4 Turbo [49] in generating different types of UML diagrams. Comparing the AI results with those of human students, it can be seen that the diagrams produced by AI are generally less complete and less accurate, especially for class and use case diagrams. Deployment and sequence diagrams were closer to the human versions. The study concludes that GPT-4 Turbo can be a useful aid, but still requires human validation.

Campanello et al. [50] examined the effectiveness of GPT4 [51] for reverse engineering UML class diagrams from source code. Comparing the generated diagrams with those produced by experts, the authors found that GPT4 achieved very good results for identifying attributes and methods (F1 close to 90%). However, its performance is lower in detecting relationships between classes, including associations and generalizations. The study also tested the addition of indices in the prompts, but this did not provide any notable improvement. The approach concludes that GPT4 has good fine-grained abstraction capability, but remains limited for global conceptual decisions, which justifies the need for human control in UML modeling tasks.

C. LIMITATIONS OF EXISTING APPROACHES

Despite the wealth of work devoted to the automatic extraction of UML class diagrams from textual specifications, several recurring limitations compromise their effectiveness and adoption in real contexts.

First, the quality of the training data remains a central issue. Most existing corpora have incomplete, automatically generated, or overly generic annotations. The lack of granularity in tags, confusion between relationship types (association, aggregation, etc.), and the approximate processing of complex entities (multi-word or ambiguous) directly affect the reliability of extractions.

Second, the most advanced machine learning models, particularly those based on deep learning or LLMs, lack explainability. Their decisions often come from opaque layers, making it difficult for the user to understand, validate, or correct results. This lack of transparency is generally accompanied by a lack of editorial control, which limits their integration into interactive or collaborative tools.

Third, heavy reliance on annotated data reduces the portability of these models to other application domains or languages. Few approaches guarantee explicit traceability between source texts and produce UML artifacts, making verifications or adjustments complex, especially in industrial contexts that require rigor, compliance, and maintenance.

Finally, most contributions were poorly adapted to real-world use cases. The texts used in experiments are often simplified or artificial and are far removed from the semantic and structural complexity of documents from professional software projects. Furthermore, integration into model-driven

[1] https://github.com/atlanmod/zoo

architecture (MDA) chains remains rare, limiting their use in model-oriented tool environments.

D. POSITIONING OF OUR CONTRIBUTION

Our approach is a direct response to the limitations identified above by proposing a modular explainable architecture that complies with the MDA paradigm. It introduces an explicit textual DSL that acts as an intermediate layer between the automatic extraction of UML entities (via LLMs) and the generation of UML class diagrams and source code. This DSL, positioned at the Platform-Independent Model (PIM) level, allows validation, editing, and traceability of the extracted elements, thus providing structured editorial control, unlike end-to-end approaches that are often opaque and difficult to adjust.

Table 1 summarizes the positioning of our contribution with respect to previous approaches. Compared to rule-based methods, machine learning (ML), and LLMs, our pipeline provides higher transparency, validation capacity, and compatibility with MDA principles, while maintaining the possibility of human intervention through DSL editing.

To address the paucity of annotated datasets, we built a dedicated and manually annotated corpus, according to an enriched IOB schema, covering a wide spectrum of UML entities (classes, attributes, methods, and relationships) and integrating the fine distinction between relationship types. This corpus allows the training of NER models that are more robust and better adapted to real specifications.

Finally, our contribution materializes in the form of a complete operational pipeline combining extraction, validation via DSL, UML transformation, and code generation (Python). This pipeline guarantees a reliable, explainable, and interactive modeling chain that is better aligned with the requirements of industrial environments and compatible with advanced software engineering processes.

IV. CREATING THE ANNOTATED DATASET FOR UML EXTRACTION

This section presents the process of creating an annotated dataset to train the UML class diagram extraction models. It details the origin and selection of texts, design of an enriched IOB annotation schema adapted to UML entities, and an illustrative example. The objective was to ensure sufficient semantic and syntactic coverage to enable effective training while highlighting the challenges encountered.

A. ORIGIN OF THE TEXTS

Building a reliable dataset for automatic UML element extraction requires sufficient linguistic and semantic diversity to ensure the robustness of the supervised learning models. With this in mind, we selected textual specifications from various sources representative of current practices in software documentation.

Open-source repositories on GitHub[2] are an important first source. Many projects present functional descriptions in README files, specification documents, or user stories, which are often structured according to professional conventions. These texts enable the extraction of realistic cases written by developers or business analysts in various contexts.

We also leveraged content from technical forums and community platforms such as Stack Overflow.[3] Although less formal, these exchanges contain descriptions of expected behaviors or data structures that are rich in reusable concepts in the modeling logic.

Finally, more traditional software documentation was integrated, including user manuals, Software Requirements Specifications (SRS), and object-oriented design tutorials. These texts provide a more normative, often well-structured language that is essential for balancing the corpus between technical language and more free-form expressions.

This diversity of sources ensures a good representation of the types of formulations encountered in software projects, from highly structured descriptions to free specifications, thus promoting the generalization of the model to heterogeneous texts.

B. SELECTION CRITERIA

The selection of the texts to be annotated was not performed randomly. It followed rigorous criteria, focusing on the quality of content from a UML modeling perspective. Two major axes guided this process: the semantic richness of specifications and their syntactic clarity.

First, UML semantic richness refers to the explicit or implicit presence of structuring elements of the UML language in a text. This includes the mention of classes, attributes, behaviors (methods), and relationships, such as association, composition, and inheritance. The chosen texts had to offer sufficient density to the UML entities to justify their annotation and maximize model learning. Specifications that were too poor or generic were discarded.

Second, syntactic clarity plays a crucial role. The selected texts had to have a relatively stable grammatical structure, with well-formed sentences to facilitate their processing by tokenization and automatic annotation models. This does not mean excluding complex or technical texts; rather, it seeks a balance between syntactic variety and interpretability.

Finally, an effort was made to maintain a balance between short and long specifications, allowing both unitary descriptions (of a class or function) and more global specifications describing complete scenarios. This mix aims to strengthen the model's ability to generalize across different levels of conceptual granularity.

C. ENRICHED IOB ANNOTATION SCHEME

To accurately extract UML components from textual specifications, this study adopts an annotation scheme based on the Inside, Outside, Beginning (IOB) format, enriched to meet the specificities of the software-modeling domain. The IOB format, widely used in Named Entity Recognition (NER),

[2]www.github.com

[3]www. Overflow.com

TABLE 1. Comparative positioning of existing approaches and our proposed pipeline.

Approach	Transparency	Validation / Correction	Explainability	MDA integration	User editing support
Rule-based	High	Limited (rule-dependent)	Medium (local explanations)	Low	Low
Machine Learning	Medium	Medium (post-processing)	Low	Low to medium	Low
LLMs	Low	Low (opaque module)	Low to medium	Low to medium	None
Proposed pipeline	High	High (integrated mechanisms)	High (explicit DSL)	High	Supported

allows sequential encoding of the structure of multi-word entities by identifying the beginning (B), continuation (I), or absence (O) of a given entity [16].

The proposed schema was extended to include specialized semantic tags corresponding to the targeted UML elements:

- `B_CLASS_SOURCE/I_CLASS_SOURCE`: identify the start and continuation of a source class involved in the UML relationship.
- `B_CLASS_TARGET/I_CLASS_TARGET`: identify target classes in relationships.
- `B_ATTRIBUTE/I_ATTRIBUTE`: mark the attributes defined in the classes.
- `B_METHOD/I_METHOD`: indicate the associated methods or behaviors.
- `B_ASSOCIATION`, `B_COMPOSITION`, `B_AGGREGATION`, `B_INHERITANCE`: tags specific to UML relationship types, each expressing a distinct structural or hierarchical link.

This level of granularity allows the learning model to recognize not only basic entities but also their structural role and contextual position within the software architecture described in the text.

The choice of IOB format is justified by its compatibility with modern sequential models that learn to predict sequences of labels from tokenized text [52]. This format guarantees a linear, flexible annotation that is well-suited to the recognition of disjoint entities or entities embedded in natural formulations. In addition, it facilitates the reconstruction of complex entities in post-processing, which is an essential criterion for generating usable UML diagrams from model predictions.

By adopting this enriched schema, the dataset not only identifies UML components but also encodes the implicit semantic relationships between these elements, thus laying the foundation for the structured generation of UML diagrams from text.

D. EXAMPLE OF UML ANNOTATION ON A TEXT EXTRACT

To illustrate the structure of the constructed dataset, consider the following example taken from a software specification:

`'Each user can register and modify their credentials . A profile is linked to one user only.'`

In this example, UML entities are expressed implicitly without recourse to the explicit technical formulations generally expected in formal specifications (Table 2).

TABLE 2. Example of UML annotation on a text extract.

Token	Label	Explanation
Each	O	Outside UML entity
user	B_CLASS_SOURCE	Main source class
can	O	Outside UML entity
register	B_METHOD	Implicit method associated with the user class
and	O	Outside UML entity
modify	B_METHOD	Another implicit method owned by user
their	O	Outside UML entity
credentials	B_ATTRIBUTE	Implicit attribute of the user class
.	O	Outside UML entity
A	O	Outside UML entity
profile	B_CLASS_TARGET	Target class associated with user
is	O	Outside UML entity
linked	B_ASSOCIATION	Implicit association relationship between user and profile
to	O	Outside UML entity
one	O	Outside UML entity
user	B_CLASS_SOURCE	Repeated reference to the user class
only	O	Outside UML entity
.	O	Outside UML entity

The word `user` is here interpreted as a source class because of its role as the main actor in the actions described (`register` and `modify`), which are recognized as methods. The term `credentials`, although introduced in a possessive manner (`their credentials`), is identified as an attribute belonging to the class `user`. Moreover, `profile` constitutes a target class, involved in an association relationship with `user`, deduced from the verbal expression `is linked to`.

This annotation highlights the ability of the enriched IOB schema to model entities and relationships from indirect functional formulations, which are common features of natural specifications. It also highlights the importance of contextual NER models capable of disambiguating entities from the context of use, which reinforces the interest of using transformer models in the UML extraction process.

E. QUANTITATIVE ANALYSIS OF THE CORPUS

The annotated corpus implemented in this study consists of 132 textual specifications, from various sources (open source repositories, technical forums, functional documents, etc.), ensuring appreciable heterogeneity in terms of style, syntactic complexity, and domains covered. These texts were segmented into 915 distinct sentences, totaling 11,460 tokens that were manually annotated according to the enriched IOB

TABLE 3. Distribution of entities.

Entity	Number of occurrences	Percentage (%)
B_CLASS_SOURCE	1,920	18.0%
I_CLASS_SOURCE	765	7.2%
B_CLASS_TARGET	1,635	15.4%
I_CLASS_TARGET	640	6.0%
B_ATTRIBUTE	1,510	14.2%
I_ATTRIBUTE	580	5.4%
B_METHOD	1,350	12.7%
I_METHOD	520	4.9%
B_ASSOCIATION	660	6.2%
B_COMPOSITION	510	4.8%
B_AGGREGATION	300	2.8%
B_INHERITANCE	260	2.4%

schema. Regarding UML entities, the corpus includes a total of 10,650 occurrences, as shown in Table 3.

This distribution highlights a high density of annotations on classes and their attributes, which form the core of UML class diagrams, whereas specific relationships (such as inheritance or composition) are less frequent but crucial for the structure of the models.

The linguistic diversity of the corpus was carefully studied to ensure generalization of the trained models. Short sentences, explanatory blocks, enumerations, conditional structures, and narrative-style texts were integrated to reflect the different ways specifications could be formulated. This stylistic and semantic richness constitutes an essential lever for training robust models capable of adapting to multiple contexts in software engineering.

F. DIFFICULTIES ENCOUNTERED

The creation of the annotated corpus revealed several major challenges, linked to the nature of natural language and implicit structure of software specifications. Three types of difficulties particularly affected the annotation process and directly influenced the performance of the trained models:

- **Lexical ambiguities**: Some terms exhibit semantic versatility, which makes their annotation contextual. For example, the word '`Type`' can denote an attribute (`Type of account`), a verb (`users type their names`) or a class (`AccountType`). This ambiguity forces the annotator to interpret the syntactic role and exact meaning of the word in context, which can introduce subjectivity into labeling. Other technical terms like '`item`', '`order`', or '`message`' are also frequently ambiguous depending on their grammatical position or the application domain.
- **Implicit entities**: Many UML entities, especially relationships, are not explicitly expressed in the text, but must be inferred from the overall structure or meaning. For example, a sentence like '`Each order contains multiple items.`' implies a compositional relationship between classes `Order` and `Item`, without using a technical keyword like '`composition`'. This phenomenon makes identifying relationships more complex than identifying atomic entities (classes, attributes, and methods) because it requires more in-depth semantic interpretation.
- **Coreferences and contextual dependencies**: Certain entities are introduced in one statement and referenced in subsequent statements in the form of pronouns or elliptical noun phrases. For example: '`The class Car has several attributes. It also includes the getSpeed method.`':
  Here, '`it`' refers to `Car`, but without coreference resolution, an automatic model might fail to establish the link. This type of inter-sentential dependency is common in long or paragraph-structured specifications, and requires discourse processing beyond a single sentence.

To mitigate these difficulties and improve the efficiency of automatic extraction, we classified natural language specifications according to their level of refinement, considering their structure, syntactic clarity, and lack of ambiguity. This categorization allowed us to better adapt the application of NER techniques to the different text qualities encountered. The distribution of our corpus, according to this classification, is shown in Table 4.

V. PROPOSED METHODOLOGY

This section describes the technical approach used to automatically transform textual specifications into usable software artifacts. It is organized around a structured pipeline based on the principles of model-driven architecture (MDA), in which a domain-specific language (DSL) plays a central role as an intermediate representation. The process includes linguistic analysis, semantic structuring, UML modeling, and code generation, following a progression that ensures traceability, modularity, and automation.

The pipeline we propose is based on a hybrid NLP, DSL, and MDA architecture, in which each step transforms information at a different level of abstraction, consistent with the principles of model-driven architecture (MDA). The process takes place in four main steps: from raw textual specification (CIM level) to a structured intermediate language (DSL/PIM), then to UML modeling (PSM level), and finally to code generation (Figure 1).

1) **Text specifications (CIM)**: The starting point of the pipeline is text written in natural language that expresses functional requirements, use cases, or business rules. This level is independent of technical considerations. This corresponds to the Computation Independent Model (CIM) in the MDA paradigm.
2) **Semantic extraction via NLP**: A Transformer-type model previously trained on our annotated corpus performs UML entity recognition (NER). It identifies the classes, attributes, methods, and relationships in the text.
3) **Intermediate structuring (DSL/PIM)**: The extracted entities are organized in a textual DSL, designed to formally represent UML components. This pivotal

TABLE 4. Distribution of the corpus by specification refinement level.

Refinement level	Summary description	Specifications	Occurrences
Controlled and structured language	Standardized syntax, restricted vocabulary, absence of ambiguity	40	3,200
Semi-structured specifications	Natural language organized with patterns, lists, or tabular formats	50	4,050
Unstructured specifications	Free text, heterogeneous style, ambiguities, and domain-specific jargon	35	2,800
Informal or conversational descriptions	Relaxed narrative style, pronouns, incomplete sentences	7	600
Total	—	**132**	**10,650**

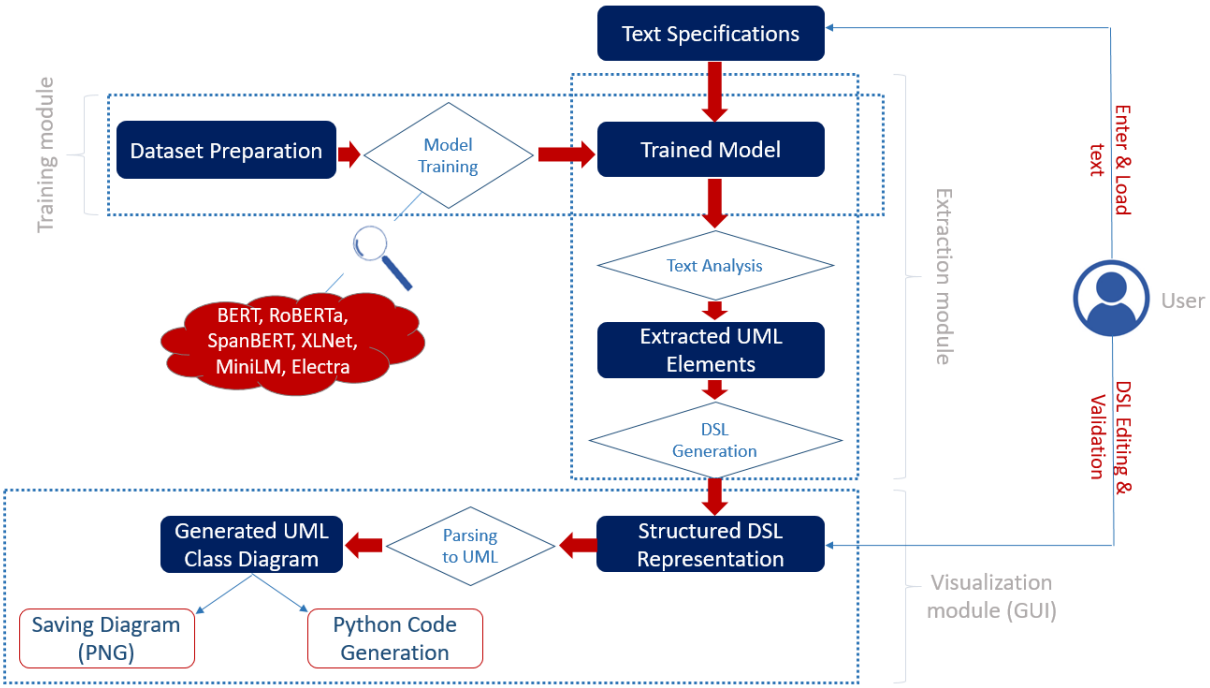

FIGURE 1. Pipeline overview.

language constitutes a platform-independent model (PIM) representation, which is readable, valid, and modifiable. It plays a crucial role in ensuring a clear separation between linguistic extraction and formal modeling.

4) **UML Transformation (PSM)**: From the DSL, a transformation engine automatically generates a UML class diagram, conforming to the syntax of the modeling language. This model constitutes a Platform-Specific Model (PSM) when it is enriched with technical details and targeted at a given execution platform.
5) **Code generation**: In the final phase, the UML model is converted into Python source code based on custom transformation rules. This code constitutes an executable artifact that can be directly integrated into a software project.

The entire pipeline was implemented in an experimental environment featuring a graphical interface for text entry, displaying the generated DSL, visualizing the UML diagram, and exporting the source code. This modular architecture allows user interaction at each step, including the manual editing of the DSL, and promotes full traceability from the original text to the produced code.

The technical artifacts of this work (scripts, configurations, and data samples) are made available in the public GitHub repository,[4] in order to foster reproducibility and scientific reuse.

A. EXTRACTING UML ENTITIES

The first operational step of the pipeline is to automatically analyze the input text to extract the relevant UML components: classes, attributes, methods, and relationship types. This task is performed using a Named Entity Recognition (NER) model trained specifically to detect software entities in natural language specifications. This subsection details the models used and the format of the data processed in the input and output of the NER module.

1) FINE-TUNED MODELS

For this task, we selected a series of transformer-like models, pre-trained on large general-purpose text corpora,

[4]https://github.com/zakariababaalla/uml-text2code-pipeline

and then fine-tuned them on our annotated corpus described in the previous section. The evaluated models included BERT, RoBERTa, XLNet, SpanBERT, MiniLM, and Electra. Each model was trained on the dataset with a sequential classification objective, assigning each token a label from the enriched IOB schema.

Fine-tuning was performed with the HuggingFace Transformers library [53] using the AdamW optimizer [54], a learning rate of 2e-5, a batch size of 16, and an epoch-based evaluation strategy. Performance was measured using standard metrics (precision, recall, and F1-score), considering only significant tokens (excluding those labeled `O` or subtokens ignored via the value -100).

The experimental results presented in Section VI show that these models achieve very high F1-scores, particularly for class and relation type entities, demonstrating their ability to capture syntactic and semantic regularities of software specifications.

2) INPUT AND OUTPUT DATA

As input, the module receives tokenized sentences from natural-language text specifications. Each sentence was preprocessed to be compatible with the model constraints (tokenization, padding, and alignment with the sub-tokens generated by the tokenizer).

As an output, the system produces an IOB-type label associated with a UML category for each token. For example, from the sentence: '`Each customer places an order`', the module can produce the following sequence:

```
Each     → O
customer → B_CLASS_SOURCE
places   → B_METHOD
an       → O
order    → B_CLASS_TARGET
```

These outputs are then grouped by entity and normalized to feed the next step, DSL generation. This structured output format is particularly suited for automatic transformation into a formal syntax while allowing manual verification or post-correction before model generation.

B. STRUCTURING VIA THE INTERMEDIATE DSL

Once the UML entities are extracted by the recognition module, they are transformed into an explicit and structured textual representation using a DSL (Domain-Specific Language) designed specifically to model UML components in a clear, formalized, and automatically manipulable manner. This step plays a central role in our pipeline and constitutes at the same time a syntactic validation layer, a manual editing space, and a pivot point for the generation of UML artifacts and source code.

1) DSL SYNTAX

The DSL adopted in our approach follows a textual syntax inspired by modern configuration languages, adapted to the requirements of object-oriented UML modeling. It allows

```
start: bloc_class+ relation*

bloc_class: "Class" NAME ":" bloc_body

bloc_body: (attributes | methods)+

attributes: "Attributes:" attribute_list
methods: "Methods:" method_list

attribute_list: NAME ("," NAME)*
method_list: NAME ("," NAME)*

relation: "Relation:" NAME relation_type NAME

relation_type: "association" | "aggregation" | "composition" | "inheritance"

NAME: /[A-Z][a-zA-Z0-9_]*/
```

FIGURE 2. Grammar of the intermediate DSL used for UML generation.

classes, their attributes, their methods, and the relationships that unite them to other classes to be described in a structured manner. Its syntax is based on a simple hierarchy of indentation, explicit keywords (`Class`, `Attributes`, `Methods`, `Relation`), and minimal use of separators. This structure is formalized by declarative grammar (Figure 2), guaranteeing both human readability and automatic generability in standardized formats.

An example of a DSL block automatically generated from a natural language sentence is as follows:

```
Class Customer:
  Attributes: name, email
  Methods: placeOrder
Class Order:
  Attributes: id, date

Relation: Customer association Order
```

Each class was defined by its name, followed by its attributes and methods. Relationships are described separately as "source relation target" triples, allowing associations, compositions, aggregations, or inheritance to be captured.

2) PARSING AND VERIFICATION

Once generated, the DSL is subjected to syntactic parsing and validation phases. Unlike an approach using formal grammar such as Lark [55], our system relies on a custom-developed manual parser (Figure 3). This parser interprets the structure of the DSL line-by-line and identifies the definitions of classes, attributes, methods, and relationships. It also performs elementary checks, such as consistency of relationships or detection of redundant classes, thus ensuring that the structure of the DSL respects the constraints defined by our approach.

To handle ambiguous, incomplete, or noisy inputs from LLMs, the pipeline integrates a multi-level quality control mechanism. On the one hand, the parser applies automatic correction rules (e.g., implicit addition of missing braces, completion of omitted attributes, and standardization of keywords). By contrast, inconsistencies or anomalies that cannot be resolved deterministically are flagged and logged

```
Input: dsl_file (list of lines)
Output: uml_model, validation_report
Begin
    classes ← []
    relations ← []
    anomalies ← []
    current_class ← None

    For line in dsl_file:
        line ← preprocess(line)
        # Normalization (trimming, keyword standardization, noise removal)
        If line is incomplete or ambiguous:
            line ← auto_correct(line)
            If still ambiguous:
                anomalies.append("Ambiguity detected in: " + line)
                Continue
        If line starts with "Class":
            name ← extract after "Class"
            If name is missing:
                anomalies.append("Missing class name in line: " + line)
                name ← "Class_Undefined"
            current_class ← new Class(name)
            classes.append(current_class)
        Else if line starts with "Attributes":
            attrs ← split after ":" by ","
            If attrs is empty:
                anomalies.append("No attributes defined in line: " + line)
            current_class.attributes.add_all(attrs)
        Else if line starts with "Methods":
            methods ← split after ":" by ","
            If methods is empty:
                anomalies.append("No methods defined in line: " + line)
            current_class.methods.add_all(methods)
        Else if line starts with "Relation":
            parts ← extract source, type, target
            If parts incomplete:
                anomalies.append("Incomplete relation in line: " + line)
            Else:
                relations.append(new Relation(parts))
    uml_model ← UMLModel(classes, relations)
    # Logging
    validation_report ← generate_report(anomalies)
    # Optional display in graphical interface
    If user_intervention_required(validation_report):
        uml_model ← launch_GUI_correction(uml_model, validation_report)
    Return uml_model, validation_report
End
```

FIGURE 3. Algorithm for manual DSL parsing and UML model generation.

in a validation report (which can be saved as a `.txt` file). This report is used for both manual review and progressive improvement of the prompts provided to the LLM. Finally, owing to the correction graphical interface, the user can intervene to adjust classes, add or remove attributes, or correct relationships before the final generation.

Thus, the combination of automatic corrections, traceable reporting, and manual validation guarantees increased robustness against uncertainties inherent in natural language and the potential errors produced by LLMs.

3) INPUT/OUTPUT EXAMPLE

For the sentence: '`The administrator manages user accounts and reviews reports.`' the NER model produces the following entities.

`administrator` → `B_CLASS_SOURCE`
`manages` → `B_METHOD`
`user accounts` → `B_CLASS_TARGET`
`reviews` → `B_METHOD`
`reports` → `B_CLASS_TARGET`

These entities are then automatically converted into DSL:

```
Class Administrator:
  Methods: manageAccounts, reviewReports
Class UserAccount:
Class Report:

Relation: Administrator association UserAccount
Relation: Administrator association Report
```

This DSL then becomes the structured starting point from which the UML class diagram is generated and then the source code. It also serves as a support for validation, documentation, and manual enrichment in user-centered logic.

C. UML GENERATION FROM DSL

Once the entities are structured in the DSL, the system automatically generates a UML class diagram, translating this intermediate textual representation into a formal graphical visualization. This phase allows the transition from a syntactic to a visual structure, while respecting the standard conventions of the UML language.

1) DSL TO UML DIAGRAM TRANSFORMATION

The transformation engine reads the DSL content, analyzes and validates it in the previous step, and then constructs a UML graph from the specified classes, attributes, methods, and relationships. Each block `Class` of the DSL is mapped to a UML box containing its members, and each line `Relation` is interpreted as a directed link between classes annotated according to the relationship (association, composition, aggregation, and inheritance). This engine applies transformation rules that comply with the UML standards. For example, an instruction like: '`Relation: Order composition OrderItem`' will be visually translated by a link of type '`composition`' with a solid diamond at the class level `Order`, pointing towards `OrderItem`, respecting the visual conventions of object-oriented modeling.

2) VISUALIZATION WITH INTEGRATED GRAPHICAL INTERFACE

The UML visualization was performed within a graphical interface developed by Tkinter [56] and integrated into the pipeline execution environment (Figure 4). This interface dynamically displays the classes, relationships, and structures generated from the DSL in the form of interactive diagrams.

Each class is represented by a box containing its name, attributes, and methods and is displayed as structured text. Relationships are rendered by lines and arrows connecting the boxes, with custom styles (solid lines, arrows, and diamonds) indicating the UML relationship type.

The user can interact with the interface to:

- Visualize the results of NER extraction and DSL structuring, along with the validation report.
- Manually edit the DSL in parallel.
- Dynamically regenerate the diagram after modification.
- Export the UML class diagram to image format. (`.PNG`)
- Save the generated source code to a python file. (`.py`)

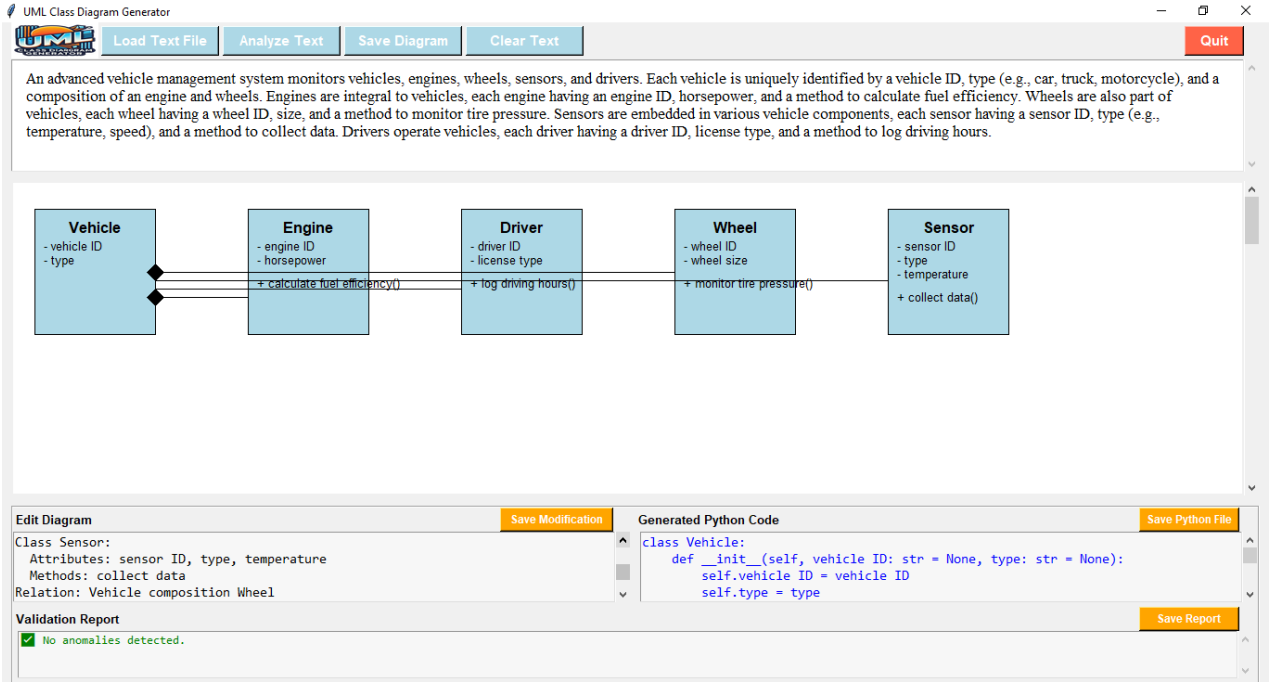

FIGURE 4. Graphical interface of our system.

TABLE 5. Transformation of a class defined in DSL into a python class.

DSL	Generated Python code

`Class Customer:`
`  Attributes: name, email`
 	
`  Methods: placeOrder`
	`class Customer:`
	`def __init__(self, name=`
	`  None, email=None):`
	`  self.name = name`
	`  self.email = email`
	`def placeOrder(self):`
	`  pass`

TABLE 6. Transformation of a relation from DSL into python code.

DSL	Generated Python code

`Relation: Customer`
`association Order`
	`class Customer:`
	`def __init__(self,`
	`orders=None):`
	`self.orders = orders if`
	`orders is not None else []`

- Store the validation report in a text file. (`.txt`)

This embedded visualization closes the loop of the NLP → DSL → UML cycle, providing immediate and controllable rendering of the UML model generated from simple text. It also facilitates the inspection and validation of the model before moving on to the code generation phase.

D. AUTOMATIC CODE GENERATION

The final step of our pipeline is to automatically produce Python source code from the previously generated UML model. This transformation corresponds to the implementation phase following the platform-specific model (PSM) level in the MDA architecture, which involves adapting the abstract representation of the system (PIM, carried by the intermediate DSL), embodied in the form of a UML diagram (PSM), to a target implementation platform, Python. This step not only validates the usefulness of the pipeline until the end of the development cycle, but also allows for the generation of functional prototypes from simple textual specifications.

TABLE 7. Transformation of an inheritance relation from DSL into python code.

DSL	Generated Python code

`Relation: Admin`
`inheritance User`
	`class Admin(User):`
	`def __init__(self):`
	`super().__init__()`

1) UML TO PYTHON CODE MAPPING

The code generation process relies on a mapping mechanism between UML elements (from the DSL or diagram) and their equivalents in Python. Each UML class is transformed into a Python class with a constructor (`__init__`) as well as attributes and methods.

For example, a class `Customer` defined is automatically transformed as follows (Table 5):

Attribute names become instance variables, and methods are generated as empty function definitions (`pass`) that are ready to be completed by the developer.

The UML relationships detected in the DSL were also interpreted to generate logical links between classes (Table 6).

This association relationship is translated into code as an object-type attribute in the source class `Customer`.

2) TRANSFORMATION RULES AND SPECIAL CASES

Generation relies on a set of hand-coded transformation rules implemented as conversion functions that traverse the UML structure (generated from the DSL). The following rules are considered:

- Entity type (class, attribute, method).
- Types of relationships (association, composition, aggregation, inheritance).
- Presence or absence of optional elements (custom builders).

Table 7 lists how an inheritance relationship can be translated into Python.

All generated files can be exported to a project directory containing one `.py` file per class, or grouped into a single module. The resulting code follows good development practices and can be optionally documented, making it a functional starting point for software prototypes. This automatic code generation completes the pipeline and allows for an end-to-end transition from a simple specification text to a set of usable Python classes, while ensuring consistency with the generated UML model and the initially extracted entities.

VI. EXPERIMENTAL EVALUATION

The effectiveness of our pipeline relies heavily on the quality of the UML entity extraction from textual specifications. This section presents the experimental evaluation conducted to measure the system's performance, starting with a description of the experimental protocol and the models tested, before detailing the results per entity and the overall robustness of the process.

A. EXPERIMENTAL PROTOCOL

The objective of this experimental phase was to evaluate the ability of the models to correctly extract UML entities from the textual specifications. To do so, several NER models based on transformer architectures have been compared using a rigorous protocol that relies on a manually annotated corpus. This subsection describes the data used, evaluation metrics applied, and models selected for comparison.

1) TEST DATA AND EXPERIMENTAL SETTING

The annotated corpus was divided into two subsets: a training set (80%) and a test set (20%). The latter included approximately 185 sentences, taken from a total of 915 sentences, and covered all annotated UML categories: source and target classes, attributes, methods, and relationship types (association, composition, aggregation, and inheritance). This distribution ensures a balanced and representative evaluation of model performance, regardless of the level of syntactic complexity of the analyzed textual specifications. Although the dataset was still modest in size (11,460 tokens), the diversity of sources and richness of annotations provided a relevant basis for evaluating model robustness in varied and realistic contexts.

2) EVALUATION METRICS

Performance is measured using three classic metrics in named entity recognition: precision, recall, and F1-score, calculated at the scale of each type of entity, as well as globally [57]:

- **Precision**: proportion of predicted entities that are actually correct.

$$Precision = \frac{VP}{VP + FP} \tag{1}$$

- **Recall**: proportion of expected entities that were correctly detected.

$$Recall = \frac{VP}{VP + FN} \tag{2}$$

- **F1-score**: harmonic mean between precision and recall.

$$F1\text{-}score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3}$$

Here, VP denotes true positives, FP denotes false positives, and FN denotes false negatives. An entity is considered correct only if its span and label exactly match those of the reference annotation, according to a strict evaluation based on exact matching with the expected annotations [58].

3) MODELS COMPARED

Six transformer-type models pre-trained on general linguistic corpora were fine-tuned on our annotated UML corpus:

- **BERT** (base-cased): a 12-layer bidirectional model, pre-trained on Wikipedia and BookCorpus.
- **RoBERTa**: Improving BERT with more training data and optimized pre-training tasks.
- **XLNet**: autoregressive model based on permutations, suitable for long sequences.
- **SpanBERT**: Specialized in predicting continuous segments, well-suited to structure extraction.
- **MiniLM**: a lightweight version of BERT for low-resource environments.
- **Electra**: Discriminative model based on a token replacement task.

Each model was trained for 25 epochs, with a learning rate of 2e-5, batch size of 16, per-epoch evaluation strategy, and limit of 512 tokens per sequence. Training and evaluation were performed with the HuggingFace Transformers library and the Trainer module using the AdamW optimizer. The performance is reported in the next section with a detailed comparison table for each UML entity.

B. UML EXTRACTION RESULTS

The performances obtained by the six evaluated models are presented in terms of precision, recall, and F1-score (Table 8), calculated for each type of targeted UML entity: classes, attributes, methods, and the different relationships

TABLE 8. Performance of language models for UML extraction.

Model	Metrics	Class	Attribute	Method	Association	Aggregation	Composition	Inheritance
BERT	Precision	0.9829	0.9822	0.9881	0.9601	0.9800	0.9800	0.9863
	Recall	0.9883	0.9780	0.9848	0.9667	0.9435	0.9639	0.9500
	F1-score	0.9855	0.9800	0.9864	0.9552	0.9614	0.9718	0.9678
RoBERTa	Precision	0.9861	0.9500	0.9750	0.8779	0.9407	0.9500	0.9429
	Recall	0.9774	0.9500	0.9540	0.9368	0.8496	0.9333	0.9036
	F1-score	0.9817	0.9500	0.9645	0.9001	0.8765	0.9409	0.9201
SpanBERT	Precision	0.7591	0.5066	0.7491	0.5797	0.6000	0.5500	0.6000
	Recall	0.7999	0.5663	0.7447	0.7093	0.4351	0.5097	0.5552
	F1-score	0.7748	0.5303	0.7472	0.6283	0.4817	0.5281	0.5642
XLNet	Precision	0.9856	0.9735	0.9742	0.9116	0.9361	0.9500	0.9604
	Recall	0.9871	0.9722	0.9789	0.9473	0.8490	0.9167	0.9592
	F1-score	0.9862	0.9728	0.9765	0.9271	0.8824	0.9309	0.9587
Electra	Precision	0.8502	0.7194	0.8000	0.5807	0.3500	0.3500	0.5000
	Recall	0.8091	0.5990	0.7612	0.6041	0.2626	0.3294	0.3703
	F1-score	0.7933	0.6213	0.7730	0.5438	0.2966	0.3394	0.3880
MiniLM	Precision	0.9300	0.8994	0.8636	0.6839	0.6000	0.7882	0.8000
	Recall	0.9585	0.8119	0.8588	0.7767	0.5147	0.7549	0.7735
	F1-score	0.9423	0.8308	0.8612	0.7251	0.5470	0.7706	0.7820

(association, composition, aggregation, and inheritance). The results showed significant differences between the models, particularly depending on the nature of the entities to be extracted.

- **UML Classes**: Class identification remains the task most mastered by all the models. BERT, RoBERTa, and XLNet obtained very high F1-scores (0.9855, 0.9817, and 0.9862, respectively), demonstrating their excellent ability to recognize class names, even when they are complex or compound. The MiniLM confirmed its reliability while being lightweight (F1 = 0.9423). In contrast, SpanBERT (F1 = 0.7748) and Electra (F1 = 0.7933) displayed lower performances, indicating difficulties with more complex lexical structures.
- **Attributes**: Attribute extraction remains a more challenging task, mainly because of its lexical proximity to the methods. BERT maintained a clear advantage (F1 = 0.9800), followed by RoBERTa (F1 = 0.9500) and XLNet (F1 = 0.9728), with overall solid results. MiniLM offered satisfactory compromise (F1 = 0.8308). In contrast, SpanBERT (F1 = 0.5303) and Electra (F1 = 0.6213) showed relatively poor performance owing to frequent confusion between attributes and other types of entities.
- **Methods**: The results of the extraction method are generally good for most models. BERT (F1 = 0.9864) and RoBERTa (F1 = 0.9645) confirm their robustness. XLNet and MiniLM, with F1-scores of 0.9765 and 0.8612, respectively, maintained respectable performance. Electra (F1 = 0.7730) and SpanBERT (F1 = 0.7472) remained the same, highlighting the increased difficulty in distinguishing method signatures in less explicit contexts.
- **UML Relations**: Detecting UML relationships (association, aggregation, composition, and inheritance) remains the most complex task. XLNet clearly stands out with F1-scores exceeding 0.92 for association, and close to 0.96 for other relationships, thanks to its ability to capture long relationships. BERT and RoBERTa also maintained excellent results (F1 > 0.90 for all relationships). MiniLM performs well on associations (F1 = 0.7251), but remains modest for more complex relationships. SpanBERT and Electra displayed poor performances, particularly on composition and inheritance, with F1-scores often below 0.60, revealing difficulties in contextual interpretation.

These results (Figure 5) confirm that BERT, RoBERTa, and XLNet were the most suitable for UML extraction. They allow us to consider reliable automation of the generation of UML class diagrams from textual specifications, particularly in the context of model-driven architecture (MDA) pipelines.

C. DSL VALIDATION

Beyond UML entity recognition, the pipeline quality also relies on the reliability of the intermediate DSL representation. This section evaluates syntactic validity, the rate of necessary corrections, and the robustness of the DSL in the face of linguistic variability in the specifications. The goal is to verify the extent to which automatically generated structures are usable or require adjustments before being transformed into UML class diagrams or codes.

1) SYNTACTIC ACCURACY

The syntactic correctness of the DSL reflects the system's ability to produce description blocks that fully conform to the defined grammar (Figure 2) without structural errors or format inconsistencies [58]. This conformity is automatically ensured by a dedicated parser (Figure 3), which systematically checks the syntactic validity of each generated block.

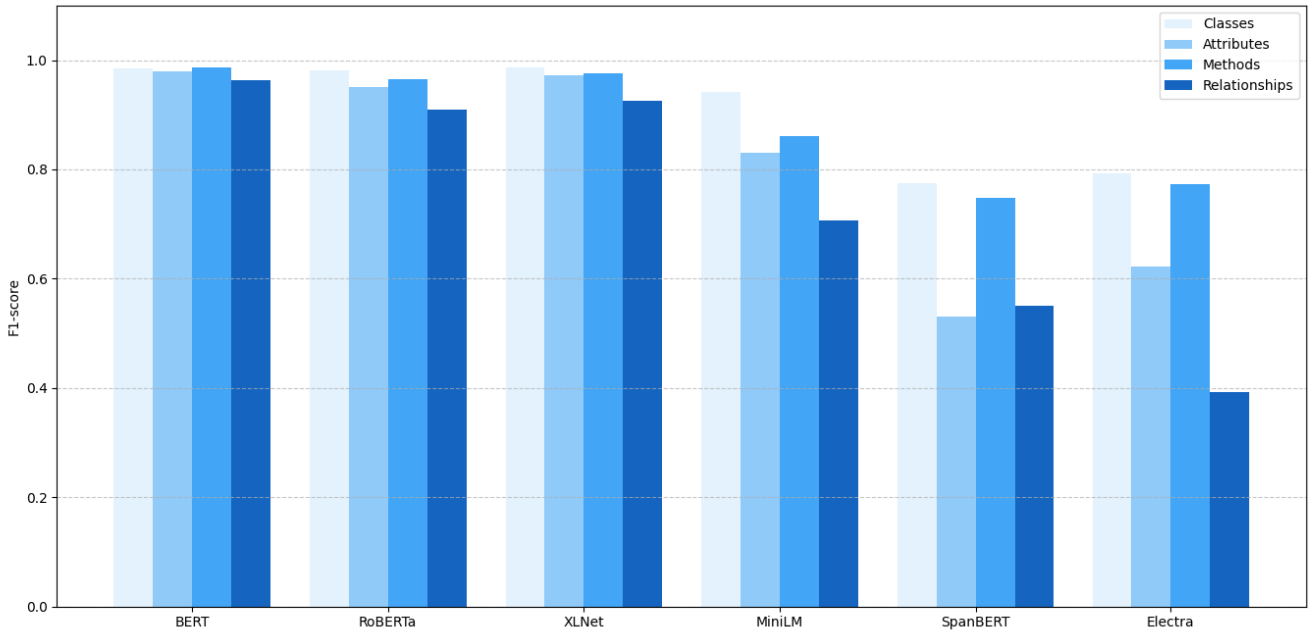

FIGURE 5. Comparison of the models' performances on UML extraction.

Owing to the explicit hierarchical design of the DSL and its strict rules, 100% of the DSL blocks generated in the test set (26 specifications) were recognized as syntactically valid on the first attempt. No parsing errors or manual corrections were necessary, demonstrating the robustness of the language and its perfect fit with the structure of the extracted entities.

These results show that syntactic validity is not a weak point in the process; once the entities are correctly identified, the DSL guarantees reliable, stable generation that conforms to the defined grammar. Thus, it acts as a rigorous formalization layer, ensuring a smooth transition to usable representations.

2) MANUAL CORRECTION RATE

Another reliability indicator is the rate of manual corrections required for the generated DSL blocks before their transformation into UML class diagrams. Based on the evaluation of the BERT model, selected for its high performance (F1-scores above 96% for all UML entities), the majority of DSL blocks were produced correctly or almost correctly. Based on the specifications of the test set, the following results were obtained:

- 74% of DSL blocks were accepted without any modification.
- 20% required minor edits (e.g., adding an omitted attribute or correcting a method name).
- 6% requested a partial class or relationship rewrite, usually due to imprecise interpretation of complex relationships.

These results are consistent with BERT's performance in semantic extraction and confirm that, in the overwhelming majority of cases, the generated DSL is directly usable or requires only minimal adjustments. This reinforces its role as a reliable, editable, and robust pivot layer between textual specifications and UML modeling.

3) ROBUSTNESS TO LINGUISTIC VARIATION

The robustness of the DSL was evaluated based on specifications expressing the same UML concepts through a variety of linguistic styles: active or passive voice, nominalizations, enumerations, and long or elliptical sentences. The results show that the BERT model, coupled with DSL, effectively handled this syntactic diversity in the majority of cases, particularly for the identification of classes and methods.

However, some errors persist, particularly in the following situations:

- Implicit relationships, expressed by vague or contextual prepositions (`used by`, `responsible for`).
- Attributes inserted into relative clauses that dilute the main information.
- Multilingual or poorly punctuated text hinders the correct segmentation of entities.

Despite these limitations, DSL structuring plays a key role in syntactic normalization by transforming diverse text snippets into clear, uniform, and formal blocks. This mechanism stabilizes the detected structures and enhances the reliability of the entire UML modeling pipeline.

D. UML AND CODE GENERATION

The evaluation of our pipeline was not limited to entity recognition or DSL structuring. It must also measure the ability of the system to produce usable software artifacts,

A computer consists of one or more monitors, a case, an optional mouse, and a keyboard. A case has a metal chassis, a motherboard, several memory modules (RAM, ROM, and cache), an optional fan, storage media (floppy disk, hard disk, CD-ROM, DVD-ROM, etc.), and peripheral cards (sound, network, graphics). A computer always has at least one floppy disk drive or hard disk.

FIGURE 6. Scenario 1 (Hardware architecture of a personal computer).

A document includes a table of contents, an index, and several chapters. Each chapter includes several paragraphs. The document can use up to three standardized fonts for publishing documents at this publisher.

If a document is deleted, the chapters, paragraphs, index, and table of contents are also deleted. The images in each paragraph illustrate the content; without it, they are meaningless.

The chapter is identified by a number, the index by a name, the table of contents by a name, and the paragraph by a number starting at 1 and ending with the last paragraph of the chapter.

The font is identified by a font number.

FIGURE 7. Scenario 2 (Logical structure of an editorial document).

We aim to automate the management of a municipal library.
To achieve this, we analyzed its operations and identified the following rules and characteristics:
Each member is identified by a first name and a last name.
The library contains documents and registered members.
Members can be registered or removed upon request.
New documents are periodically added to the collection.
Documents can be of type newspaper or volume.
Volumes are classified as dictionaries, books, or comics.
Each document has a title.
Volumes also specify an author.
Comics additionally indicate the name of the designer.
Newspapers include a publication date, in addition to the standard document information.
Among all documents, only books are eligible for borrowing.
A member is allowed to borrow or return a book.
It must be possible to check which books are currently borrowed by a member.
When a book is borrowed, a return date is defined at the moment of the loan, and this date may be extended upon request.

FIGURE 8. Scenario 3 (Modeling a municipal library).

including readable UML class diagrams and structured Python source code. This subsection presents qualitative results obtained from concrete use cases, analyzing the completeness of the generated model and the syntactic consistency of the produced code.

1) TESTED USE CASES

Three thematic scenarios (Figure 6, Figure 7, Figure 8) were used to validate the complete pipeline chain (based on BERT) from textual input to the automatic generation of UML diagrams and Python code.

Each case was expressed in natural language specifications and then submitted to the full pipeline. UML diagrams and Python code were automatically generated from the intermediate DSL.

2) RESULTS OBTAINED

The UML class diagrams automatically generated by the pipeline (Figure 9) were compared with those obtained after manual review of the DSL by an expert (Figure 10) to measure the gap between the automatic generation and a version optimized by human intervention. Similarly, the corresponding Python source code was compared with the reference implementations to evaluate the structural and semantic fidelity of the generation.

Table 9 synthesizes the pipeline's performance across the three tested scenarios, integrating both classical extraction metrics and complementary criteria of syntactic validity, semantic fidelity, and code executability.

The results demonstrate satisfactory overall reliability, with a good compromise between accuracy and coverage. Complementary criteria confirm the robustness of the pipeline: the DSL maintains strong syntactic validity, the UML diagrams remain semantically faithful to the ground truth, and the generated code was executable in all studied cases. However, the variability observed between scenarios highlights the dependence of the system on the linguistic and conceptual structure of the source text.

3) COMPARATIVE PERFORMANCE ASSESSMENT

Across all three scenarios tested, we also compared the performance of our approach to that of generative models (GPT-3.5 and DeepSeek [59]) according to several complementary criteria (Table 10).

The results revealed significant differences between the three approaches. Our pipeline is distinguished by its robustness in UML structuring, with high and consistent scores, but shows limitations in terms of code quality and consideration of business rules. GPT-3.5 offers a balanced compromise between structuring and generation but remains slightly behind DeepSeek. The latter dominates most technical criteria, reaching scores close to 0.95 in UML, code quality, and business rule management, although it presents a relative weakness in pedagogical clarity. These trends are visually confirmed by the graphs in Figure 11, where the average per scenario and the overall comparison between the models highlight these differences.

In conclusion, our pipeline, thanks to the DSL, demonstrates a promising generalization capacity and solid methodological anchoring, but its performance remains sensitive to the syntactic and semantic complexity of the source texts. Generative models, on the other hand, offer great flexibility but require additional quality controls. Coupling the two approaches thus emerges as a promising avenue: the rigor of our pipeline (validation, logging, and deterministic corrections) could usefully complement the generative power of LLMs, reinforcing both the reliability and richness of the produced artifacts.

4) CODE CONSISTENCY AND GENERABILITY MEASURES

The generated Python code corresponds to a skeleton conforming to the obtained UML diagram, where each class includes a constructor `__init__`, initialization of attributes, and empty methods (`pass`). This code is not functional in itself, but provides a structured basis for prototyping or iterative development (Figure 12).

The results show that 100% of UML classes produced valid and executable Python code without errors. However, imperfections persist, including generic attribute names, duplicates, or poorly discriminating methods. These limitations do not prevent execution, but require refinement for production use. Qualitatively, the cross-evaluation with

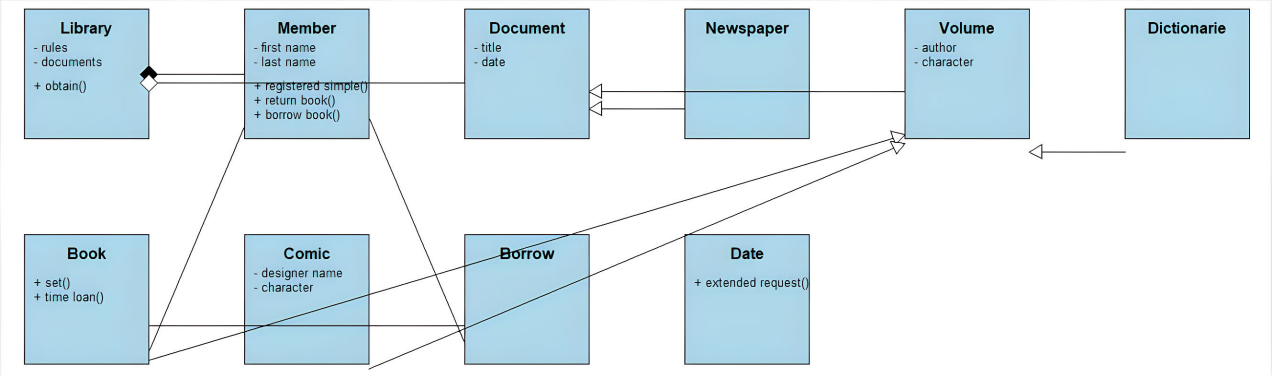

FIGURE 9. Class diagram generated by the pipeline for scenario 3.

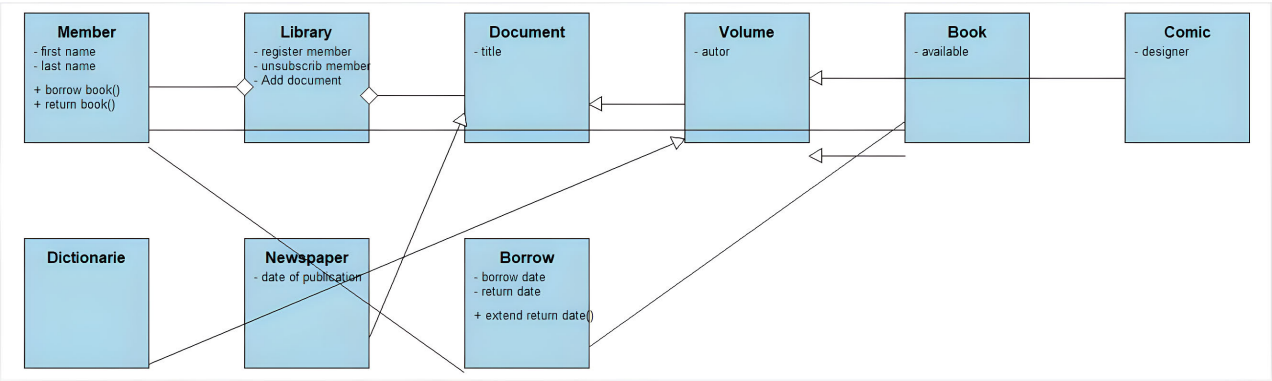

FIGURE 10. Class diagram corrected by the expert for scenario 3.

TABLE 9. Pipeline performance metrics: Extraction accuracy, syntactic validity, semantic fidelity, and code executability.

Scenario	Precision	Recall	F1-score	DSL syntactic validity	UML semantic fidelity	Code executability
1	100%	100%	100%	100%	100%	100%
2	69%	78%	73%	95%	86%	100%
3	67%	73%	69%	94%	81%	100%
Average	**78.7%**	**83.7%**	**80.7%**	**96.3%**	**89%**	**100%**

GPT-3.5 and DeepSeek highlights a clear trend: our pipeline is robust in UML structuring (average score of 0.8), whereas generative models outperform our approach in code quality and business rule management (average of 0.5 vs. 0.71 and 0.91, respectively). On the other hand, our solution retains a relative advantage in pedagogical clarity because of the transparency offered by the DSL and integrated validation mechanisms.

These performances validate the robustness of the Platform-Specific Model (PSM) level and confirm the relevance of DSL as a reliable intermediary between textual understanding and code generation. The pipeline, tested on scenarios of increasing complexity, thus demonstrates the technical feasibility and structural consistency of the model-driven architecture (MDA) approach, from natural language text (CIM) to the intermediate DSL (PIM), then to the UML diagram (PSM) and the generated Python code. The whole constitutes a solution adapted to the phases of rapid prototyping, model-driven engineering and automation of object-oriented design, while opening the way to a hybrid coupling between deterministic pipelines and generative models.

VII. DISCUSSION

The experimental results obtained across the entire pipeline demonstrate the value of a hybrid approach combining the NLP, DSL, and MDA principles for automating software modeling. One of the major contributions of this approach is the introduction of an intermediate textual DSL, positioned as a pivotal representation between linguistic analysis and the generation of formal software artifacts. This DSL provides a new level of control for the automatic transformation

TABLE 10. Comparison of UML quality and generated code between our approach, GPT-3.5 and DeepSeek.

Scenario	Model	UML quality	Code quality	Business rules	Executability	Educational clarity	Average
1	Our approach	1.0	0.4	0.5	1.0	0.7	0.72
	GPT-3.5	1.0	0.7	0.7	1.0	0.6	0.80
	DeepSeek	1.0	0.9	0.9	1.0	0.5	0.86
2	Our approach	0.7	0.5	0.5	1.0	0.8	0.70
	GPT-3.5	0.8	0.7	0.7	1.0	0.6	0.76
	DeepSeek	0.9	0.9	0.9	1.0	0.5	0.84
3	Our approach	0.7	0.5	0.5	1.0	0.7	0.66
	GPT-3.5	0.8	0.7	0.8	1.0	0.6	0.78
	DeepSeek	0.9	0.9	1.0	1.0	0.5	0.86

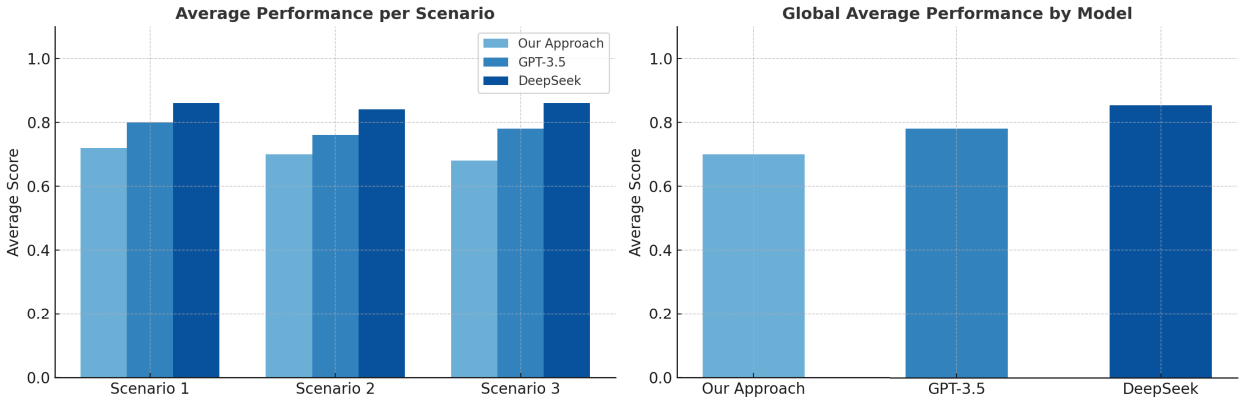

FIGURE 11. Comparison of performance by scenario and by model.

```python
class Library:
    def __init__(self, rules: str = None, documents: str = None):
        self.document = Document()  # Aggregation
        self.member = Member()  # Composition
        self.rules = rules
        self.documents = documents
    def obtain(self):
        pass

class Member:
    def __init__(self, first_name: str = None, last_name: str = None):
        self.first_name = first_name
        self.last_name = last_name
    def registered_simple_request(self):
        pass
    def return_book(self):
        pass
    def borrow_books(self):
        pass

class Document(Volume):
    def __init__(self, title: str = None, date: str = None):
        self.title = title
        self.date = date

class Newspaper:
    def __init__(self):
        pass

class Volume(Book):
    def __init__(self, author: str = None, character: str = None):
        self.author = author
        self.character = character
```

FIGURE 12. Generated python code snippet for scenario 3.

processes. Unlike direct generation approaches, it allows the user to intervene in the extracted entities, verifying, correcting, or enriching them before they are translated into UML diagrams or codes. This significantly enhances the traceability and flexibility of the system, making it possible to track the software production chain from start to finish, based on natural language specifications.

On a practical level, the DSL's syntax, designed to be readable and hierarchical, makes it easy for non-technical profiles to use. Thus, it has become a medium for exchange between developers, analysts, and business users. In addition, by serving as a formal entry point for diagram or code generation tools, it provides a way to enhance interoperability with other software engineering tools, and can be enriched to accommodate semantic annotations, business rules, or additional constraints.

Compared to direct approaches without intermediate structuring, our method stands out for its ability to keep the user in the loop, make the system's choices visible, and facilitate manual adjustment. Whereas 'NLP-to-UML' pipelines often operate as opaque modules, producing results that are difficult to interpret or correct, the introduction of an explicit DSL enables fine-grained control over the output. This transparency, reinforced by integrated validation mechanisms, constitutes a major advantage: it allows anomalies to be flagged, logged, and, in some cases, automatically corrected when inconsistencies arise during extraction. Thus, the pipeline can absorb part of the errors generated by the NER model, ensuring a more reliable continuity in the generation of artifacts. This dual mechanism — manual control by the user and automatic correction via the DSL — significantly enhances the robustness of the system, while providing valuable traceability for validation, teaching of modeling, and the industrialization of safe and reliable transformation pipelines.

The proposed architecture offers good portability. Because of its modularity, it can be adapted to other languages by

retraining the NER model on a multilingual annotated corpus. DSL can also be specialized for different application domains by integrating specific business structures. Similarly, the code generation engine, initially designed for Python, can be reconfigured to produce other object-oriented programming languages such as Java and C#. The choice of Python as a first target is justified by its readability, its widespread use in rapid prototyping, and the simplicity of mapping UML constructs into Python classes and methods. Moreover, Python allows fast validation of the generated code thanks to its lightweight execution and its compatibility with numerous testing frameworks. Nevertheless, the modular design of the code generation component ensures that language-specific templates can be incorporated with minimal effort, as in Java or C#, where the engine can produce classes with explicit types and adapted structures while preserving UML-to-code mapping principles. This adaptability opens the way for the wider use of pipeline in CASE[5](Computer-Aided Software Engineering) environments, rapid prototyping tools, or model-driven software production chains.

VIII. PERSPECTIVES

The results obtained in this study open several promising avenues for the future evolution of the system. The first area of enrichment concerns the direct integration of business rules into DSL. Currently, the focus is on the static structure of the system (classes, attributes, methods, and relationships). The DSL can be extended to capture functional or behavioral constraints, thus facilitating the generation of richer models that are closer to real requirements. In parallel, the development of a visual DSL integrated into an interactive interface would offer a better user experience. A graphical interface allowing editing of the model by drag and drop, while automatically synchronizing modifications with the textual DSL code, would strengthen the usability of the system, making it accessible to non-technical profiles.

Another fundamental lever for improvement concerns the enrichment and scalability of the dataset used for training and evaluation. Our approach can be adapted to larger and more diverse corpora by leveraging semi-automatic annotation supported by linguistic rules and pretrained models, which reduces manual effort while preserving quality, as well as incremental fine-tuning of NER models on specialized subcorpora, enabling progressive adaptation to new domains. Moreover, linguistic and structural diversification of the corpus would enhance the system's generalization ability, its robustness to less standardized formulations, and its applicability to documents from different domains. In addition, the integration of software-oriented pretrained models (CodeBERT, GraphCodeBERT…) provides a promising avenue to strengthen UML entity extraction and the detection of complex relationships. Finally, multilingual support is a key perspective, through the creation of annotated corpora in multiple languages (French, Arabic, Spanish, etc.) and the fine-tuning of adapted NER models.

The comparative analysis also highlights the complementary strengths and limitations between our pipeline and generative models. Based on the DSL, our pipeline provides a robust methodology and solid generalization capabilities, but remains sensitive to the syntactic and semantic complexity of source texts. LLMs, while more flexible, require stronger quality control. Thus, the hybridization of the two approaches appears promising: the rigor of the pipeline, grounded in validation, logging, and deterministic corrections, could be combined with the generative power of LLMs, enhancing both the reliability and richness of the produced artifacts. To address implicit relations and coreference issues, automatic coreference resolution and prompt-engineering strategies for LLMs could be integrated, providing more accurate entity linking and relationship inference.

Other perspectives concern the extension of the pipeline to other types of UML diagrams, such as sequence, activity, or state diagrams. These dynamic representations require a more detailed understanding of the system's interactions and behaviors but can rely on the same extraction logic, enriched by new types of semantic labels. Moreover, to meet software production requirements, code generation can be extended to other programming languages, such as Java or C#, by adapting the transformation rules to each target. These developments would strengthen the adaptability of the system in heterogeneous environments. Finally, large-scale industrial testing and benchmarking remain essential to validate the robustness, efficiency, and practical usability of the proposed pipeline in real development contexts, positioning it as a modular, scalable, and generalizable solution in automated software production chains.

IX. CONCLUSION

Software modeling is a fundamental step in the development cycle, but it remains largely manual, time-consuming, and subject to multiple interpretations, especially when specifications are formulated in natural language. This work is part of the desire to reduce this cognitive load and make the production of formal software models more reliable by proposing an automated chain, structured around the principles of model-driven architecture (MDA), enriched by the integration of an intermediate DSL, and driven by advanced natural language processing (NLP) techniques.

The article thus presents an original solution based on a hybrid combination (NLP, DSL, and MDA) that allows movement, from free text (CIM level) to a structured intermediate textual DSL (PIM level), then to a formal UML model (PSM level), and finally to an executable source code. This complete chain is supported by several major contributions. It is based on the development of a corpus annotated according to an enriched IOB schema, specifically adapted to UML entities, such as classes, attributes, methods, and relations. It also relies on fine-tuning transformer models for entity recognition from natural language specifications.

[5]It refers to the set of software tools that assist engineers in the various stages of software development.

Added to this is the design of textual DSL, conceived as a pivotal intermediate representation between linguistic extraction and formal modeling. Finally, the entire process is orchestrated in an integrated end-to-end pipeline, from semantic text understanding to automatic generation of UML diagrams and Python code, with a dedicated visualization interface.

The experimental results confirm the effectiveness of the proposed system, which combines UML entity extraction via BERT, formalization via an intermediate textual DSL, and automatic generation of UML diagrams and Python code. The high performance of the classes and methods illustrates the model's ability to capture the structure of natural language specifications. The evaluation of three varied scenarios achieved an average F1-score of 80.7%, reflecting a good compromise between precision and recall, even in the presence of implicit formulations or complex business logic. The DSL plays a central role in structuring, traceability, and control, whereas the generated code, although skeletal, constitutes a usable basis for prototyping. Thus, the entire pipeline demonstrated its robustness, generalizability, and usefulness in an automated object-oriented modeling process.

The introduction of the DSL as a pivotal layer is one of the most significant contributions of this study. It not only ensures a clear separation between the linguistic comprehension process and the software transformation process but also provides the user with the ability to interact, edit, or correct the generated model. In other words, this approach puts humans back in the loop of the automated process, reconciling the power of artificial intelligence with the need for human supervision. Unlike the direct approaches (NLP to UML), which are often perceived as opaque modules, our solution promotes transparency, interpretability, and scalability.

In summary, this work shows that it is possible to articulate informal specifications and formal software artifacts with rigor and efficiency based on a modular, robust, and controllable architecture. It provides concrete perspectives for the construction of intelligent assistants for software modeling, semi-automatic generation of object-oriented prototypes, and integration of such chains into modeling environments or teaching platforms. The future addition of business rules, semantic validation mechanisms, multilingual support, additional generation targets (Java, C#, etc.), and the expansion of training and evaluation dataset could transform this approach into an adaptable and generalizable foundation for model-driven software engineering.

REFERENCES

[1] G. Booch, *The Unified Modeling Language User Guide*. London, U.K.: Pearson Education, 2005.

[2] OMG, "Unified modeling language (OMG UML)," Version 2.5.1, Object Manage. Group, Needham, MA, USA, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[3] C. Larman, *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. London, U.K.: Pearson Education, 2012.

[4] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Reading, MA, USA: Addison-Wesley, 2018.

[5] T. Yue, L. C. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," *Requirements Eng.*, vol. 16, no. 2, pp. 75–99, Jun. 2011.

[6] H. Jia, R. Morris, H. Ye, F. Sarro, and S. Mechtaev, "Automated repair of ambiguous problem descriptions for LLM-based code generation," 2025, *arXiv:2505.07270*.

[7] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed. Draft Version, 2023. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3

[8] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Reading, MA, USA: Addison-Wesley, 2003.

[9] T. B. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 1877–1901.

[10] P. Lemberger and F. R. Chaumartin, *Le Traitement Automatique Des Langues: Comprendre Les Textes Gráce Á L'Intelligence Artificielle*. Paris, France: Dunod, 2020.

[11] M. Calamo, M. Mecella, and M. Snoeck, "Assessing the suitability of large language models in generating UML class diagrams as conceptual models," in *Proc. Int. Conf. Bus. Process Model.*, 2025, pp. 211–226.

[12] Q. Lyu, M. Apidianaki, and C. Callison-Burch, "Towards faithful model explanation in NLP: A survey," *Comput. Linguistics*, vol. 50, no. 2, pp. 657–723, Jun. 2024.

[13] M. Danilevsky, K. Qian, R. Aharonov, Y. Katsis, B. Kawas, and P. Sen, "A survey of the state of explainable AI for natural language processing," 2020, *arXiv:2010.00711*.

[14] J. E. Zini and M. Awad, "On the explainability of natural language processing deep models," *ACM Comput. Surv.*, vol. 55, no. 5, pp. 1–31, May 2023.

[15] K. Hölldobler, B. Rumpe, and I. Weisemöller, "Systematically deriving domain-specific transformation languages," in *Proc. ACM/IEEE 18th Int. Conf. Model Driven Eng. Lang. Syst. (MODELS)*, Sep. 2015, pp. 136–145.

[16] E. F. T. K. Sang and F. De Meulder, "Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition," 2003, *arXiv:cs/0306050*.

[17] L. Radosky and I. Polasek, "Executable multi-layered software," 2025, *arXiv:2501.08186*.

[18] IEEE Computer Society, *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Standard 830-1998, IEEE, New York, NY, USA, 2011.

[19] I. K. Raharjana, D. Siahaan, and C. Fatichah, "User stories and natural language processing: A systematic literature review," *IEEE Access*, vol. 9, pp. 53811–53826, 2021.

[20] R. Sonbol, G. Rebdawi, and N. Ghneim, "The use of NLP-based text representation techniques to support requirement engineering tasks: A systematic mapping review," *IEEE Access*, vol. 10, pp. 62811–62830, 2022.

[21] M. Sabetzadeh and C. Arora, "Practical guidelines for the selection and evaluation of NLP techniques in RE," *arXiv:2401.00000*.

[22] P. Evitts and D. Hinchcliffe, *A UML Pattern Language*. IN, USA: Indianapolis, 2000.

[23] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. San Rafael, CA, USA: Morgan & Claypool, 2017.

[24] J. Rumbaugh, *The Unified Modeling Language Reference Manual*. London, U.K.: Pearson Education, 2005.

[25] M. Melouk, Y. Rhazali, and H. Youssef, "An approach for transforming CIM to PIM up to PSM in MDA," *Proc. Comput. Sci.*, vol. 170, pp. 869–874, Jan. 2020.

[26] R. Soley, "Model driven architecture," Object Management Group, Needham, MA, USA, OMG White Paper omg/2000-11-05, Nov. 2000.

[27] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003.

[28] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, 2018, pp. 4171–4186.

[29] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[30] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy, "SpanBERT: Improving pre-training by representing and predicting spans," *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 64–77, Dec. 2020.

[31] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 32, 2019, pp. 5754–5764.
[32] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," 2020, *arXiv:2003.10555*.
[33] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," in *Proc. Adv. Neural Inf. Process. Syst.*, 2000, pp. 5776–5788.
[34] Z. Babaalla, H. Abdelmalek, A. Jakimi, and M. Oualla, "Extraction of UML class diagrams using deep learning: Comparative study and critical analysis," *Proc. Comput. Sci.*, vol. 236, pp. 452–459, Jan. 2024.
[35] P. More and R. Phalnikar, "Generating UML diagrams from natural language specifications," *Int. J. Appl. Inf. Syst.*, vol. 1, no. 8, pp. 19–23, Apr. 2012.
[36] M. Mohanan and P. Samuel, "Open NLP based refinement of software requirements," *Int. J. Comput. Inf. Syst. Ind. Manag. Appl.*, vol. 8, p. 8, Jan. 2016.
[37] G. A. Miller, "WordNet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, 1995.
[38] S. K. Shinde, V. Bhojane, and P. Mahajan, "NLP based object oriented analysis and design from requirement specification," *Int. J. Comput. Appl.*, vol. 47, no. 21, pp. 30–34, Jun. 2012.
[39] A. R. Martinez, "Part-of-speech tagging," *Wiley Interdiscipl. Rev., Comput. Statist.*, vol. 4, no. 1, pp. 107–113, 2012.
[40] N. Bashir, M. Bilal, M. Liaqat, M. Marjani, N. Malik, and M. Ali, "Modeling class diagram using NLP in object-oriented designing," in *Proc. Nat. Comput. Colleges Conf. (NCCC)*, Mar. 2021, pp. 1–6.
[41] Shweta, R. Sanyal, and B. Ghoshal, "A hybrid approach to extract conceptual diagram from software requirements," *Sci. Comput. Program.*, vol. 239, Jan. 2025, Art. no. 103186.
[42] A. D. H. Aunsa, V. S. Barletta, P. Buono, C. M. N. Faisal, A. Piccinno, Z. Saeed, and L. Provenzano, "Defining classes and semantic relationships from user scenarios through a heuristic approach," in *Proc. Int. Symp. End User Develop.*, 2025, pp. 331–343.
[43] S. Yang and H. Sahraoui, "Towards automatically extracting UML class diagrams from natural language specifications," in *Proc. 25th Int. Conf. Model Driven Eng. Lang. Syst.*, Oct. 2022, pp. 396–403.
[44] R. Saini, G. Mussbacher, J. L. C. Guo, and J. Kienzle, "Automated, interactive, and traceable domain modelling empowered by artificial intelligence," *Softw. Syst. Model.*, vol. 21, no. 3, pp. 1015–1045, Jun. 2022.
[45] Y. Rigou and I. Khriss, "A deep learning approach to UML class diagrams discovery from textual specifications of software systems," in *Proc. SAI Intell. Syst. Conf.*, Sep. 2022, pp. 706–725.
[46] OpenAI, "Optimizing language models for dialogue," OpenAI Blog, San Francisco, CA, USA, Nov. 2022.
[47] D. Rouabhia and I. Hadjadj, "Behavioral augmentation of UML class diagrams: An empirical study of large language models for method generation," 2025, *arXiv:2506.00788*.
[48] B. Al-Ahmad, A. Alsobeh, O. Meqdadi, and N. Shaikh, "A Students-centric evaluation survey for exploring the impact of LLMs on UML modeling," *Information*, vol. 16, no. 7, p. 565, Jul. 2025, doi: 10.3390/info16070565.
[49] K. Khakzad Shahandashti, M. Sivakumar, M. Mahdi Mohajer, A. B. Belle, S. Wang, and T. C. Lethbridge, "Evaluating the effectiveness of GPT-4 turbo in creating defeaters for assurance cases," 2024, *arXiv:2401.17991*.
[50] V. Campanello, S. Shahbaz, V. Indykov, and D. Strüber, "On the use of GPT-4 in the reverse engineering of class diagrams," *J. Object Technol.*, vol. 24, no. 2, pp. 2:1–14, 2025, doi: 10.5381/jot.2025.24.2.a14.
[51] J. Achiam et al., "GPT-4 technical report," 2023, *arXiv:2303.08774*.
[52] L. A. Ramshaw and M. P. Marcus, "Text chunking using transformation-based learning," in *Natural Language Processing Using Very Large Corpora*. Dordrecht, The Netherlands: Springer, 1999, pp. 157–176.
[53] T. Wolf et al., "HuggingFace's transformers: State-of-the-art natural language processing," 2019, *arXiv:1910.03771*.
[54] P. Zhou, X. Xie, Z. Lin, and S. Yan, "Towards understanding convergence and generalization of AdamW," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 46, no. 9, pp. 6486–6493, Sep. 2024.
[55] E. Shinan, "Lark: A modern parsing library for Python," GitHub Repository, San Francisco, CA, USA. [Online]. Available: https://github.com/lark-parser/lark
[56] Q. Charatan and A. Kans, "Python graphics with Tkinter," in *Programming in Two Semesters: Using Python and Java*. Cham, Switzerland: Springer, 2022, pp. 211–254.
[57] D. S. Batista, "Named-entity evaluation metrics based on entity-level," Blog, May 2018. [Online]. Available: https://www.davidsbatista.net/blog/2018/05/09/Named_Entity_Evaluation/
[58] L. Netz, J. Reimer, and B. Rumpe, "Using grammar masking to ensure syntactic validity in LLM-based modeling tasks," in *Proc. ACM/IEEE 27th Int. Conf. Model Driven Eng. Lang. Syst.*, Sep. 2024, pp. 115–122.
[59] A. Liu et al., "DeepSeek-V3 technical report," 2024, *arXiv:2412.19437*.

ZAKARIA BABAALLA (Member, IEEE) received the master's degree in software engineering and information systems from the Faculty of Science and Techniques of Errachidia, University of Moulay Ismail, Meknes, Morocco, in 2021. He is currently pursuing the Ph.D. degree with the GL-ISI Team. He also teaches at the qualifying level within the Ministry of Education in Morocco. His research interests include software engineering, particularly on the automation of UML class diagram extraction from textual system specifications, relying notably on deep learning and natural language processing techniques.

ABDESLAM JAKIMI (Member, IEEE) is a Professor of computer sciences and engineering with the Department of Computer Sciences, Faculty of Sciences and Technology of Errachidia, University of Moulay Ismail. He is currently the Head of the Department of Computer Sciences, the President of the Scientific Association for Academic Researchers (ASCA), and the Chair of the International Conference in Computer Science and Applications (ICCSA). He has been teaching in the areas of software engineering and artificial intelligence since 2010. His research interests include software engineering, artificial intelligence, data visualization, blockchain, the IoT, and smart applications. He is also a member of several scientific associations. He served as the chairperson and a member for steering committees and program committees for several international conferences, workshops, and symposiums.

MOHAMED OUALLA received the Ph.D. degree in computer science from the Faculty of Sciences, Ibn Tofaïl University, in July 2017. He is currently an Associate Professor with the Faculty of Sciences and Technology of Errachidia, Department of Computer Science, Moulay Ismaïl University. He is also the coordinator of an engineering program and the treasurer of the Scientific Association of Academic Researchers (ASCA). He teaches in the fields of software development, blockchain and Web3, and artificial intelligence. His research interests include software engineering, blockchain, the IoT, artificial intelligence, and cloud computing.