Model-Driven Development of Chatbot Microservices

Adel Vahdati and Raman Ramsin(✉)

Department of Computer Engineering, Sharif University of Technology, Azadi Avenue, Tehran, Iran
{adel.vahdati97,ramsin}@sharif.edu

Abstract. Conversational agents and chatbots are gaining prominence in software systems by providing functionalities beyond traditional GUIs. These intelligent assistants facilitate software development tasks such as deployment, error handling, and scheduling. However, chatbot development remains challenging due to productivity, reusability, scalability, and maintainability issues.

We propose a model-driven methodology for chatbot development in four phases: computation-independent model construction, platform-independent model construction, platform-specific model construction, and code generation. The methodology enhances productivity by automating code generation and improves reusability through computation-independent and platform-independent definitions. Additionally, it introduces a novel approach to categorizing, enumerating, parameterizing, and representing user intents. We obtain data for training natural language understanding services and leverage microservice architecture and architectural design patterns to enhance scalability, maintainability, and interoperability. The methodology has been evaluated based on three groups of criteria: criteria relevant to the generic software development lifecycle, criteria specific to model-driven development, and criteria relevant to chatbots.

Keyword: Model-driven methodology · Chatbot microservice · Natural Language processing

1 Introduction

Software systems are now adopting a new type of interface beyond the traditional GUI. Conversational agents, intelligent assistants, and conversational user interfaces (CUI) are becoming increasingly popular [1]. Additionally, conversational agents are already aiding software development activities such as automating deployment tasks, assigning errors and issues to team members, and scheduling tasks [2]. Their integration into social networks as communication channels has enhanced stakeholder participation in task automation and collaborative modeling [2, 3].

In conversational agents, user interaction occurs through text, voice messages, or interactive images (as in Gesture Bots). The agent always has a dialogue mechanism, with the only difference being the interface or medium through which this dialogue happens [1].

A chatbot mimics human conversation through two-way communication using natural language. To provide a useful conversation, a chatbot platform must offer the following features [4]:

- Natural language processing (NLP) and natural language understanding (NLU): understanding user input and extracting relevant information.
- Conversation flow management
- Performing necessary actions: such as searching a database or calling other services.

Leading companies like Google (Dialogflow), Microsoft (Microsoft Bot Framework), Amazon (Amazon Lex), and IBM (Watson) have provided various tools and frameworks to create conversational agents [5]. These tools offer a framework, cloud environment, and GUI to define the conversation flow. Existing frameworks utilize machine learning (ML) algorithms to identify the user's intention based on the message sent by the user; for instance, Amazon provides services such as Lex, Comprehend, and Polly to help create intelligent assistants [6].

In model-driven development (MDD), a system is modeled at different levels of abstraction. Model transformations are used to refine high-level abstract models into lower-level models or code [7, 8]. In chatbot development, MDD can help reduce accidental complexity [8], leading to higher productivity, performance, and reusability [9].

We propose a model-driven methodology that guides the process of creating a chatbot. This methodology encompasses four phases: construction at the computation-independent modeling (CIM) level, construction at the platform-independent modeling (PIM) level, construction at the platform-specific modeling (PSM) level, and construction at code level; sets of activities and products have been specified for each phase, and metamodels have been defined at different levels of abstraction to describe the problem/solution domains.

The CIM-level construction phase of our proposed methodology focuses on understanding the problem domain and user goals through natural language conversations. This phase involves creating CIM models to analyze the problem domain and requirements, resulting in a requirements model. User goals are then extracted to form an intent model. Our approach is generic and adaptable to various contexts, categorizing and enumerating user intents, identifying necessary parameters, and using a metamodel for intent representation. We also introduce the CRAC method (Concept, Responsibilities, Asynchronous Collaboration) for analyzing the problem domain and extracting the requirements, modeling domain concepts and system functions, and capturing asynchronous system interactions through events.

The PIM-level construction phase focuses on designing the chatbot microservice and its interactions with essential services, using PIM models that are platform-agnostic. This phase involves identifying user intents, extracting necessary information, and employing AI and ML algorithms for training. We leverage MDD techniques, including a domain-specific language (DSL) for describing questions and model-to-text transformations for seamless integration with the NLU microservice. The architecture of the chatbot microservice is defined using patterns like CQRS and API Controller, with metamodels abstractly describing the design aspects. Additionally, we define a bidirectional conversation flow between the chatbot and users, ensuring effective communication through a proposed dialogue flow metamodel.

The PSM-level construction phase involves implementing solutions and generating code in C# within the.NET framework. This phase introduces two metamodels for describing the solution domain using class and interface concepts, and the metamodel for configuring software projects, including external services like NLU and Messaging microservices.

The code-level construction phase introduces a metamodel to model project structures, folders, files, and their contents. PSM-level models are transformed into the solution model using model-to-model (M2M) transformations, and solution code is subsequently generated from this model by model-to-text (M2T) transformations.

This paper is structured as follows: Sect. 2 provides a review of the research background; the challenges of chatbot development are discussed in Sect. 3; in Sect. 4, the proposed methodology and architecture are introduced; the proposed methodology is evaluated in Sect. 5; and in Sect. 6, conclusions and directions for future work are presented.

2 Related Works

Several frameworks have emerged to simplify the creation and deployment of chatbots. Notably, Xatkit [10] is a chatbot development framework that leverages MDD and DSLs. Designers define user intentions and behaviors, binding them to actions and responses. The runtime engine deploys the chatbot, registers intents, establishes connections, and launches external services.

Another web-based environment, CONGA [5], employs a DSL for modeling conversational agents. Specifications are analyzed and compiled into tools like Rasa or Dialogflow. A recommendation component assists in selecting the most suitable tool for chatbot creation.

Mahmood et al. [6] focus on dynamic user interfaces by utilizing microservice architecture and the flexibility of natural languages. User intentions are identified based on requirements, and utterances are mapped to these intents. Open API specifications orchestrate microservices according to their capabilities and availability.

Matic et al. [4] propose an architecture that allows using various natural language understanding (NLU) services without vendor lock-in. They provide a general NLU metamodel and specific metamodels for Dialogflow and Rasa NLU services, along with mapping rules for automatic object creation.

Perez-Soler et al. [3] introduce an automated solution for modeling conversational agents using natural language. Users can express incomplete or inaccurate ideas, and the framework refines the model accordingly.

Ed-douibi et al. [11] suggest a chatbot interface for querying Open Data resources. Users ask questions in natural language, and the chatbot converts them into API requests. The annotated API model configures the chatbot for querying Web APIs, with Xatkit as the generation tool.

Lastly, Perez-Soler et al. [2] demonstrate using a chatbot as an interface for querying domain-specific models, catering to non-technical users. The chatbot model is automatically generated based on the domain metamodel, implemented using Xatkit.

3 Challenges of Chatbot Development

Developing chatbots presents several significant challenges. First, reliance on proprietary components and services during design and runtime creates dependency on service providers [5]. Ensuring compatibility across platforms and different tool providers is also a critical concern [9].

Model-driven development (MDD) can simplify the process of describing various types of user interfaces (UIs) and creating rich UI experiences [1]. However, metamodeling and language engineering remain complex tasks. Additionally, techniques are needed to analyze model quality and tools that can reflect changes in requirements [9].

Current MDD methodologies often lack sufficient focus on requirements engineering [9]. To create chatbots using the MDD approach, employing design patterns and quality metrics becomes essential [9]. Furthermore, chatbots must be maintained and synchronized with evolving requirements, necessitating methods to enhance maintainability, adaptability, and scalability [9]. Table 1 outlines some of the critical questions that arise during chatbot development [12].

Table 1. Critical questions that arise during chatbot development [12].

1	How to find the most suitable tool for creating a chatbot based on its requirements?
2	How to design a chatbot independent of the development tool and platform?
3	How to analyze and evaluate a chatbot before implementation?
4	How to keep up with the rapid growth of the ecosystem and tools for developing chatbots?
5	How to support the migration process of chatbots to a new tool or platform?
6	How to integrate a chatbot with new NLU services provided by different vendors?
7	How to integrate chatbots with new communication channels provided by different vendors?
8	How to solve the coupling between a chatbot and a specific intent recognition service?
9	How to obtain training phrases for ML algorithms to recognize user intents?

4 Proposed Chatbot Development Methodology

Our proposed methodology for chatbot development emphasizes modeling at various abstraction levels, model transformations, and code generation. The abstraction levels provide a structured way to represent chatbot-related concepts and behaviors. We start with high-level conceptual models (CIM-level) that capture user intents, system responses, and domain-specific knowledge. As we refine the models, we move to platform-independent models (PIM-level) that consider architectural patterns and interaction flows. Then, we reach platform-specific models (PSM-level) that address implementation details and technology choices. Finally, code artifacts are generated at the lowest level of abstraction.

Model-to-model transformations play a crucial role in our methodology. They create lower-level models from higher-level ones. For instance, transforming a CIM-level intent model into a PIM-level interaction model bridges the gap between user requirements and system behavior. These transformations ensure consistency and traceability across abstraction levels.

Beyond models, we generate solution code using model-to-text transformations. These transformations map models to actual implementation artifacts. Based on the desired architecture and design patterns, we produce code snippets, service interfaces, and communication protocols. Our overall solution architecture follows the microservice paradigm. Figure 1 shows the overall solution based on the microservice architecture. It includes three essential microservices:

1. Chatbot Microservice: Handles user interactions, natural language understanding (NLU), and context management, and delegates request fulfillment to the relevant Business services.
2. NLU Microservice: Focuses on intent recognition, entity extraction, and language processing.
3. Messaging Microservice: Manages communication channels and message routing.

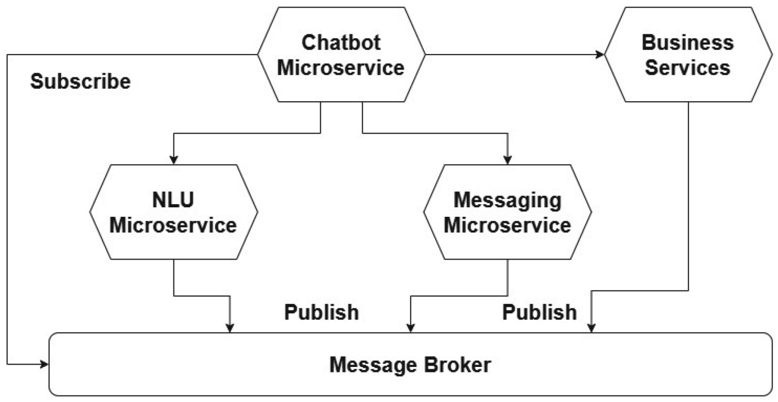

Fig. 1. Overall solution architecture.

Building upon our previous methodology [12], we have tailored the process specifically for developing chatbot microservices. Figure 2 provides an overview of the methodology and the modeling process across different abstraction levels. As seen in this figure, the methodology spans four phases: CIM-level construction, PIM-level construction, PSM-level construction, and Code-level construction. While this paper primarily focuses on the chatbot microservice, the same principles apply to developing the NLU and Messaging microservices: at the CIM level, we create CRAC models specific to NLU and Messaging; the PIM level involves designing CQRS and Controller models tailored for each microservice; based on the desired platform and technology stack, we transform these models to the PSM level; and finally, using model-to-text transformation, solution code is generated for both NLU and Messaging microservices. In summary, our model-driven methodology provides a comprehensive framework for building intelligent conversational agents, extending seamlessly to the auxiliary microservices critical for chatbot functionality.

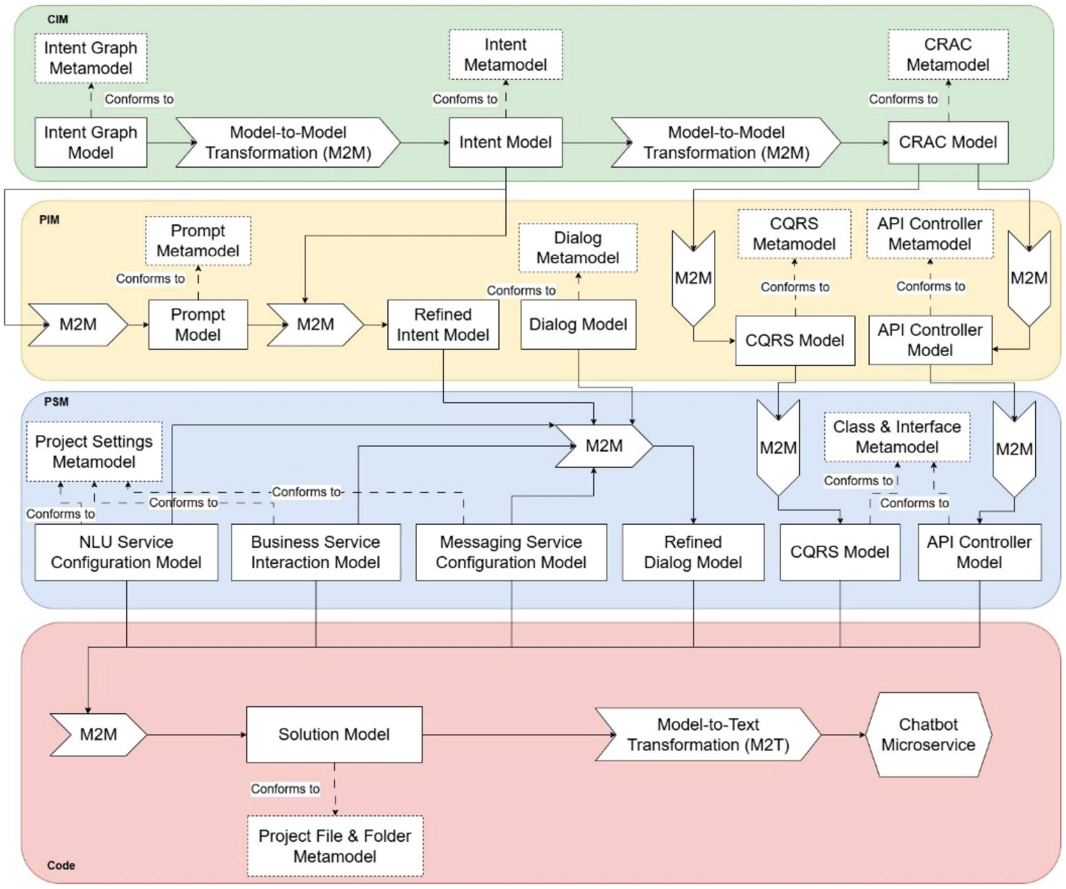

Fig. 2. MDD methodology for developing chatbot microservices.

4.1 CIM-Level Construction

This phase aims to explore the problem domain and understand user goals through natural language conversations. Models are described at the highest level of abstraction, known as the computation-independent model (CIM). Initially, the problem domain and requirements undergo analysis, resulting in a requirements model. Subsequently, user goals are extracted from this model using model transformations, leading to the creation of the intent model.

We propose a generic approach to automatically model user intents expressed through natural language conversations with chatbots. Unlike domain-specific methods, our approach is not tied to any particular problem domain. Instead, it can adapt to various contexts where users interact with chatbots to achieve specific tasks. The key steps in our approach are as follows:

1. Intent Categorization
2. Intent Enumeration
3. Parameter Identification
4. Intent Representation

We begin by categorizing user intents based on the context of the conversation. These intents represent high-level actions or requests that users express. For instance, in the context of infrastructure provisioning, we identify categories such as "Provisioning Resources," "Configuration Management," "Deployment Automation," and "Orchestration." Each category represents a distinct aspect of resource management. Table 2 shows a prompt template for user intent categorization, instructing ChatGPT's response format based on the intent graph metamodel shown in Fig. 3.

Within each category, we enumerate the specific intents that users express. These intents correspond to high-level tasks or actions that users want to perform. For example in the "Provisioning Resources" category, we identify the list of intents as "Create Virtual Machine (VM) (CreateVM)", "Create Container Cluster (CreateCluster)", "Create Load Balancer (CreateLB)", and "Create Database Instance (CreateDB)". Table 3 shows a prompt template for user intent enumeration.

For each intent, we identify the necessary parameters. These parameters capture the essential information required to fulfill the user's request. For example, common parameters of "Create Virtual Machine (VM) (CreateVM)" intent include "vmName", "vmSize", "image", "network", and "storage".

Table 4 shows a prompt template for intent parameters identification.

To capture the essence of user intents, we introduce an intent metamodel (Fig. 4). This metamodel serves as a structured representation that encapsulates the intent, its associated parameters, and their relationships. By leveraging this metamodel, we can automatically generate intent models from conversational data.

We propose a method for analyzing problem domains and extracting the requirements that we call the CRAC method (Concept, Responsibilities, Asynchronous Collaboration); this method provides a structured approach to understanding domain-specific concepts and their associated responsibilities [12].

Table 2. Categorizing user intents (intent categorization).

Template

Categorize user intents within a specific problem domain: {problem_domain}. Provide your answer in json format, including the problemDomain and list of catgories (the name and description of each category).

Prompt

Categorize user intents within a specific problem domain: “Infrastructure as Code”. Provide your answer in json format, including the problemDomain and list of catgories (the name and description of each category).

Answer

```
{
 “problemDomain”: “Infrastructure as Code”,
 “categories”: [
   {
    “name”: “Provisioning Resources”,
    “description”: “Creating, managing, and scaling infrastructure resources using code”
   },
   {
    “name”: “Configuration Management”,
    “description”: “Defining and maintaining consistent configurations for infrastructure components”
   },
   {
    “name”: “Deployment Automation”,
    “description”: “Automating the deployment process for applications and services”
   },
   {
    “name”: “Orchestration”,
    “description”: “Coordinating and managing complex workflows involving multiple resources”
   }
 ]
}
```

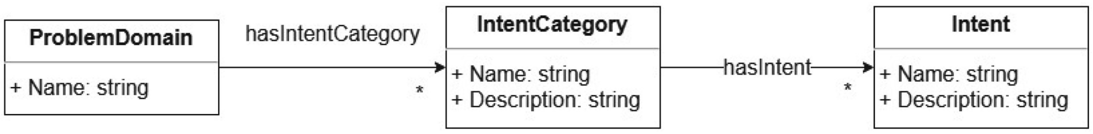

Fig. 3. Intent graph metamodel.

According to CRAC, domain concepts are modeled as instances of the Domain-Concept class, representing fundamental elements. These concepts serve as building

Table 3. Finding the intents related to a specific category (intent enumeration).

Template

Find the list of user intents in the {problem domain} problem domain related to {category} category. Provide your answer in json format, including the problemDomain, category, and list of intents (the name and description of each intent).

Prompt

Find the list of user intents in the "Infrastructure as Code" problem domain related to "Provisioning Resources" category. Provide your answer in json format, including the problemDomain, category, and list of intents (the name and description of each intent).

Answer

```
{
 "problemDomain": "Infrastructure as Code",
 "category": "Provisioning Resources",
 "intents": [
   {
    "name": "Create Virtual Machine (VM) (CreateVM)",
    "description": "Provision a virtual machine with specified configuration"
   },
   {
    "name": "Create Container Cluster (CreateCluster)",
    "description": "Provision a container cluster for deploying containerized applications"
   },
   {
    "name": "Create Load Balancer (CreateLB)",
    "description": "Provision a load balancer to distribute incoming traffic"
   },
   {
    "name": "Create Database Instance (CreateDB)",
    "description": "Provision a database instance with specified settings"
   }
 ]
}
```

blocks for the problem domain. Additionally, high-level system functions are captured as Commands. These commands are responsible for transitioning the system between valid states. For example, a "Create Virtual Machine" command triggers the provisioning process, resulting in an event that alters the system state.

Table 4. Identifying parameters required to fulfill the user’s request (parameter identification).

Template

Can you provide me with the required parameters for performing the {intent} intent in the {problem domain} problem domain related to {category} category? Provide your answer in json format, including the problemDomain, category, intent, and list of parameters (the name and description of each parameter).

Prompt

Can you provide me with the required parameters for performing the “Create Virtual Machine (VM) (CreateVM)” intent in the “Infrastructure as Code” problem domain related to “Provisioning Resources” category? Provide your answer in json format, including the problemDomain, category, intent, and list of parameters (the name and description of each parameter).

Answer

```
{
  “problemDomain”: “Infrastructure as Code”,
  “category”: “Provisioning Resources”,
  “intent”: “Create Virtual Machine (VM) (CreateVM)”,
  “parameters”: [
    {
      “name”: “vmName”,
      “description”: “The name of the virtual machine”
    },
    {
      “name”: “vmSize”,
      “description”: “The desired size or configuration for the VM”
    },
    {
      “name”: “image”,
      “description”: “The base operating system image for the VM”
    },
    {
      “name”: “network”,
      “description”: “The network configuration for the VM”
    },
    {
      “name”: “storage”,
      “description”: “The storage options for the VM”
    }
  ]
}
```

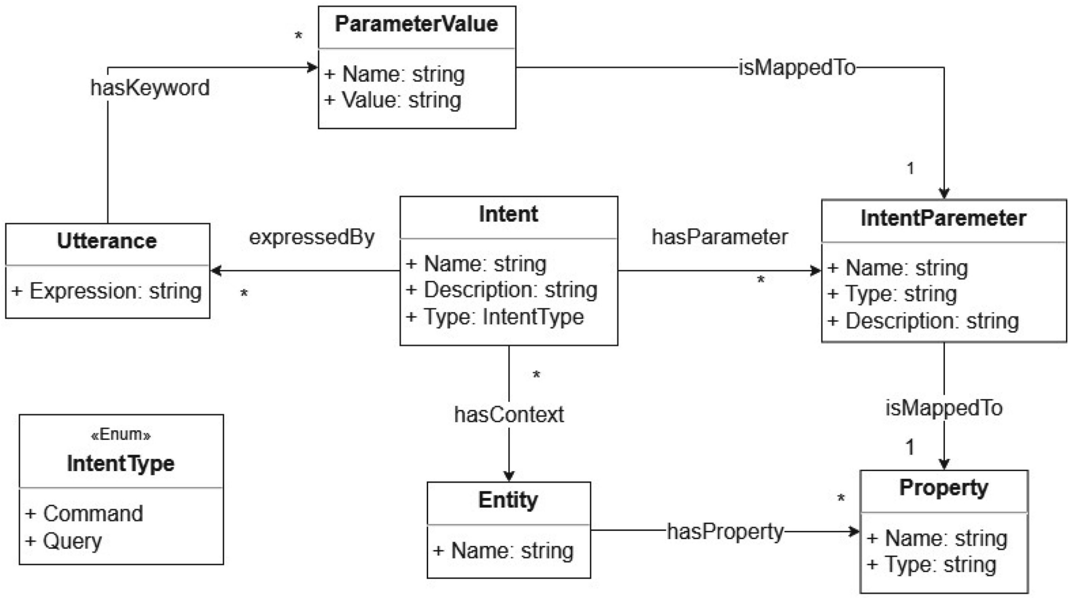

Fig. 4. Intent metamodel.

The CRAC method acknowledges that system interactions are often asynchronous. Events play a crucial role in capturing these collaborations. When a command is executed, it generates an Event that reflects the outcome of the action. These events provide a way to track system changes, notify relevant components, and maintain consistency across distributed systems.

Figure 5 shows the CRAC metamodel. To bridge the gap between user intents and the CRAC model, model-to-model transformations (M2M) is executed. These transformations map intent-related information to corresponding CRAC elements (Table 5). By aligning the intent model with the CRAC representation, we ensure context-aware responses and facilitate efficient system design.

4.2 PIM-Level Construction

In this phase, our objective is to design the chatbot microservice and establish its interactions with essential services. The platform-independent models (PIM) created during this stage remain agnostic to specific platforms and service providers.

We begin by identifying user intents and extracting the necessary information to fulfill their requests. An AI model and machine learning (ML) algorithm are typically employed for training. Tasks involve finding training phrases for each intent, identifying key parameters, and mapping them to generic or custom entities. In our previous work [12], we illustrated the approach, demonstrated prompt generation, and leveraged model-driven development techniques. By describing questions using a DSL and converting them via model-to-text transformations, we facilitate seamless integration with NLU microservices. We've introduced a metamodel to describe these questions, as depicted in Fig. 6.

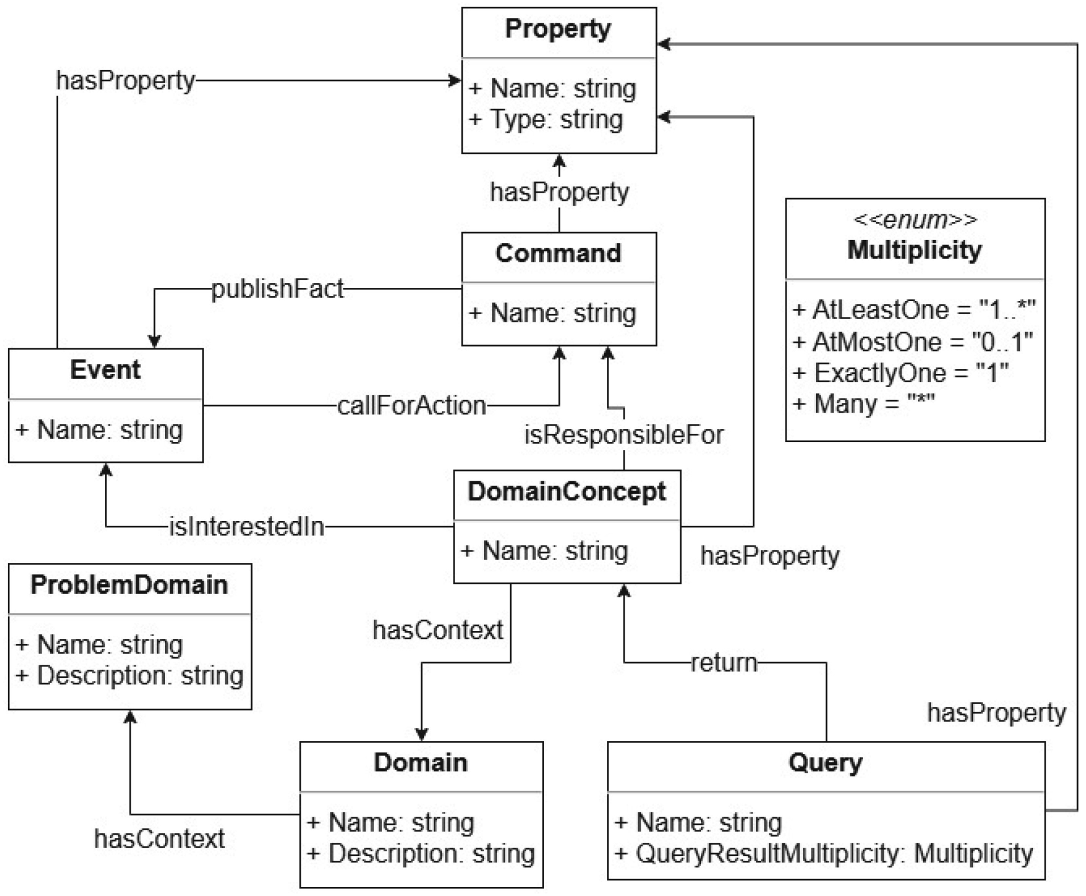

Fig. 5. CRAC metamodel.

The second activity focuses on defining the chatbot microservice architecture. We employ various architectural patterns, including the CQRS (Command Query Responsibility Segregation) and API Controller patterns. The CQRS metamodel abstractly describes design aspects related to CQRS, such as command handlers, query handlers, events, and queries. Similarly, the API Controller metamodel provides an abstract syntax for describing web request entry points and their distribution via controller patterns. Table 6 and Table 7 show transformation rules for converting the CRAC model to CQRS and API Controller models.

The third activity involves defining the bidirectional conversation flow between the chatbot microservice and the user. For each intent in the model, we outline how the chatbot interacts with users and specify actions to fulfill their requests. To achieve this, we propose a metamodel for describing the dialogue flow, ensuring effective bidirectional communication between the chatbot microservice and users, as shown in Fig. 7.

Table 5. Intent-to-CRAC model transformation rules.

Intent Metamodel	CRAC Metamodel
Entity	DomainConcept
Intent {Intent.Type = Command}	Command
Intent {Intent.Type = Query}	Query
Property	Property
IntentParameter	Property
Entity ➜ Property: hasProperty	DomainConcept➜Property: hasProperty
Intent {Intent.Type = Command}➜IntentParameter: hasParameter	Command➜Property: hasProperty
Intent {Intent.Type = Query}➜ntentParameter: hasParameter	Query➜Property: hasProperty
Intent {Intent.Type = Command}➜Entity: hasContext	DomainConcept➜Command: isResponsibleFor
Intent ➜ Entity: hasContext	Query➜DomainConcept: return

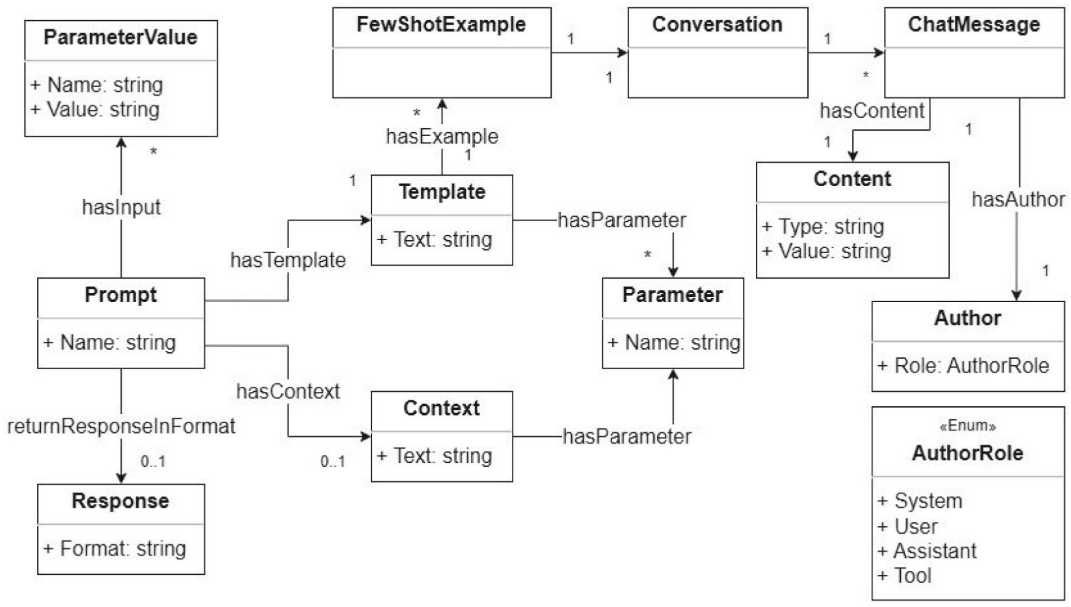

Fig. 6. Prompt metamodel.

4.3 PSM-Level Construction

At the PSM level, two metamodels have been provided for implementing solutions and generating code in the C# language within the.NET framework. The goal of the "Class & Interface" metamodel (a partial view of which is provided in Fig. 8) is to provide an

Table 6. CRAC-to-CQRS model transformation rules.

CRAC Metamodel	CQRS Metamodel
Command	CommandHandler➜Command: canHandle
Query	QueryHandler➜Query: canHandle
DomainConcept➜Event: isInterestedIn	EventHandler➜Event: canHandle

Table 7. CRAC-to-APIController model transformation rules.

CRAC Metamodel	API Controller Metamodel
DomainConcept➜Command: isResponsibleFor	APIController➜Command: dispatchCommand
Query➜DomainConcept: return	APIController➜Query: dispatchQuery

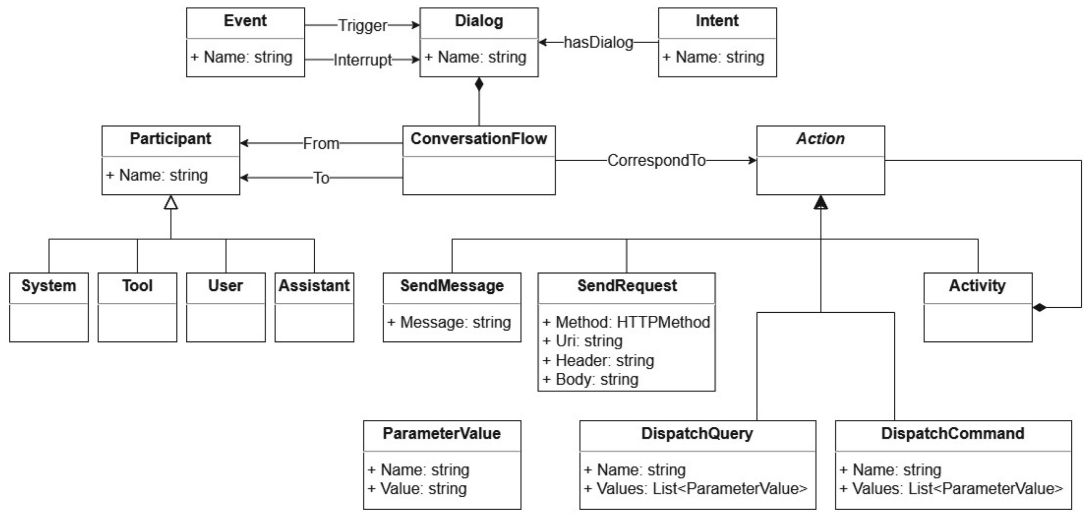

Fig. 7. Dialog metamodel.

abstract syntax that allows describing the solution domain using class and interface concepts in the C# programming language. The purpose of the "Project Settings" metamodel (Fig. 9) is to provide an abstract syntax for describing the settings and configuration of a software project in the.NET framework, including external services such as NLU and Messaging microservices.

By combining information from the Business Service Interaction Model, NLU Service, and Messaging Service Configuration Models at the PSM level with the Refined Intent Model and Dialog Model at the PIM level, we generate the Refined Dialog Model. Additionally, the CQRS and API Controller models at the PIM level are transformed into corresponding models at the PSM level using model-to-model transformation.

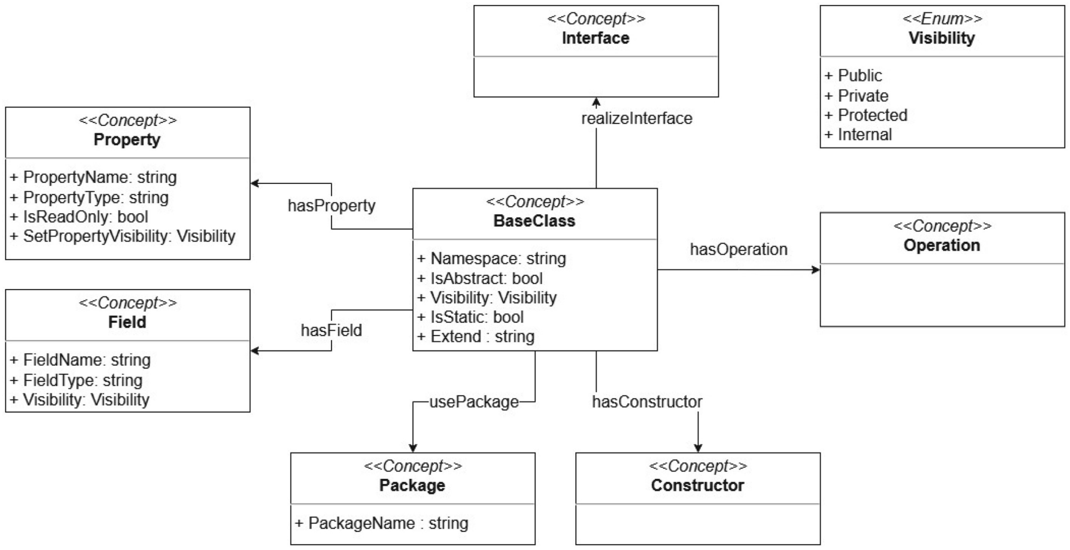

Fig. 8. Partial Class & Interface metamodel (from the 'BaseClass' perspective).

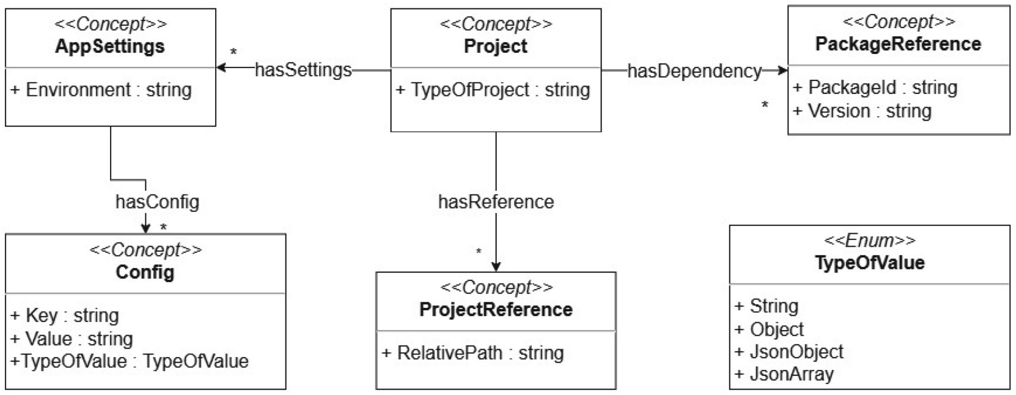

Fig. 9. Project Settings metamodel.

4.4 Code-Level Construction

At the code level, a simple metamodel called the "Project File & Folder" metamodel has been established to model project structure, folders, files, and their content, as shown in Fig. 10. Through model-to-model transformation, PSM-level models are converted into the "Solution Model," which adheres to the "Project File & Folder" metamodel. Subsequently, solution code is generated from the "Solution Model" using model-to-text transformation.

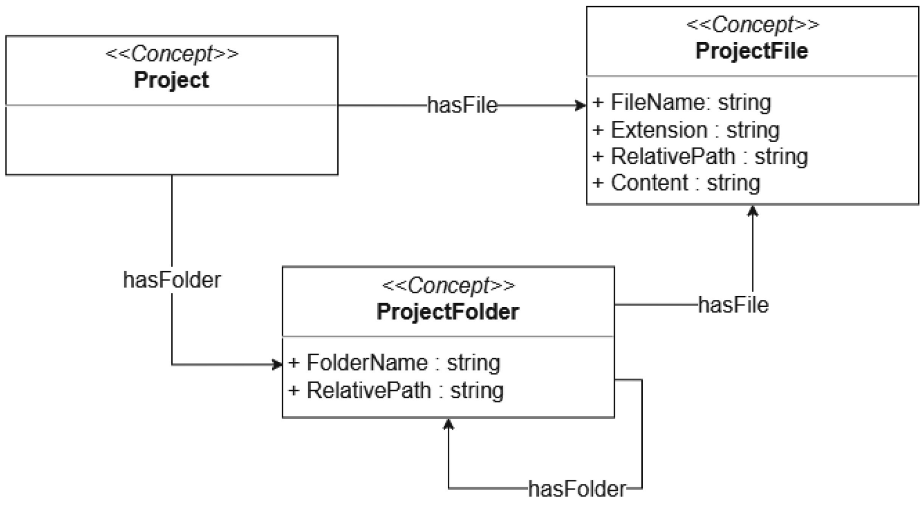

Fig. 10. Project File & Folder metamodel.

5 Evaluation

We have used a criteria-based approach to evaluate the effectiveness of our proposed methodology. The evaluation criteria fall into three distinct categories: criteria related to the generic software development lifecycle (SDLC), criteria specific to model-driven development (MDD), and criteria directly applicable to chatbots. The results of evaluating the proposed methodology based on these three categories of criteria are presented in Table 8.

For problem domain analysis, our approach fully supports understanding the domain-specific requirements, facilitating effective solution design. Regarding the generic SDLC, our methodology fully supports Requirements Engineering, Analysis, Design, and Implementation phases. However, it does not provide complete coverage of the entire lifecycle, as Test, Deployment, and Maintenance phases are not explicitly addressed. In terms of umbrella activities, our methodology partially supports cross-cutting concerns, which play a crucial role in software development.

Reusability is a key aspect, and our methodology leverages the model-driven development (MDD) approach, promoting the creation of reusable artifacts. Specifically related to MDD, our methodology supports modeling at different levels of abstraction (CIM, PIM, PSM, and Code). It also enables seamless Model-to-Model transformation across these levels and model-to-text transformation at the Code level.

In the context of chatbots, our methodology addresses domain knowledge modeling, user intent modeling, and conversation flow modeling. Additionally, it covers essential tasks for setting up Natural Language Understanding (NLU) services and architectural design, and satisfies essential quality attributes such as scalability, flexibility, and interoperability. Furthermore, it excels in handling conversation aspects, including understanding user queries and providing relevant responses.

Table 8. Criteria-based evaluation results.

Category	Criteria		Level
Generic SDLC Criteria	Coverage of Generic Lifecycle	Requirements Engineering	●
		Analysis	●
		Design	●
		Implementation	●
		Test	◐
		Deployment	○
		Maintenance	○
	Coverage of Umbrella Activities	Project Management	○
		Quality Assurance	◐
		Risk Management	◐
	Reusability		●
	Adaptability		◐
MDD-related Criteria	CIM / PIM / PSM Creation		●
	Model Transformation		●
	Metadata Management		○
	Verification & Validation		◐
	Automatic Testing		○
	Traceability between Models		◐
	Tool Support		○
Chatbot-related Criteria	Chatbot Input / Output		Text
	Domain		Closed-Domain
	Approaches		AI-based
	Knowledge Data Structures		(Semi)Structured
	Domain Knowledge Modeling		●
	User Intent Modeling		●
	Conversation Flow Modeling		●
	Training Phrase Elicitation / Annotation		●
	NLU Service Providers		Vendor-Independent
	Communication Channels		Vendor-Independent
	Architectural Design		●
	Quality Attributes	Scalability	●
		Flexibility	●
		Maintainability	●
		Interoperability	●
		Usability	◐
		Availability	◐
		Performance	◐
		Security	◐
	Conversational Aspects	Understanding	◐
		Answering	◐
		Navigation	◐
		Error handling	◐
		Relevance	◐
		Consistency	◐

Legend: Full support ● ; Partial support ◐ ; No Support ○

6 Conclusions

We have introduced a model-driven methodology for chatbot development, addressing challenges related to productivity, reusability, scalability, and maintainability. By leveraging computation-independent models (CIMs), platform-independent models (PIMs), and platform-specific models (PSMs), our approach guides developers through the chatbot creation process. We emphasize seamless transitions between these levels of abstraction, enabling efficient model-to-model and model-to-text transformations.

Our methodology not only automates code generation but also introduces a novel approach to user intent representation. By categorizing, enumerating, parameterizing, and representing user intents, we enhance the effectiveness of natural language understanding (NLU) services. Additionally, we adopt microservice architecture and architectural design patterns to improve scalability, maintainability, and interoperability. As chatbots continue to evolve, our model-driven approach offers a valuable framework for building intelligent conversational agents.

Future research will focus on providing tool support for our methodology and defining metamodels for common communication platforms and NLU services.

References

1. Planas, E., Daniel, G., Brambilla, M., Cabot, J.: Towards a model-driven approach for multiexperience AI-based user interfaces. Softw. Syst. Model. **20**(4), 997–1009 (2021). https://doi.org/10.1007/s10270-021-00904-y
2. Perez-Soler, S., Daniel, G., Cabot, J., Guerra, E., de Lara, J.: Towards automating the synthesis of chatbots for conversational model query. In: Enterprise, Business-Process and Information Systems Modeling, (pp. 257–265) (2020). https://doi.org/10.1007/978-3-030-49418-6_17
3. Perez-Soler, S., Guerra, E., de Lara, J.: Flexible modelling using conversational agents. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 478–482 (2019). https://doi.org/10.1109/MODELS-C.2019.00076
4. Matic, R., Kabiljo, M., Zivkovic, M., Cabarkapa, M.: Extensible chatbot architecture using metamodels of natural language understanding. Electronics **10**(18), 2300 (2021). https://doi.org/10.3390/electronics10182300
5. Perez-Soler, S., Guerra, E., de Lara, J.: Creating and Migrating Chatbots with Conga. 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 37–40 (2021). https://doi.org/10.1109/ICSE-Companion52605.2021.00030
6. Mahmood, R., Joshi, A., Lele, A., Pennington, J.: Dynamic Natural Language User Interfaces Using Microservices. HAI-GEN+ User2agent@ IUI (2020). https://ceur-ws.org/Vol-2848/user2agent-paper-1.pdf
7. Rodrigues da Silva, A.: Model-driven engineering: a survey supported by the unified conceptual model. Comput. Lang., Syst. Struct. **43**, 139–155 (2015). https://doi.org/10.1016/j.cl.2015.06.001
8. Alam, O., Corley, J., Masson, C., Syriani, E.: Challenges for reuse in collaborative modeling environments. MODELS Workshops, pp. 277–283 (2018)
9. Martínez-Gárate, Á.A., Aguilar-Calderón, J.A., Tripp-Barba, C., Zaldívar-Colado, A.: Model-driven approaches for conversational agents development: a systematic mapping study. IEEE Access **11**, 73088–73103 (2023). https://doi.org/10.1109/ACCESS.2023.3293849

10. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Xatkit: a multimodal low-code chatbot development framework. IEEE Access **8**, 15332–15346 (2020). https://doi.org/10.1109/ACCESS.2020.2966919
11. Ed-douibi, H., Cánovas Izquierdo, J.L., Daniel, G., Cabot, J.:. A model-based chatbot generation approach to converse with open data sources. In: Web Engineering (Vol. 12706, pp. 440–455) (2021). https://doi.org/10.1007/978-3-030-74296-6_33
12. Vahdati, A., Ramsin, R.: "Model-driven methodology for developing chatbots based on microservice architecture". In: Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD'24), 2024, pp. 247–254 (2024)