Highlights

Applying model-driven engineering to the domain of chatbots: the Xatkit experience

Gwendal Daniel, Jordi Cabot

- Model-driven engineering can be successfully applied to new domains such as AI-based software
- Model-driven tools are suitable for prototype creation but have limitations when it comes to developing industrial-strength solutions
- Commercial success of model-driven based approaches depends on numerous factors beyond technical ones

Applying model-driven engineering to the domain of chatbots: the Xatkit experience

Gwendal Daniel[a], Jordi Cabot[b]

[a] *Universitat Oberta de Catalunya (UOC), Avinguda Carl Friedrich Gauss, 5, Castelldefels, 08860, Spain*
[b] *Luxembourg Institute of Science and Technology, 5, Avenue des Hauts-Fourneaux, Esch-sur-Alzette, 4362, Luxembourg*

Abstract

Chatbots are becoming a common component of many types of software systems. But they are typically developed as a side feature using ad-hoc tools and custom integrations. Moreover, current frameworks are efficient only when designing simple chatbot applications while they still require advanced technical knowledge to define complex interactions and are difficult to evolve along with the company needs. In addition, the deployment of a chatbot application usually requires a deep understanding of the targeted platforms, especially back-end connections, increasing the development and maintenance costs.

In this paper, we discuss our experiences building, evolving and distributing the Xatkit framework. Xatkit is a model-based framework built around a Domain-Specific Language to define chatbots (and voicebots and bots in general) in a platform-independent way. Xatkit also comes with a runtime engine that automatically deploys the chatbot application and manages the defined conversation logic over the platforms of choice.

Xatkit has significantly evolved since its initial release. This paper focuses on describing the evolution and the reasons (technical and non-technical) that triggered them. We believe our lessons learned can be useful to any other initiative trying to build a successful industrial-level chatbot platform, and in general, any type of model-based solution.

1. Introduction

The specification and the implementation of the User Interface (UI) of a system is a key aspect in many software development projects. Often, this

UI takes the form of a Graphical User Interface (GUIs) that encompasses a number of visual components [1] to offer rich interactions between the user and the system. But nowadays, a new generation of UIs which integrate more interaction modalities (such as chat, voice and gesture) is gaining popularity. Moreover, many of these new UIs are becoming complex software artifacts themselves, for instance, through AI-enhanced software components that enable even more natural interactions, including the possibility to use Natural Language Processing (NLP) via chatbots or voicebots. These NLP-based interfaces are commonly referred to as Conversational User Interfaces (CUIs).

There is no doubt that CUIs are becoming more and more popular every day. The most relevant example is the rise of bots [2], or the recent release of a ChatGPT[1], a novel AI-based chatbot which attracted significant public attraction. Indeed, chatbots and other types of CUIs have proven useful in various contexts to automate tasks and improve the user experience, such as automated customer services [3], education [4], e-commerce [5] and even software development [6].

This widespread interest and demand for chatbot, and in general CUI, applications has emphasized the need to be able to quickly build complex chatbot applications. This means that chatbots should go beyond simply answering predefined text in response to user requests. Instead, any non-trivial chatbot requires accessing an orchestration [7] of internal and external services to perform the requested user actions (e.g. to check and query the data to be served back to the user or to actually execute some processes/actions in response). As such, chatbots are becoming complex software artifacts that require a more methodical development approach to be developed with the proper quality standards and expertise in a variety of technical domains, ranging from natural language processing to an in-depth understanding of the APIs of the targeted instant messaging platforms and third-party services to be integrated.

So far, chatbot development platforms have mainly addressed the first challenge, typically by relying on external *intent recognition providers*, which are natural language (NL) processing frameworks providing user-friendly interfaces to define conversation assets. As a trade-off, chatbot applications are tightly coupled to their *intent recognition providers*, hampering their maintainability, reusability and evolution. Typically, once the chatbot designer

[1]https://openai.com/blog/chatgpt

chooses a specific chatbot development platform, she ends up in a vendor lock-in scenario, especially with the NL engine coupled with the platform. Similarly, current chatbot platforms lack proper abstraction mechanisms to easily integrate and communicate with other external platforms the company may need to interact with.

We believe the right way to tackle all these issues is by raising the level of abstraction at what chatbots (and any other types of bots) are defined. To this purpose, we created Xatkit [8], a novel model-based chatbot development framework providing domain-specific languages for the platform-independent definition of chatbots plus a runtime engine able to execute such bot definitions. Indeed, Xatkit embeds a dedicated chatbot-specific modeling language to specify user intentions, computable actions and callable services, combining them in rich conversation flows. Conversations can either be started by a user awakening Xatkit or by an external event that prompts a reaction from Xatkit (e.g. alerting a user that some event of interest fired on an external service the bot is subscribed to).

The resulting chatbot definition[2] is independent of the intent recognition provider (which can be configured as part of the available Xatkit options). Moreover, it frees the designer from the technical complexities of dealing with messaging and backend platforms, as Xatkit can be deployed through the Xatkit runtime component on them without performing any additional steps.

This paper follows up on the original Xatkit presentation and discusses the evolution of the platform and all the lessons learned after the application of our model-based chatbot infrastructure on industrial strength projects. We believe many of these reflections would be interesting for the model-based engineering community at large, especially for practitioners and researchers who are developing a professional MDE product, even if they are not targeting the chatbot domain.

The rest of the paper is structured as follows. The next section introduces some preliminary concepts. Then, Section 3 recaps the initial Xatkit model-based components, while Section 5 describes the key technical evolution departing from that original version based on we experience we gained building non-trivial chatbots. Section 6 follows up with additional evolutions

[2]In this text, we do not distinguish between bots, chatbots, and voicebots, since Xatkit supports all of them via its set of supported platforms

focused on aligning better Xatkit to the commercial needs we detected. Finally, Section 7 comments on the related work and Section 8 concludes the work.

2. Preliminary concepts

Bots are classified into different types depending on the channel employed to communicate with the user. For instance, in *chat*bots the user interaction is through textual messages, in *voice*bots it is through speech, while in *gesture*bots it is through interactive images. Note that in all cases, bots are the mechanism to implement a conversation, it just changes the medium where this conversation takes place. As such, bots always follow the common working pattern depicted in Figure 1.

As you can see in the figure, the conversation capabilities of a bot are usually designed as a set of *intents*[3], where each intent represents a possible user's goal when interacting with the bot. The bot then waits for its CUI front-end to detect a match of the user's input text (called *utterance*) with one of the intents the bot implements. The matching phase may rely on external Intent Recognition Providers (e.g. DialogFlow, Amazon Lex, IBM Watson, ...). When there is a match, the bot back-end executes a set of *actions* implementing the required behavior. These actions may represent simple responses such as sending a message back to the user, as well as advanced features required by complex chatbots like database querying or external service calling. Finally, we define a *conversation path* as a particular sequence of received user *intentions* and associated *actions* (including non-messaging actions) that can be executed by the chatbot application.

As an example, we show in Figure 2 a bot that gives the weather forecast for any city in the world. Following the working schema sketched in Figure 1, this weather bot defines several intents, such as *asking for the weather forecast*. When a user writes (considering a chatbot) or says (considering a voicebot) *"What is the weather today in Barcelona?"* or *"What is the forecast for today in Barcelona?"*, the intent *asking for the weather forecast* is matched and "Barcelona" is recognized as a city parameter (also called "entity") to be used when building the response. Then, the bot calls an

[3]We will comment more on open-domain bots, relying on a pretrained language model to have general conversations, in later sections

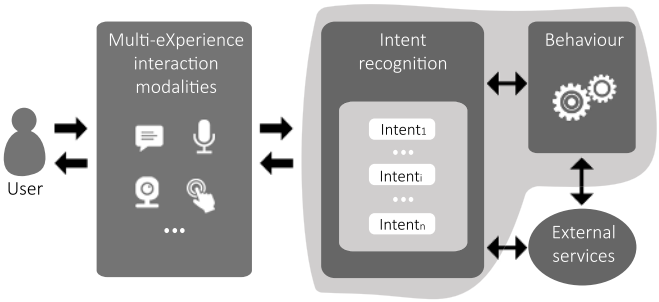

Figure 1: Common Bot working pattern.

external service (in this case, the REST API of OpenWeather[4]) to look up this information and give it back to the user.

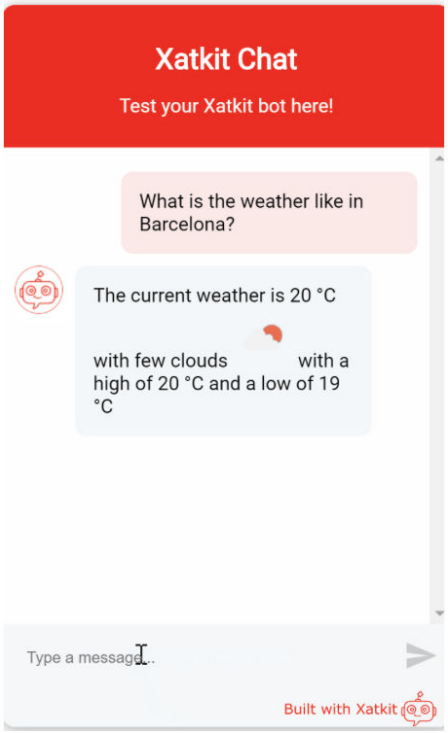

Figure 2: Screenshot of our example chatbot

[4]https://openweathermap.org

3. Xatkit: a model-driven engineering (MDE) solution for chatbots

These new types of UIs are hard to specify [9], test, verify and debug [10], and require a different and specialized skill set [11]. Therefore, we believe they could benefit from applying a model-based approach to reduce the complexity of all elements, and the respective technology stacks, involved in their creation.

This is why we started Xatkit in 2018. The following subsections cover the key points of the initial Xatkit components (see [8, 12] for a deeper explanation) while we cover later on how Xatkit has evolved as part of the maturity process (technical and commercial) Xatkit has undergone more recently.

Xatkit is an open-source tool freely available on GitHub[5].

3.1. Xatkit architecture

Figure 3 presents an overview of our MDE-based chatbot approach and its main components. At design time, the chatbot designer specifies the chatbot under construction using two domain-specific languages (DSLs) that are part of **Xatkit Modeling Infrastructure**:

- **Intent Package** to describe the user *intentions* using training sentences, contextual information extraction, and matching conditions.
- **Execution Package** to bind user *intentions* to response *actions* as part of the chatbot behavior definition.

The actions in the Execution part of the bot often involve a set of orchestrated calls to services provided by the available *Platforms*. Platforms are defined by a Platform designer via a separate **Platform Package** and, once available, are enabled for all existing bots. Platforms are organized in a taxonomy so the chatbot designer can choose generic actions (e.g. a textual reply, something available in all chat-based platforms) or more specific ones (e.g. attaching a file to a message, only available in some specific platforms like Slack). The resulting platform definition hides all the technical details of the communication with the platforms.

These models are complemented with a *Deployment Configuration* file that specifies the *Intent Recognition Provider* to use (e.g Google's DialogFlow [13] or IBM Watson Assistant [14]), platform specific configuration parameters

[5] `https://github.com/xatkit-bot-platform`

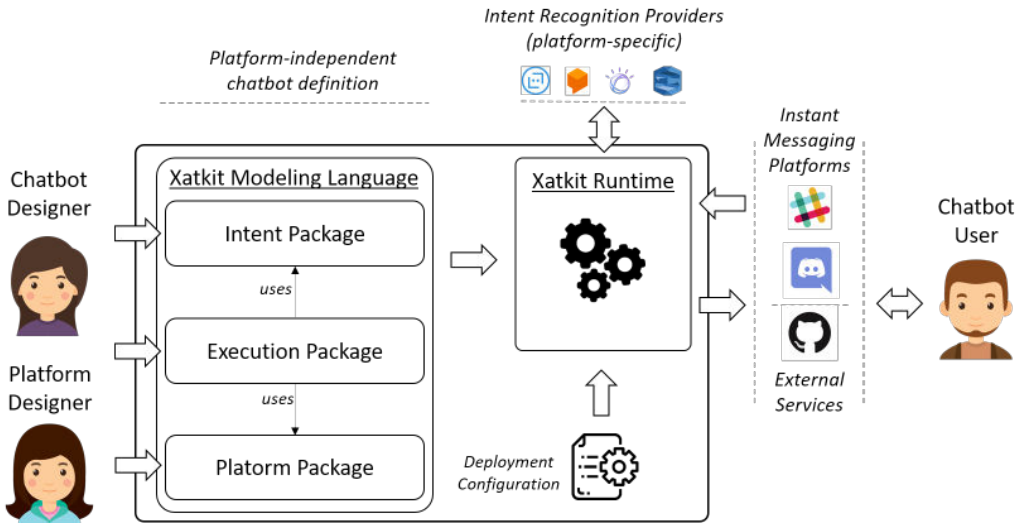

Figure 3: Xatkit Framework Overview

(e.g. OAuth credentials), as well as custom execution properties, which for instance can introduce some limited variability in the bot behavior. Note that in the Xatkit infrastructure, all the *intent recognition providers* implement a common interface that allows switching from one to another transparently through configuration properties. Support for new providers can be easily achieved by implementing this common interface.

These assets constitute the input of the **Xatkit Runtime** component that starts by deploying the created chatbot. This implies registering the user *intents* to the selected *Intent Recognition Provider*(which involves translating the intents in the bot definition into the primitives/mechanisms available in that specific provider), connecting to the *Instant Messaging Platforms*, and starting the *External Services* specified in the *execution* model. Then, when a user input is received, the runtime forwards it to the *Intent Recognition Provider*, gets back the recognized intent and performs the required action associated to that intent based on the chatbot *execution* model.

This infrastructure provides three main benefits:

- The *Xatkit Modeling Language* packages decouple the different dimensions of a chatbot definition, facilitating the reuse of each dimension across several chatbots.

- Each sublanguage is totally independent of the concrete deployment

and intent recognition platforms, easing the maintenance and evolution of the chatbot.

- The *Runtime* architecture can be easily extended to support new platform connections and computable actions. This aspect, coupled with the high modularity of the language, fosters new contributions and extensions of the framework.

The next subsection focuses on the intent definition and execution sublanguages as the key model-based components of the infrastructure and those that have been evolving the most over time, as we will explain in sections 5 and 6. For more details on the infrastructure for the runtime execution and deployment, we refer the readers to [8].

3.2. Xatkit modeling languages

In the following, we introduce the Xatkit Modeling Language, composed of a set of interrelated chatbot Domain-Specific Languages (DSL) that provides primitives to design the user intentions and the corresponding execution logic.

The Xatkit language is defined through two main components [15]: (i) an abstract syntax (metamodel) defining the language concepts and their relationships (generalizing the primitives provided by the major intent recognition platforms [13, 14, 16]), and (ii) a concrete syntax in the form of a textual notation to write chatbot descriptions conforming to the abstract syntax. In the following we use the abstract syntax to describe the DSL packages and primitives, and the textual to show, via examples based on our running example, how those concepts can be used to create bots.

To decouple the definition of the user intentions the chatbot should recognize from the actions the chatbot should execute in response to those intents, our language is split up into two different sublanguages: the *Intent* and the *Execution* packages.

3.2.1. Intent Package

Figure 4 presents the metamodel of the *Intent Package*, that defines a top-level *IntentLibrary* class containing a collection of *IntentDefinitions*. An *IntentDefinition* is a *named* entity representing a user intention. It contains a set of *Training Sentences*, which are input examples used to detect the user intention underlying a textual message. *Training Sentences* are split

into *TrainingSentenceParts* representing input text fragments — typically words — to match.

Each *IntentDefinition* defines a set of *outContexts*, that are named containers used to persist information along the conversation and customize intent recognition. A *Context* embeds a set of *ContextParameters* which define a mapping from *TrainingSentenceParts* to specific *EntityDefinitions*, specifying which parts of the *TrainingSentences* contain information to extract and store. *EntityDefinitions* can be either *BaseEntityDefinitions*, i.e. generic entities that are provided for all the intent recognition platforms such as *city* or *date*, or *MappingEntityDefinitions* that represent user-designed entities represented by a value and a list of synonyms.

IntentDefinitions can also reference *inContexts* that are used to specify matching conditions. An *IntentDefinition* can only be matched if its referenced *inContexts* have been previously set, i.e. if another *IntentDefinition* defining them as its *outContexts* has been matched, and if these *Contexts* are active with respect to their *lifespans*. Finally, the *follow* association defines *IntentDefinition* matching precedence, and can be coupled with *inContext* conditions to finely describe complex conversation paths.

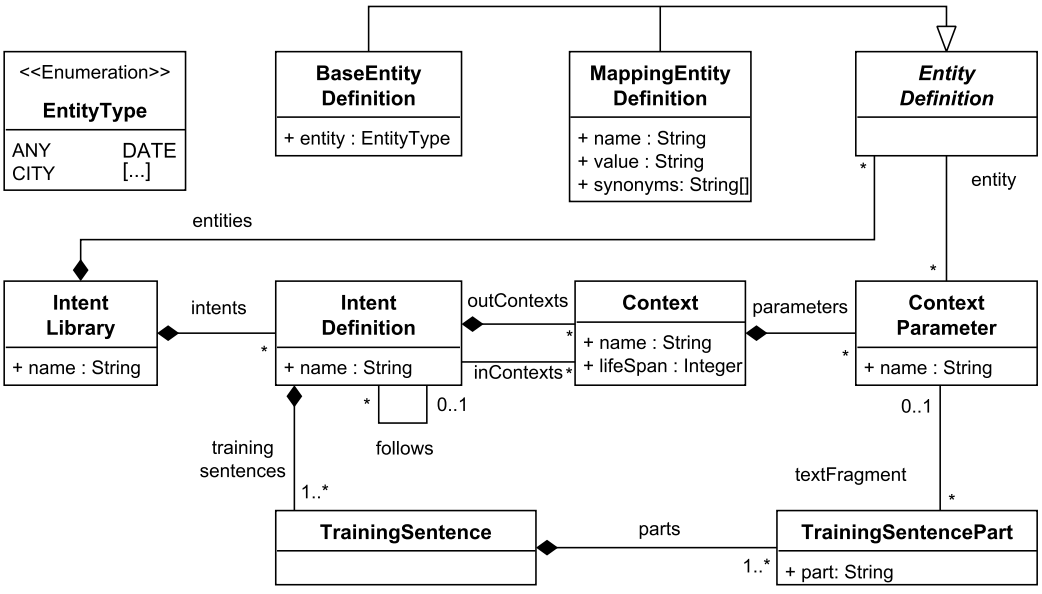

Figure 4: Intent Package Metamodel

Listing 1 shows a (partial) instance of the *Intent Package* from the Weather

Bot running example introduced in Section 2.

The model defines the *IntentLibrary* `Example`, that contains one *IntentDefinitions* called `HowIsTheWeather` that does not depend on any other intent, and it simply includes a couple of training sentences specifying alternative inputs used to ask for the weather. It also defines an output context *Weather* to collect the context parameter `cityName` for which the user is asking the weather information. The parameter uses the `city` *BaseEntityDefinition* for their value extraction that will match with any recognized city name. Alternatively, if we wanted to restrict the match to a specific number of cities supported by the bot we could have created our own *MappingEntityDefinition* with just the cities of interest.

Listing 1: Example Intents for the WeatherBot

```
1  library Example
2
3  intent HowIsTheWeather {
4    inputs {
5      "How is the weather today in XXX?"
6      "What is the forecast for today in XXX?"
7    }
8    creates context Weather {
9      sets parameter "cityName" from fragment "XXX" (entity city)
10   }
11 }
```

3.2.2. Execution Package

The *Execution Package* (Figure 5) is an event-based language that represents the chatbot execution logic.

An *ExecutionModel imports Platforms*[6] and *IntentLibraries*, and specifies the *ProviderDefinitions* used to receive user inputs and events. The *ExecutionRule* class is the cornerstone of the language, which defines the mapping between received *IntentDefinitions*/*EventDefinitions* and *Actions* to compute.

The *Action* class represents the reification of a *Platform ActionDefinition* with concrete *ParameterValues* bound to its *Parameter* definitions. These Actions are part of the definition of the Platform. The *value* of a *ParameterValue* is represented as an *Expression* instance. Xatkit *Execution* language

[6]The complete definition of the platform package as well as a taxonomy of concrete platforms implementing it has been detailed in our previous work [8]

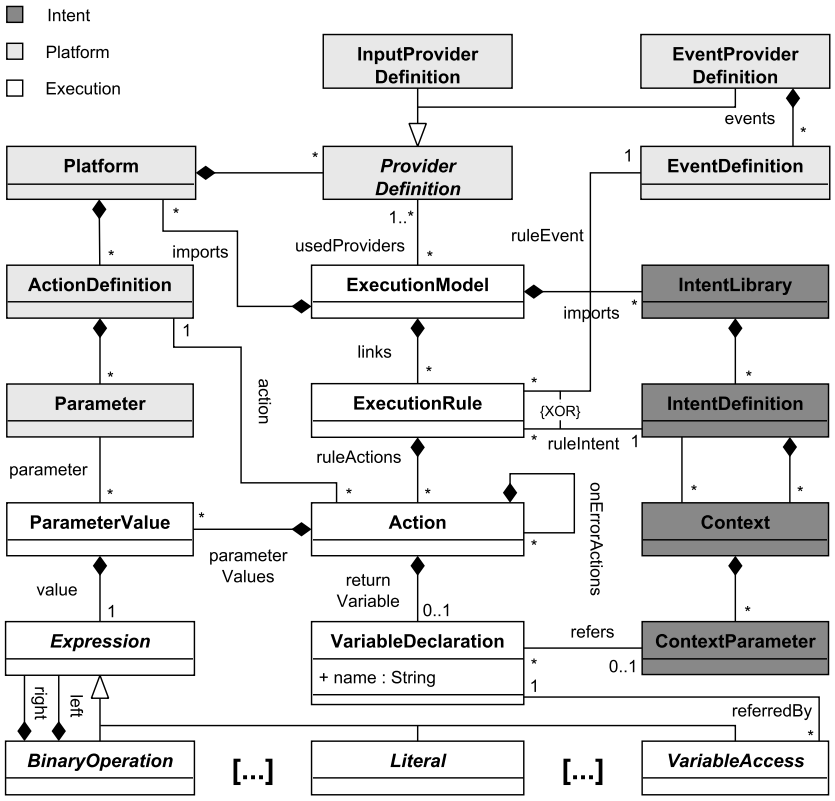

Figure 5: Execution Package Metamodel

currently supports *Literals*, *Unary* and *Binary Operations*, as well as *VariableAccesses* that are read-only operations used to access *ContextParameters*.

An *Action* can also define an optional *returnVariable* that represents the result of its computation, and can be accessed from other *Actions* through *VariableAccess Expressions*, allowing to propagate information between computed actions.

Listing 2 shows an excerpt of the *Execution* model from our running example. It *imports* the `Weather` and the, predefined, `Core` *IntentLibrary* and the `Rest` and `React` *Platforms*. The `Rest` one will be used to process the request via an API call to the `OpenWeatherMap` API, the latter to get the input text from the user from the chatbot widget embedded on a webpage. With the API response information, it builds a response, sends it back to the user via the `React` platform and gets back to the `Init` state waiting for the next question.

Listing 2: Chatbot Execution Language Example

```
1
2  import library "WeatherBot/src/WeatherBot.intent" as WeatherBotLib
3  import library "CoreLibrary"
4  import platform "RestPlatform"
5  import platform "ReactPlatform"
6
7  use provider ReactPlatform.ReactIntentProvider
8
9  Init {
10   Next {
11     intent == HowIsTheWeather --> HandleHowIsTheWeather
12   }
13 }
14
15 HandleHowIsTheWeather {
16   Body {
17     val city = context.get("Weather").get("cityName") as String
18     val queryParamters = newHashMap
19     queryParamters.put("q", city)
20     val response = RestPlatform.GetJsonRequest("http://api.openweathermap.
           org/data/2.5/weather", queryParamters, emptyMap, emptyMap)
21     if(response.status === 200) {
22
23                 // Processing of the JSON of the response and extracting a
                       temp value (among others)
24                 ReactPlatform.Reply("The current temperature is  "+temp}
25
26               }
27         }
28   Next {
29     _ --> Init
30   }
31 }
```

4. Xatkit as a commercial open-source solution

Xatkit follows a commercial open-source model where the platform is released as an open source while the company built around the platform pursues a commercial exploitation based on the development and commercialization of bots for specific domains and markets.

4.1. Xatkit as an open-source platform

Xatkit was initially released as an open-source project in 2018. Since then, it has grown to become a fully-fledged GitHub organization with around 2000 commits and over 80 different repositories (counting the private ones) covering the core of the framework but also all the connectors with external platforms such as messaging applications (e.g. Slack), NLU engines (e.g.

DialogFlow) and logging and monitoring components (e.g. to store bots log data on different types of databases).

As part of this growth, several other people have contributed to the development of the platform, including bachelor and master students. For instance:

- A voice connector to Alexa was built by a master student from Politecnico di Milano (Italy)[7]
- A bachelor student from the Technical University of Catalonia (Spain) worked on the preprocessors to detect toxic texts in a conversation[8] while another one from this same university took care of implementing a monitoring component on top of InfluxDB[9]
- Two master students from Universidad de la República (Uruguay) contributed the Xatkit Twitter platform
- The Facebook connector was developed by a team of three master students from the University of Tartu (Estonia)
- There are currently two master students from HVL (Norway) working on the automatic derivation of bots from API definitions

There was even a high school student that used Xatkit to build an anti-bullying bot. Students would use the bot to answer questions anonymously about their friends and their feelings and the bot would then be able to signal there was a potential bullying situation in the class [10]. Some of these contributions became part of the "official" components / repositories, while others remained under the *lab* brand to highlight that they can be used but are not maintained by Xatkit.

Xatkit has also been used by other researchers in their work. For instance, [17] proposes a Xatkit bot to help in the definition of smart contracts, while [18] creates a Xatkit bot for educational purposes.

[7] `https://www.politesi.polimi.it/handle/10589/152764`

[8] `https://upcommons.upc.edu/handle/2117/351418`

[9] `https://upcommons.upc.edu/handle/2117/329235`

[10] The bot made the news in the Spanish newspapers and the author even wrote a book, titled *Yo también soy diferente: Un libro contra el bullying* where she explains her own experience

We believe all these contributions and adoptions by external people show that Xatkit is a platform easy to extend and use, as even students from different backgrounds and countries and with no prior experience with Xatkit or chatbot development were able to contribute to the platform even under time constraints (most works had to be completed in one semester while the students were also taking other courses).

Note that, separately, we also conducted a usability study on Xatkit that we describe in detail here [8]. In short, the study showed that Xatkit was evaluated very positively in a number of categories, ranging from the overall experience with Xatkit, the usability of Xatkit's modeling language, the power of the platform abstraction mechanism, the benefits of defining chatbots at a higher-abstraction level and how this helps to the portability between messaging platforms.

4.2. Xatkit as a company and its first deployed bots in real systems

In 2020, we incorporated the Xatkit company, including as shareholders the two authors of this article together with two research institutions we were affiliated with. While the company itself closed down in 2022 due to both main founders leaving Spain to pursue other career opportunities incompatible with their role as main shareholders in the company, we are still open to commercial opportunities as part of development contracts with our new affiliations.

Already in 2019, Xatkit got seed funding (80K euros) from the Catalan Government to mature the technology and perform a market analysis of the chatbot domain. As part of this market analysis, we conducted over 30 interviews with companies from different sectors (education, pharma, ecommerce, ...). Based on this, we decided Xatkit could be a viable company and even participated and won a local startup competition (the "spinUOC"[11]).

We then worked and deployed several commercial bots for internal and external clients. Among them:

- A bot to evaluate the financial health of startups based on a set of standard questions about their finances and projections.
- A FAQ-like bot for *Barcelona Activa* able to answer questions around the courses Barcelona Activa offers. The bot had to be able to understand questions in both Catalan and Spanish.

[11] `https://hubbik.uoc.edu/en/programmes/spinuoc-2023`

- An eCommerce bot that would automatically read and understand the catalog of products of an eCommerce site (in particular, an eCommerce site built with WooCommerce, the most popular eCommerce extension for WordPress sites) and answer questions about them. We mention again this specific bot in Section 6.

besides other short-term prototypes and experiments (e.g. our project to use chatbots to let citizens talk with open data sources[19]) that were not finally deployed in production systems.

All these industrial experiences (including all the feedback from the open-source users) are the foundation of the Xatkit evolution we describe in what follows.

5. Technical-driven evolution

The next subsections highlight a few technical evolutions and trade-offs we had to perform after the initial release, once we started using Xatkit to build more than "toy bots" and we began to realize some limitations on our initial design decisions. Such initial decisions were mostly driven by our previous experience with other existing modeling frameworks and our knowledge on how they were internally built. In this sense, we are confident some of these evolutions and reflections could be useful recommendations for future MDE tool builders as informed recommendations that could influence their own design trade-offs. Still, these recommendations are based on our own experience, so we do not claim they are universal nor they are immediately generalizable to any other MDE development or commercialization scenario.

5.1. Moving to an Internal DSL to avoid reinventing the wheel

While a pure stand-alone MDE infrastructure provided us a powerful toolkit to implement a first version of Xatkit, we quickly realized that the MDE toolkits and language workbenches we used were lacking seamless integration capabilities with more traditional software development techniques, languages and libraries.

As a consequence, evolving all the modeling artifacts required to improve the expressiveness of our DSL (e.g. adding new primitives to better support a new use case) was quite complex and time-consuming. And we had the feeling we were starting to reinvent the wheel. Indeed, once you start adding conditionals, iterators and other basic language constructs to your DSL when

you could get them for free in other languages, it is time to rethink your choices. This is not obviously the case for some DSLs that can stick to core domain-specific concepts with no need to include operational concepts that overlap with those already present in other languages. But it is the case of chatbots and many other DSLs that need to cover behavioral aspects.

Moreover, our high-level external DSL was not a relevant selling point for our customers (see Section 6), so we decided to switch to an internal DSL based on Java. The switch to a Java DSL made our development process way more efficient: we removed some simpler (and insufficient) abstractions initially designed to cover simple chatbots, and we redefined the chatbot behavioral language as a Java-based state machine framework expressive enough to handle all our use cases. The revamped engine was still based on state-of-the-art modelling tools (e.g. EMF), but the level of abstraction was a better fit for our skill set, both as chatbot developers but also Java developers.

The Internal DSL was implemented as Fluent Interface[12] to facilitate as much as possible the bot definition process. For instance, the use of a Fluent Interface facilitates concentrating all the bot definition in a single place, isolating as much as possible chatbot specification from other parts of the Java program while, at the same time, enabling the call to Java libraries when needed. As an example, the following listings show the same Weather bot specified in Section 3 with our current internal DSL proposal.

Listing 3: Example Intents for the WeatherBot (Internal DSL version)

```
1
2  val howIsTheWeather = intent("HowIsTheWeather")
3            .trainingSentence("How is the weather today in CITY?")
4            .trainingSentence("What is the forecast for today in CITY?")
5            .parameter("cityName").fromFragment("CITY").entity(city());
```

Listing 4: Example for the WeatherBot execution (Internal DSL version)

```
1
2      init
3          .next()
4          .when(intentIs(howIsTheWeather)).moveTo(printWeather);
5
6      printWeather
7          .body(context -> {
8              String cityName = (String) context.getIntent().getValue("cityName");
9              Map<String, Object> queryParameters = new HashMap<>();
10             queryParameters.put("q", cityName);
```

[12] https://en.wikipedia.org/wiki/Fluent_interface

```
11              ApiResponse<JsonElement> response =
12                restPlatform.getJsonRequest(context, "http://api" +
13                   ".openweathermap.org/data/2.5/weather", queryParameters, ...);
14              if (response.getStatus() == 200) {
15                 // Processing of the JSON response
16                  reactPlatform.reply(context,
17                  MessageFormat.format("The current temperature is {0}, temp));
18              }
19          })
20          .next()
21          .moveTo(awaitingInput);
```

The listings resemble those we had presented before but now we can fully leverage the power and libraries of Java, e.g. when calling the external API without the need to create ourselves the extension to our DSL required to call and process REST API calls.

5.2. Try to get your DSL primitives right as soon as possible

As commented above, evolving your DSL is expensive. Not only when you radically move from an external to an internal one but also when you evolve it to add or modify the core metamodel primitives of the language (e.g. if you are building your infrastructure on top of EMF, you will need to update the Ecore model, regenerate the artifacts,...).

An example is our evolution from tree-based bots to graph-based ones (implemented via a state machine) as the tree-based approach was simpler but we soon realized it was too limited. We have also been adding quite a few attributes to our metamodel classes that we did not think about at the beginning[13].

And there are more evolutions that we would like to perform but that we do not dare to address as they affect a larger number of classes. For instance, we would like to have a better support for uncertainty modeling [20, 21] as chatbot behavior is full of uncertainty (in the matching, in the recognition of the parameters, regarding the quality of the response) and we would like to offer to chatbot designers better ways to specify with more detail how the

[13] And here we ended up cheating a little bit as we were doing some of these modifications directly in the EMF generated classes instead of doing it the proper way, i.e. updating the Ecore model and regenerating the classes every time. While this is not the right way to do it, we opted to follow a more pragmatic approach to partially reduce the cost of evolving the DSL

bot should behave depending on all these uncertainties[14]. This could even enable some future self-learning strategies based on the user tolerance to the bot confidence levels.

Our recommendation here is to spend some time trying to build a variety of examples before setting too much into the abstract syntax for your DSL to minimize as much as possible the need for an immediate evolution. It will need to evolve, for sure, but try at least to start with a solid and validated base.

5.3. Balancing a model-based perspective with optimized hand coded components

The CUI landscape is a vibrant ecosystem providing a continuous stream of new technologies, tools, and approaches to improve conversational agents. We designed Xatkit as an extensible framework to integrate these new solutions as fast as possible with a limited effort.

Notably, Xatkit *processors* are independent libraries that can be plugged into a bot to extend its capabilities. Xatkit's execution engine provides hooks to integrate these processors at various places in the bot life-cycle. *Pre-processors* adapt the user input, for example, by filtering out some words we don't want to send to the NLP engine. *Post-processors* compute additional information on top of the NLP engine's result, for example sentiment analysis or emoji interpretation. These processors are based on embedded libraries like StanfordNLP[22], or remote APIs like Hugging Face[15].

As everything in Xatkit, all processors follow unified interfaces that enable a plug and play approach depending on the needs of a specific project. In the project configuration properties, you can define the type of processor you need and the concrete implementation you would like to use.

While we always keep this platform-independent perspective to be able to reuse existing solutions, we also had to develop our own NLP components completely tailored to what we needed on some projects. The most recent example is the Xatkit NLU Server[16], a custom NLP engine that can be fully configured for specific domains.

[14] Right now we just have a confidence level for the intent match as the only condition that can be evaluated as part of a bot state transition

[15] https://huggingface.co/

[16] `https://github.com/xatkit-bot-platform/xatkit-nlu-server`

The goal here is to find the balance between just staying at the modeling level and then simply deploying your solution on top of existing components or getting "your hands dirty" and building some components yourself on top of which deploy your bots.

Both options are not contradictory as long as the bot language still remains independent of the low-level solution, even your own, so that clients are free to choose to deploy your bot on top of other third-party solutions. Make sure you avoid the temptation to take the easy route and hard-code your own integration directly to the modeling framework (faster to implement as then you do not need to respect all interfaces external components need to follow, but it would break the platform independent philosophy). For instance, to develop our own NLP Server we implemented the same connectors and interfaces we did when integrating DialogFlow or nlp.js even if it would have been faster to skip these decoupling layers and build a direct integration able to directly access the internal runtime components. Thanks to this, now you can switch any existing bot to use our NLP Server if needed just by changing the configuration properties.

6. Commercial-driven evolution

This section covers additional evolutions and reflections that were more driven by our "exposure" to clients interested in adding a chatbot to their businesses. By clients, we mean both non-paying users and paying ones even though, as a company, we were targeting the latter. We use the term commercial to refer to both, evolution required to align better with client requirements and evolution required to evolve the product to reach an industrial strength quality level.

6.1. Platform independence is not always a selling point

The concept of platform-independence is at the core of many of the MDE tools and often highlighted as one of the key benefits of following a model-driven approach. However, it turns out that this was not really an important selling point for any of the clients we discussed with. In our opinion, platform-independence becomes more relevant once a domain reaches a certain level of maturity and companies have more technical options to choose from, and they have already experimented with some of them, potentially already having suffered as well the pain of migrating from one to another.

When pitching Xatkit, we were always emphasizing the benefits of modeling bots and then choosing where to deploy them. And that this would also facilitate their migration from one platform or NLP provider to another.

But our clients were never genuinely interested in this[17], they just wanted a solution for today, and whether this solution would be easier to migrate in 5-years time was not really an important aspect of their decision. It turned out that being able to adapt the look and feel of the chatbot widget to match the corporate color palette was much more important for them.

MDE helped us for sure to adapt to the current technology stack and infrastructure each client wanted (one wanted to use DialogFlow while the other was more into a complete on-premises solution due to data protection constraints and therefore was more interested in using nlp.js as local engine). MDE was a good paradigm to build bots faster ourselves but, in our experience, clients do not need/want to know how we build their solution as this is the pain point we are having today. Possibly, a reason was also that for most of them, ours was one of the first chatbots they requested and therefore they lack the experience to foresee future requirements.

6.2. Build your model-based solution with scalability in mind

It is often said that *premature optimization is the root of all evil* but for small teams building a model-based infrastructure, initial decisions can be very costly to change afterward.

In our case, we started with a Java, Eclipse/EMF and Xtext stack as this was our "natural" option. We have already discussed how part of this stack changed but we could not afford to entirely start from scratch. And while Java is a great language, most of the NLP and ML developments are first released in Python and therefore, probably building Xatkit in Python (instead of having to wrap Python APIs and libraries) would have been a wise choice[18].

Another challenge was linked to the release of the eCommerce bot aimed at helping visitors to browse and ask questions about the catalog of an online shop. Xatkit was designed to be a single-bot application but deploying and maintaining a separate bot for every single shop was not scalable. We ended

[17]Not MDE related, but we had the same experience when highlighting that Xatkit was open source, this was not an important factor either for them

[18]Note that it is also easy to create DSLs in Python, similar to what we do with Xtext in Java, thanks to frameworks such as textX [23]

up developing an ad hoc solution for the eCommerce bot where the bot relied on a specialized Xatkit runtime that could be deployed as a kind of multi-tenant [24] generic ecommerce bot. This multi-tenant bot, or *metabot*, was able to determine which shop the request was coming from, and depending on that, it would instantiate the right bot state machine for the shop to properly answer the question.

If we had to develop more bots like this one, targeting a number of clients in the same domain, we will need to think how to be able to better model the individual bots and their interrelationships within a multi-tenant perspective.

6.3. Stop looking for the "right" concrete syntax

A critical decision when designing a DSL is the choice of the concrete syntax. But we often fail to remember that the same DSL could be linked to many different concrete syntaxes, each with their own trade-offs (e.g. in terms of expressiveness and usability).

We quickly realized that each user profile required a different syntax. More MDE minded people would be happy with the Eclipse environment required in our initial external DSL editor but our initial tests showed that most end users thought the Eclipse-based editor was too complicated. Even installing and using Eclipse itself was seen as too difficult.

Once we realized that, we dropped the idea of going for a common syntax aimed at being a good compromise among all the user profiles (us, technical end-users, non-technical users, ...) and decided to, on the one hand, create the "right" syntax for us (see Section 3.2.1) and then experimented with other syntaxes more oriented towards non-technical users. This even included an Excel-based syntax where users could simply use Excel to define their bots. Obviously, using our Excel template (see Figure 6) is very user-friendly, but it is limited to simple chatbots mostly covering FAQ-like and Question/Answering bots with limited conversation capabilities[19].

Even more, we also found out that, often, our clients had no interest in defining the bot themselves, they just wanted to hand us out the documentation (website, support emails and tickets, manuals,..) to create the bot and have us built the bot for them.

[19]Nothing prevented us from creating a more powerful Excel-based interface that could be used to create very generic bots but we felt that Excel was not the right interface for that, and we opted to use Excel as interface for non-tech people focused on simple bots while redirecting tech users to the DSL for more generic and powerful bots

Topic	*Supported Language*	
Questions	**Answers**	**Informations**
Do you speak?	Yes! We support more than 30 languages!	
Do you understand $language		
Can I create a bot in $langauage?		⚠ The entry langauage is not a valid entity.
Can I talk to you in $language?		
Do you know $language?		
I want a bot in $language		

Figure 6: Excel as a concrete syntax for creating chatbots

When creating a model-based solution, it is important to be clear about who will be the user profile/s we are targeting. For Xatkit, we were the main users ourselves and therefore we could adapt the MDE tooling to our own needs. Ask yourself whether you are selling a model-based tool or a service that you internally develop with your own model-based infrastructure. The right notation in both cases may be very different.

6.4. Modeling interactions with Large Language Models (LLMs)

Instead of building your own intent-based chatbot where you define the full list of intents/questions the chatbot should be able to match, you could use a generative chatbot based on large language models (LLMs) to have more open conversations with your website visitors. However, it is well-known such chatbots tend to "hallucinate" and invent facts, which, we believe, is something too risky for a public-facing chatbot, making them a far from ideal option as the core solution [25].

Nevertheless, clients do want this type of chatbots as this is all they hear about on the tech news. As a compromise, and as, indeed, they are partially useful, we have experimented with LLMs as *default fallback*. That is, when the bot does not understand the user request, it will resort to a LLM to try to provide the answer while alerting the user that the answer could be wrong. Figure 7 shows the schema we have developed for an open data bot used by citizens to query data sources published by the public administration. When the bot fails to match the question with a predefined intent, it tries to automatically translate it to a SQL expression using LLMs and attempts to execute such SQL on the data with the caveat that we cannot be certain that the generated SQL actually embeds the right semantics of the original question.

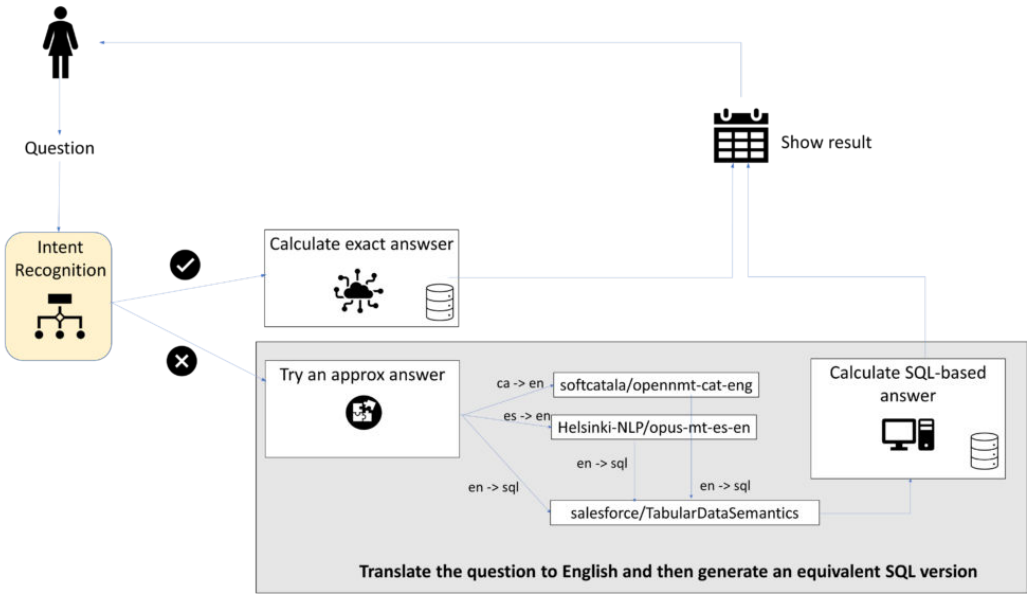

Figure 7: LLMs as default fallback

This solution was directly coded but given the increasing popularity of LLMs and the growing ecosystem of solutions/libraries/vendors providing them, there is for sure the need to start modeling the capabilities and interactions of any type of software, and in particular, chatbots, with generative AI components so that we can more easily deploy our bots on top of the hottest LLM at the time.

7. Related Work

It is well accepted in the MDE community that the adoption and success of MDE initiatives is a sociotechnical problem [26, 27, 28, 29, 30, 31, 32] and that the cost of adopting an MDE solution (or a DSL) pays off in the mid-term [33]. This paper represents an additional contribution to the discussion with two key distinctions: 1 - it covers the application of MDE on a new domain (chatbots) which has not been covered by previous works and 2 - it does it from a unique perspective, that of two researchers attempting the transition from a pure MDE research prototype to a commercial solution and explaining themselves the story.

Beyond Xatkit, it is worth noting there are other proposals for combining modeling (to some extent) with chatbot development. Among such

initiatives, we see an ongoing trend on adding low-code/no-code [34] front-ends to concrete chatbot platform. Some examples are: `Tock`[20], `Engati`[21] or `FlowXO`[22]. With a more research perspective, we have CONGA (ChatbOt modelliNg lanGuAge) [35] providing a unifying DSL for specifying some types of chatbots that can then be implemented on top of a couple of chatbot platforms (and even migrated from one to the other); and Baudat et al. [36] proposing wcs-OCaml, a new multi-tier chatbot generator library designed for use with the reactive language ReactiveML. Other works, such as `VoiceFlow`[23] focus on voicebots, providing a graphical DSL to create voice-based conversation flows that can be deployed on Google home or Alexa.

Some well-known low-code platforms like Mendix[24], GeneXus[25] or OutSystems[26] started to include chatbots as part of their system specification. But this support is rather limited and mostly consisting in either simple chatbot templates or in helping you to connect your application with an external AI component defined with a separate tool (e.g. one of the above).

As far as we know, Xatkit remains the most flexible solution with its modular architecture and runtime configuration options to be able to easily configure and deploy the modeled bot on different platforms. Nevertheless, we hope our experiences and lessons learned are helpful to other modeling initiatives around the chatbot domain. And, in general, to any domain that readers would like to approach with a modeling perspective.

As, often, chatbots are not the only UI of the system, it is also worth studying the interactions between a chatbot interface and other UIs, as part of a multiexperience user interface, something we started exploring here [37], focusing specially on the combination of our chatbot metamodel with the metamodel of other GUIs, in particular the IFML [38, 39]. On top of this, it is also interesting to see how the chatbots themselves are more and more capable of embedding graphical elements so the interweaving of chatbot interface models with interface models coming from other types of models, e.g. [40] or [41] is definitely worth considering. A clear chatbot metamodel such

[20] https://doc.tock.ai/tock/
[21] https://www.engati.com
[22] https://flowxo.com
[23] `https://www.voiceflow.com/`
[24] `https://www.mendix.com/`
[25] `https://www.genexus.com/en/`
[26] `https://www.outsystems.com/`

as the one presented here will facilitate these further integrations.

8. Conclusions

We have presented Xatkit, a model-driven chatbot development framework, and its evolution to better adapt to the requirements for bot development projects in real scenarios. We believe our experience is useful for practitioners and researchers who are developing an industrial-strength professional MDE product, but who may have little or no experience with the challenges of the professional/commercial product development.

On the technical side, there are still plenty of technical challenges to overcome, from a sublanguage to define policies to combine different NLP providers to a smoother integration of LLMs for automatic translation or default fallback support to the automatic generation of parts of the bot itself from available (un)structured documentation. All these topics are in our agenda for further work as part of a more complete proposal of a development process for chatbots that encompasses better techniques for the elicitation, testing [42] and evolution of bots and their combination and integration with other types of user interfaces, possibly as part of multi-experience interaction experience[37]. Clearly, a model-driven approach will still be the key to continue to advance Xatkit as it has been proven a key enabler of the Xatkit features and benefits so far.

Nevertheless, we also tried to make clear in our lessons learned that each of these new technical enhancements will need to be analyzed not only from a purely technical point of view but from a broader perspective to make sure we are aligned with the needs of future bot stakeholders and users.

References

[1] J. J. Garrett, Elements of user experience, the: user-centered design for the web and beyond, Pearson Education, 2010.

[2] L. C. Klopfenstein, S. Delpriori, S. Malatini, A. Bogliolo, The rise of bots: A survey of conversational interfaces, patterns, and paradigms, in: Proceedings of the 2017 Conference on Designing Interactive Systems, DIS, ACM, 2017, pp. 555–565.

[3] A. Xu, Z. Liu, Y. Guo, V. Sinha, R. Akkiraju, A new Chatbot for Customer Service on Social Media, in: Proc. of the 35th CHI Conference, ACM, 2017, pp. 3506–3510.

[4] A. Kerlyl, P. Hall, S. Bull, Bringing Chatbots into Education: Towards Natural Language Negotiation of Open Learner Models, in: Applications and Innovations in Intelligent Systems XIV, Springer, 2007, pp. 179–192.

[5] N. Thomas, An E-business Chatbot using AIML and LSA, in: Proc. of the 5th ICACCI Conference, IEEE, 2016, pp. 2740–2742.

[6] E. Shihab, S. Wagner, M. A. Gerosa, M. S. Wessel, J. Cabot, The present and future of bots in software engineering, IEEE Softw. 39 (5) (2022) 28–31. doi:10.1109/MS.2022.3176864.
URL https://doi.org/10.1109/MS.2022.3176864

[7] M. Brambilla, M. Dosmi, P. Fraternali, Model-driven engineering of service orchestrations, in: 2009 IEEE Congress on Services, Part I, SERVICES I 2009, Los Angeles, CA, USA, July 6-10, 2009, 2009, pp. 562–569.

[8] G. Daniel, J. Cabot, L. Deruelle, M. Derras, Xatkit: A multimodal low-code chatbot development framework, IEEE Access 8 (2020) 15332–15346.

[9] M. Rahimi, J. L. C. Guo, S. Kokaly, M. Chechik, Toward requirements specification for machine-learned components, in: 27th IEEE International Requirements Engineering Conference Workshops, RE, IEEE, 2019, pp. 241–244.

[10] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, P. Tonella, Testing machine learning based systems: a systematic mapping, Empir. Softw. Eng. 25 (6) (2020) 5193–5254.

[11] M. Kim, T. Zimmermann, R. DeLine, A. Begel, Data scientists in software teams: State of the art and challenges, IEEE Trans. Software Eng. 44 (11) (2018) 1024–1038.

[12] G. Daniel, J. Cabot, L. Deruelle, M. Derras, Multi-platform chatbot modeling and deployment with the jarvis framework, in: P. Giorgini, B. Weber (Eds.), Advanced Information Systems Engineering - 31st International Conference, CAiSE 2019, Vol. 11483 of Lecture Notes in Computer Science, Springer, 2019, pp. 177–193.

[13] Google, DialogFlow Website, URL: https://dialogflow.com/.
URL https://dialogflow.com/

[14] IBM, Watson Assistant Website, URL: https://www.ibm.com/watson/ai-assistant/.
URL https://www.ibm.com/watson/ai-assistant/

[15] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, Pearson Education, 2008.

[16] Amazon, Amazon Lex Website, URL: https://aws.amazon.com/lex/.
URL https://aws.amazon.com/lex/

[17] I. Qasse, S. Mishra, M. Hamdaqa, icontractbot: A chatbot for smart contracts' specification and code generation, in: 2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE), 2021, pp. 35–38. doi:10.1109/BotSE52550.2021.00015.

[18] A. Tarek, M. El Hajji, E.-S. Youssef, H. Fadili, Towards highly adaptive edu-chatbot, Procedia Computer Science 198 (2022) 397–403.

[19] M. Gomez, J. Cabot, R. Clarisó, Towards the automatic generation of conversational interfaces to facilitate the exploration of tabular data (2023). arXiv:2305.11326.

[20] M. F. Bertoa, L. Burgueño, N. Moreno, A. Vallecillo, Incorporating measurement uncertainty into OCL/UML primitive datatypes, Softw. Syst. Model. 19 (5) (2020) 1163–1189. doi:10.1007/s10270-019-00741-0.
URL https://doi.org/10.1007/s10270-019-00741-0

[21] M. Zhang, T. Yue, S. Ali, B. Selic, O. Okariz, R. Norgren, K. Intxausti, Specifying uncertainty in use case models, J. Syst. Softw. 144 (2018) 573–603. doi:10.1016/j.jss.2018.06.075.
URL https://doi.org/10.1016/j.jss.2018.06.075

[22] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, D. McClosky, The stanford corenlp natural language processing toolkit, in: Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations, 2014, pp. 55–60.

[23] I. Dejanović, R. Vaderna, G. Milosavljević, v. Vuković, TextX: A Python tool for Domain-Specific Languages implementation, Knowledge-Based Systems 115 (2017) 1–4. doi:10.1016/j.knosys.2016.10.023.
URL http://www.sciencedirect.com/science/article/pii/S0950705116304178

[24] F. Chong, G. Carraro, R. Wolter, Multi-tenant data architecture, MSDN Library, Microsoft Corporation (2006) 14–30.

[25] R. Shwartz-Ziv, A. Armon, Tabular data: Deep learning is not all you need, Information Fusion 81 (2022) 84–90.

[26] A. Vallecillo, On the industrial adoption of model driven engineering. is your company ready for mde?, International Journal of Information Systems and Software Engineering for Big Companies 1 (1) (2015) 52–68.

[27] J. E. Hutchinson, J. Whittle, M. Rouncefield, Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure, Sci. Comput. Program. 89 (2014) 144–161.
URL https://doi.org/10.1016/j.scico.2013.03.017

[28] T. Gorschek, E. D. Tempero, L. Angelis, On the use of software design models in software development practice: An empirical investigation, J. Syst. Softw. 95 (2014) 176–193.
URL https://doi.org/10.1016/j.jss.2014.03.082

[29] M.-J. Escalona, N. P. de Koch, G. Rossi, A quantitative swot-tows analysis for the adoption of model-based software engineering, Journal of Object Technology 21 (4) (2022).

[30] F. D. Giraldo, S. Espana, O. Pastor, W. J. Giraldo, Considerations about quality in model-driven engineering: Current state and challenges, Software Quality Journal 26 (2018) 685–750.

[31] A. Bucchiarone, J. Cabot, R. F. Paige, A. Pierantonio, Grand challenges in model-driven engineering: an analysis of the state of the research, Softw. Syst. Model. 19 (1) (2020) 5–13. doi:10.1007/s10270-019-00773-6.
URL https://doi.org/10.1007/s10270-019-00773-6

[32] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM computing surveys (CSUR) 37 (4) (2005) 316–344.

[33] O. Díaz, F. M. Villoria, Generating blogs out of product catalogues: An mde approach, Journal of Systems and Software 83 (10) (2010) 1970–1982.

[34] J. Cabot, Positioning of the low-code movement within the field of model-driven engineering, in: MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, ACM, 2020, pp. 76:1–76:3.

[35] S. Pérez-Soler, E. Guerra, J. de Lara, Model-driven chatbot development, in: 39th Int. Conf. on Conceptual Modeling, ER, Vol. 12400 of LNCS, Springer, 2020, pp. 207–222.

[36] G. Baudart, M. Hirzel, L. Mandel, A. Shinnar, J. Siméon, Reactive chatbot programming, in: Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH, ACM, 2018, pp. 21–30.

[37] E. Planas, G. Daniel, M. Brambilla, J. Cabot, Towards a model-driven approach for multiexperience ai-based user interfaces, Softw. Syst. Model. 20 (4) (2021) 997–1009.

[38] R. Acerbis, A. Bongio, M. Brambilla, S. Butti, Model-driven development based on omg's IFML with webratio web and mobile platform, in: P. Cimiano, F. Frasincar, G. Houben, D. Schwabe (Eds.), 15th International Conference, ICWE 2015, Vol. 9114 of Lecture Notes in Computer Science, Springer, 2015, pp. 605–608.

[39] M. Hamdani, W. H. Butt, M. W. Anwar, F. Azam, A systematic literature review on interaction flow modeling language (ifml), in: Proceedings of the 2018 2nd International Conference on Management Engineering, Software Engineering and Service Sciences, 2018, pp. 134–138.

[40] E. Díaz, J. I. Panach, S. Rueda, D. Distante, A family of experiments to generate graphical user interfaces from bpmn models with stereotypes, Journal of Systems and Software 173 (2021) 110883.

[41] N. Mezhoudi, J. Vanderdonckt, Toward a task-driven intelligent GUI adaptation by mixed-initiative, Int. J. Hum. Comput. Interact. 37 (5) (2021) 445–458. doi:10.1080/10447318.2020.1824742.
URL https://doi.org/10.1080/10447318.2020.1824742

[42] J. Cabot, L. Burgueño, R. Clarisó, G. Daniel, J. Perianez-Pascual, R. Rodríguez-Echeverría, Testing challenges for nlp-intensive bots, in: 3rd IEEE/ACM International Workshop on Bots in Software Engineering, BotSE@ICSE 2021, IEEE, 2021, pp. 31–34. doi:10.1109/BotSE52550.2021.00014.