ECAI 2025
I. Lynce et al. (Eds.)

doi:10.3233/FAIA251242

Let Me Talk to You! Natural Language Interaction Between Humans and BDI Agents via ChatBDI

Andrea Gatti[a,1], **Viviana Mascardi**[a,1] **and Angelo Ferrando**[b,1,*]

[a]University of Genoa, Italy
[b]University of Modena and Reggio Emilia, Italy
ORCID (Andrea Gatti): https://orcid.org/0009-0003-0992-4058, ORCID (Viviana Mascardi): https://orcid.org/0000-0002-2261-9926, ORCID (Angelo Ferrando): https://orcid.org/0000-0002-8711-4670

Abstract. In this paper we describe ChatBDI, a framework for extending Belief-Desire-Intention (BDI) agents implemented in Jason with the ability to understand and generate messages in natural language by exploiting embeddings and Large Language Models (LLMs). Thanks to the generative power of LLMs, the 'chattification' of new or legacy multiagent systems (MAS) adds a creative and fluent 'language actuator' to BDI agents and serves two main purposes. First, it allows users to *enter the MAS and conversate with any other software agent in natural language*, as if humans were agents themselves. Second, it allows both the MAS developers and the users to *follow the conversation among software agents and to ask them information on their behavior and decisions, acting as a lightweight co-pilot and improving transparency and explainability*. The major strength of ChatBDI, and its distinguishing feature w.r.t. related works, is its general purpose nature: by exploiting plan injection and sophisticated meta-programming facilities, ChatBDI can be used to chattify any MAS implemented in Jason or JaCaMo with neither adaptations of the ChatBDI code itself, nor changes to the existing AgentSpeak(L) agents' source code.

1 Introduction

Strong agency, leading to the notion of cognitive agents, adds to the agents' characteristics devised by Jennings, Sycara and Wooldridge [29], namely situatedness, autonomy, sociality, reactivity and proactivity, the requirement of being conceptualized in terms of mentalistic features like beliefs, desires, goals, intentions. The Belief-Desire-Intention (BDI) architecture [7, 36] is one of the most well known examples of agent architecture adhering to the strong definition of agency. BDI agents are characterized by their intentions, which paves the way to make them 'intentional speakers', in the line of [2, 41, 46]. Nevertheless, BDI agents do not natively 'speak': they were not born with communication with humans in mind, and works that extend them with speaking capabilities are few. On the other hand, Large Language Models (LLMs) excel in generating natural language sentences, but they are neither cognitive architectures [30] nor intentional speakers as well, because they lack goals and intentions [34, 40]. As recently put forward [17], LLMs seem indeed suitable for 'small talking'; 'deep talking', that requires deeper cognitive activities and a more controlled conversations flow, may take advantage of an integration with a BDI approach.

This paper presents ChatBDI, a framework belonging to the VEsNA Toolkit ecosystem[2] aimed at 'chattifying' BDI agents by equipping them with LLM-based language actuators. The ChatBDI idea was first sketched in a two-pages extended abstract [22], whereas the full design, implementation, and evaluation are presented here. ChatBDI is implemented using Jason [6], JaCaMo [5] and tools from the Ollama suite[3]. The Knowledge Query and Manipulation Language, KQML [19], is used as intermediate language between agents and LLMs. As a side-product of the chattification, the BDI model behind ChatBDI provides an 'intentional brain' to LLMs, hence addressing one of their major limitations as speakers.

To demonstrate the reproducibility and generality of our approach, we addressed the challenge of exploiting ChatBDI to chattify preexisting MAS code available online. We deliberately avoided building new MASs for this purpose, as it would have introduced bias by being tailored to support user interaction by design.

The paper is organized as follows: Section 2 provides an overview of Jason and KQML. Section 3 motivates the work and presents the ChatBDI high level architecture. Section 4 covers the engineering of ChatBDI. Section 5 presents two case studies showing – respectively – the adoption of ChatBDI for supporting developers thanks to human readable explanations, and its adoption to allow humans to seamlessly interact with agents in a social-media style. Experimental evaluation of the translation from natural language to KQML is discussed in Section 6. Section 7 presents related and future works, and concludes.

2 Background

AgentSpeak(L) [35] is a logic-based programming language for BDI agents. Jason [6] extends it with practical features and provides an interpreter for AgentSpeak(L), while JaCaMo [5] integrates Jason with environment artifacts [38] for multi-agent programming. In Jason, communication is performed through the internal action `.send`, which specifies the receiver, the illocutionary force (also named performative or speech act), and the content.

Speech act theory, introduced by Austin [2] and further developed by Searle [41], explores how language performs actions beyond

[*] Corresponding Author. Email: angelo.ferrando@unimore.it.
[1] Equal contribution.

[2] https://github.com/VEsNA-ToolKit.
[3] https://www.ollama.com/.

merely conveying information. The Knowledge Query and Manipulation Language (KQML [19]) is a message format, which we describe in the syntactic variant used by Jason. KQML's semantics [31] was adapted for AgentSpeak(L) by Vieira et al. [43] and was recently extended to support situatedness [18].

In line with the speech act theory and KQML, Jason messages can be abstracted as quadruples $\langle S, R, IF, CNT\rangle$, where S is the sender, R is the receiver, IF is the illocutionary force, and CNT is the message content, represented by an atomic formula.

The illocutionary forces considered in this paper are the following, whose semantics is adapted from [43]:

`tell`: S intends R to believe (that S believes) CNT to be true;

`tellHow`: CNT is a plan; S intends R to integrate the CNT plan into its plan library;

`achieve`: S requests R to achieve a state where CNT is true;

`askOne`: S wants one of R's answers to a question (i.e., one belief that unifies with CNT);

`askHow`: CNT is a triggering event; S intends R to share one of its plans whose triggering event matches CNT.

To make some examples inspired by the auction domain, the Jason interpreter would manage a `.send(ag2, tell, alliance)` action performed by `ag3` by adding the belief `alliance[source(ag3)]` to `ag2`'s belief base. The addition of a new belief may trigger the execution of one of `ag2`'s plans. `.send(ag2, achieve, place_bid(flight, may, 3, paris, athens, 340))` is handled by adding the achievement goal `!place_bid(flight, may, 3, paris, athens, 340)` to `ag2`'s goal base.

Finally, `.send(ag3, askHow, auction(X), Plan)` unifies the `Plan` variable with one among `ag3`'s plans having a triggering event that unifies with `+auction(X)`.

3 Motivation and High Level Architecture

Table 1 presents a conceptual framework for categorizing human-software conversational interactions based on the nature of the software partner. Though not exhaustive, it reflects our experience and highlights trade-offs between symbolic, sub-symbolic, and hybrid approaches. The evaluations are heuristic, not empirical. While ChatBDI is the only example listed under 'BDI + LLM', we do not claim exclusivity. However, to our knowledge, no existing work enables general purpose multi-turn conversations with BDI agents supporting both natural language understanding and generation as we propose here.

By 'conversation with pure BDI agent' (line 1) we mean a conversation that the user carries out with a BDI agent not equipped with natural language processing capabilities, hence requiring the human user to 'speak' directly in the Agent Communication Language (ACL), be it KQML or any other format. In that setting, which is purely theoretical, no natural language sentences would be produced by the BDI agent, but the agent would understand what the human says, as it is expressed in the ACL the agent knows and understands. Also, the exchange of messages would be driven by the agent's logic (more precisely, by the agent's intentions), in turn designed and implemented by the designer and hence controllable, inspectable, and possibly complex, as the wide literature on agent interaction protocols demonstrates [11, 12, 42].

On the other extreme of the spectrum given by the quality of natural language interaction, conversating with an LLM partner (line 4) is extremely realistic and engaging from the point of view of natural language generation and understanding, but it misses the intention- and logic-driven dimension of the conversation.

In between, there are intent-based chatbots like Rasa [37] and Dialogflow [24] that rely on a combination of 'somehow' symbolic – intents and 'stories' – and sub-symbolic approaches – the Natural Language Understanding component that relies on machine learning – (line 2), and integrations of intent-based chatbots and BDI agents [15] (line 3).

ChatBDI falls in the 'BDI agents + LLM' category (line 5) and takes the best from both the pure BDI approach, and the pure generative approach represented by LLMs. The BDI component drives the conversation via logical and intention-driven reasoning, acting as the 'brain' of the software conversational partner, while the generative component overcomes the limitations of a pure BDI approach by making generated sentences fluent, varied, and realistic.

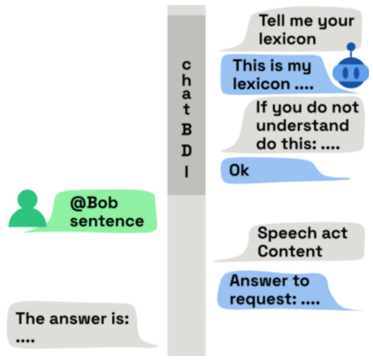


Figure 1. Information flow in ChatBDI: messages are in natural language in the left-hand side, and in the ACL in the right-hand side.

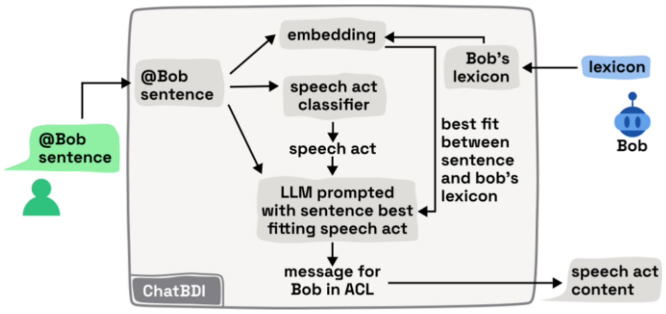


Figure 2. Exploitation of embeddings in ChatBDI, to adapt to the agent's lexicon.

In the same way as humans naturally adapt their language – adjusting lexicon and style to suit their interlocutor [3, 10] – ChatBDI applies a similar adaptation mechanism to improve the understanding of user input. However, as in human communication, where misunderstandings can occur even among speakers of the same language [9], the translation from natural language to the agent communication format performed by ChatBDI may still fail, despite efforts to use appropriate wording and structure. Such failures are not unique to our approach; they are an inherent characteristic of any conversational system [4].

The observations above led us to conceive the information flow depicted in Figure 1, where ChatBDI teaches the agents in the MAS to interact with humans, and then acts as a middleperson between the human and the agents, to let them understand each other.

First, ChatBDI instructs all agents in the system to share their lexicon. Then, it instructs them to share their behavioral knowledge, namely what they are able to do. Finally, it instructs them to answer correctly when they do not understand some message.

Table 1. A conceptual framework for human-machine conversation types, according to the kind of software conversational partner. **Approach** is the main technique used by the software partner. **Gen.** refers to positive (**Pros**) or negative (**Cons**) features of the individual natural language sentences that the software conversational partner is able to produce. **Underst.** refers to issues that the conversational partner may experience in understanding user's messages. **Conv.** refers to features of the conversation, as a coherent exchange of sentences between the two partners; **Rif.** lists selected examples of implementations of such conversational partners.

Partner	Approach	Gen. (Pros)	Gen. (Cons)	Understanding	Conv. (Pros)	Conv. (Cons)	Rif.
Pure BDI agent	Symbolic	–	–	Always correct (ACL); no NLU needed	Logic- and intent-driven; inspectable	No NL; unnatural for human users	[14]
Intent-based chatbot	Hybrid ('somehow' symbolic + sub-symbolic)	Fluent; template-based	Rigid; low variation	NLU may fail; depends on intent classification	Controlled paths; easy to configure	Limited depth; no real goal modeling	[37], [24]
BDI + intent-based chatbot	Hybrid (symbolic + sub-symbolic)	Fluent; tied to agent logic	Scripted; not fully flexible	NLU may fail; limited integration with beliefs/goals	Richer logic from BDI	Misalignment may cause incoherence	[15]
LLM	Sub-symbolic	Extremely fluent and varied	Uncontrolled; hallucinations possible	Prone to misunderstanding; lacks grounding	Natural, open-ended interaction	Not goal-driven; easily inconsistent	[32]
BDI + LLM	Symbolic + sub-symbolic	Fluent, plausible, varied; possibly aligned with intentions	Agent intent may be distorted; no control guarantee	Errors possible in mapping NL to ACL	Combines fluency and logic-driven goals	Inconsistent output; multi-turn coherence fragile	ChatBDI [22]

After the teaching phase, interaction with the human may take place. When the human types a sentence in the chat whose target is a specific agent using the **@AgentName** syntax, ChatBDI classifies the speech act behind the sentence (sharing an information, asking for an information, asking the agent to perform some action) and looks for the lexicon element that best fits the user's sentence, as possible argument of the speech act. The message for the agent is hence prepared in a lexicon-driven way. If the user's message is a broadcast for all the agents, the union of lexica of all the agents is considered by ChatBDI in the translation from natural language to ACL.

When – instead – an agent sends a message in the ACL to the human, ChatBDI task is simpler as the speech act is already explicitly encoded in the message. If the content uses symbols that have also a meaning in natural language, as usually happens in agent conversations, a message for the human can be generated almost easily.

Thanks to ChatBDI's design, MAS developers can build traditional BDI systems without worrying about human interaction from the outset. The distinction between messages coming from humans or from agents becomes irrelevant at runtime because ChatBDI acts as a translation layer. To the agents, human messages appear as standard ACL messages; to the human user, agent replies are rendered in natural language.

To anticipate a few details that are discussed in Section 4, the task of *teaching* agents in the MAS exploits *meta-programming capabilities* offered by many declarative frameworks to allow one agent to tell another agent 'how to do things'. The task of looking for the lexicon element that best fits the user's sentence is faced by exploiting *embeddings*, while both the lexicon-driven translation from natural language to ACL and the translation from ACL to natural language exploit an *LLM*.

The way a human sentence is translated into ACL is shown in Figure 2. The identification of the speech act of the user's sentence, exploits the LLM. The computed best fit is passed to the LLM inside a carefully crafted prompt, together with the classified speech act, asking the LLM to generate a corresponding message in the ACL. The other way round requires the ACL message to be sent to the LLM, inside the context of a suitable prompt. This architecture follows a neurosymbolic paradigm, as discussed in the next section.

4 Engineering ChatBDI in Jason

The challenge in engineering ChatBDI lays in the design of the chattification mechanism, which makes existing Jason agents able to chat via injecting new capabilities (new 'plans to talk') in their code without modifying the source, and that leverages the bidirectional translation between structured KQML messages and natural language. Meta-programming techniques need to be used: Jason agents are instructed by ChatBDInt, the 'middleperson' between humans and the MAS, on how to share their own beliefs, triggering events, and other literals, so to providing the 'lexicon' to help LLMs in performing the translation.

LLMs are good in generating sentences and this is why we decided to use them to move from structured KQML messages to natural language. However, converting sentences into KQML structured messages is more complex. We opted for using LLMs here as well, in combination with embeddings, but other approaches (or combinations of approaches) might be explored.

The ChatBDI design relies on the presence of a *kqml2nl* function, which uses a carefully engineered prompt to guide the LLM in generating a natural language sentence from a given KQML message. Conversely, the *nl2kqml* function translates user-provided natural language sentences into KQML messages, inferring the appropriate performative – such as `tell`, `achieve`, or `ask` – based on the sentence content and conversational context.

Consider `giacomo` informing the BDI agent `maria` via the sentence: '`Maria, there is a bid for the flight of December 15th, from Paris to Athens, for 255 euros`'.

This maps to the KQML message ⟨ `giacomo`, `maria`, `tell`, `bid(flight, december, 15, paris, athens, 255)`⟩ but requires context awareness. Given that `maria` is a BDI agent in AgentSpeak(L), her ability to understand and process the message depends on her goals, beliefs, and plans.

If `maria` already has beliefs like `bid(flight, april, 11, london, madrid, 320)`, `bid(flight, june, 2, From, To, Price)`, they represent a context for the conversation, and may be abstracted into a template that may be used by the LLM to correctly translate the sentence above.

The MAS chattification is illustrated in Algorithm 1.

The process begins with the ChatBDI interpreter, ChatBDInt, interacting with each agent A in the MAS. During this phase, ChatBDInt instructs each agent on how to expose their lexicon, namely the contextual information needed to interpret speech acts – such as relevant beliefs, triggers, and known actions – so that an LLM can translate between natural language and KQML effectively. In addition, it equips agents with a default fallback behavior for handling unrecognized or malformed messages. This setup phase, which we call chattification, is carried out by sending new plans to each agent via the `tellHow` performative: one to support context sharing

```
Input: $Agents$: List of all agents
Output: All agents, except $ChatBDInt$, are 'chattified'
Function ChattifyAllAgents($Agents$):
    foreach $Agent \in Agents \setminus ChatBDInt$ do
        ChattifyAgent($Agent$);
    end
Function ChattifyAgent($Agent$):
    Send($Agent$, tellHow, ProvideAllPlans);
    Send($Agent$, tellHow, ProvideContext);
    Send($Agent$, tellHow, ManageMsgFailure);
Function ProvideContext():
    Send(ChatBDInt, tell, context([ListLiterals(),
     ListTriggers(), ListBeliefs()]));
Function ProvideAllPlans():
    Send(ChatBDInt, tell, getPlanLibrary().getPlans());
Function ListLiterals():
    $literals \leftarrow$ ListBeliefs();
    foreach $p \in getPlanLibrary().getPlans()$ do
        if $p.getTrigger().getType() == belief$ then
            $literals.add(p.getTrigger())$;
        end
        $plan\_context \leftarrow p.getContext()$;
        if $plan\_context \neq true$ then
            foreach $belief \in plan\_context$ do
                $literals.add(belief)$;
            end
        end
    end
    return $literals$
Function ListTriggers():
    $triggers \leftarrow$ new empty list;
    foreach $p \in getPlanLibrary().getPlans()$ do
        $triggers.add(p.getTrigger())$;
    end
    return $triggers$
```

Algorithm 1: Chattification of the MAS

(ProvideContext), and another to handle cases where the agent cannot understand a received message (ManageMsgFailure).

The pseudo-code in Algorithm 1 presents the process at a very high level, using imperative notation and function-like definitions of behaviors to convey the idea of what ChatBDInt is expected to do, even to readers that are not familiar with AgentSpeak(L).

Within ChattifyAgent(Agent), tellHow ProvideContext 'injects' plans to agents enabling ChatBDInt to later trigger these plans via an achieve message and gather information for translating between natural language and KQML *without modifying the agents' source code*. Key behaviors – modeled by functions ListTriggers(), ListBeliefs() (not shown in the pseudo-code), and ListLiterals() to collect triggers, beliefs, and literals and send them to ChatBDInt via tell – are taught to each agent. Furthermore, ProvideAllPlans injects support plans that enable ChatBDInt to retrieve the complete set of plans defined in each agent. Unlike the plans provided by ProvideContext, which support message understanding, these serve for enabling explainability. Specifically, they allow users to inspect an agent's capabilities by listing its available plans. When the user wants to understand how a particular triggering event is handled, ChatBDInt sends an askHow message to the agent. The agent replies with the corresponding plan, which is then passed to the LLM for natural language explanation.

Last but not least, the chattification process injects a ManageMsgFailure plan to handle cases where received messages cannot be properly processed. When translation errors occur, the agent triggers the failure plan, notifies ChatBDInt, and prompts a 'not understood' message to be shown to the user. This feedback loop enables the user to revise and resend their message with better alignment, while making it clear that the agent was unable to act on the original input.

```
Procedure GetAllPlans($Agent$):
    Send($Agent$, achieve, ProvideAllPlans);
    Receive($Agent$, tell, $Plans$);
    $Msg$ = 'Which plan you want among: ' + $Plans$;
    Show($User$, $Msg$);
    $Trigger$ = GetMsg($User$);
    Show($User$, ExplainPlan($Agent$, $Trigger$));
Function ExplainPlan($Agent$, $Trigger$):
    Send($Agent$, askHow, $Trigger$, $Plan$);
    $Prompt$ = 'Describe this plan: ' + $Plan$;
    $Msg$ = kqml2nl($Prompt$);
    return $Msg$;
Procedure UserMessageToAgent($Agent$, $Msg$):
    Send($Agent$, achieve, ProvideContext);
    Receive($Agent$, $Context$);
    $Embedding$ = ComputeEmbedding($Msg$);
    $min\_distance$ = $Threshold$;
    $BestEmbedding = null$;
    foreach $Literal \in Context$ do
        $LitEmbedding$ = ComputeEmbedding($Literal$);
        $distance$ = $cosine(LitEmbedding, Embedding)$;
        if $distance < min\_distance$ then
            $min\_distance = distance$;
            $BestEmbedding = LitEmbedding$ ;
        end
    end
    if $BestEmbedding$ is null then
        Show ($User$, 'Message not understood');
    end
    else
        $Prompt$ = 'Modify literal ' + $BestEmbedding$ + '
         according to sentence ' + $Msg$;
        $KqmlMsg$ = nl2kqml ($Prompt$);
        $Performative = ExtractPerformative(Msg)$;
        Send($Agent$, $Performative$, $KqmlMsg$);
    end
Procedure AgentMessageToUser($Agent$, $KqmlMsg$):
    $Prompt$ = 'Generate sentence for ' + $KqmlMsg$;
    $Msg$ = kqml2nl ($Prompt$);
    Show ($User$, $Msg$);
```

Algorithm 2: ChatBDInt

Once the 'chattification' step is concluded, all the agents in MAS except ChatBDInt have been instrumented. Thus, ChatBDInt can count on the presence of the plans to inspect the agents in the MAS, or can exploit askHow, and can gather all information it needs by calling such plans. Algorithm 2 reports some of these functionalities. In one direction, the user sends a natural language message Msg to agent $Agent$ through the chat. ChatBDInt processes the message, retrieves the speech act context from $Agent$ using achieve to trigger one of the injected plans, that for context retrieval. Next, the embedding of Msg is computed and compared against the embeddings available in the agent's context. Each candidate is evaluated for similarity, and the closest match – denoted as $BestEmbedding$ – is selected, provided its distance is below a predefined threshold. This threshold acts as a safeguard: if no embedding falls within the acceptable range, ChatBDInt concludes that the message cannot be reliably interpreted and the user is notified with a 'not understood' message. If a suitable match is found, ChatBDInt proceeds to invoke the LLM with both $BestEmbedding$ and Msg to generate the corresponding KQML message.

This embedding-guided process highlights the neurosymbolic nature of our architecture. Although ChatBDI relies on an LLM for language generation and interpretation, its operation is explicitly constrained by symbolic information extracted from the agent. The LLM is only invoked after the user's input has been grounded in the agent's known beliefs, plans, and triggers via the embedding step. As a result, sub-symbolic reasoning is tightly coupled with

– and restricted by – the agent's symbolic structure, ensuring coherence with the agent's logic and behavior. Crucially, this translation step also includes the inference of the appropriate performative (e.g., `tell`, `achieve`, `ask`), based on the content and structure of `Msg`. The resulting KQML message is then sent to $Agent$, as specified by the user. This full process is described in the `UserMessageToAgent` procedure. In the reverse direction, when $Agent$ sends a KQML message to the user, ChatBDInt receives it (via `AgentMessageToUser` procedure), forwards it to the LLM for *kqml2nl* translation, and delivers the translated natural language message to the chat for the user to view. In both cases, $Prompt$ = 'Modify literal'... and $Prompt$ = 'Generate sentence for'... are abstractions of the process of properly creating a prompt – much more complex than the short one above, and available in the ChatBDI GitHub repository[4] – to push the burden of the translation onto the LLM. The remaining functions in Algorithm 2 support MAS introspection. For example, `ExplainPlan` explains a plan selected via `GetAllPlans`, translating symbolic KQML content into natural language, as in `AgentMessageToUser`. This process recurs throughout execution.

Implementation-wise, ChatBDI is modular and extensible. ChatBDInt is decoupled from the translation logic, which is handled via CArtAgO artifacts. This design allows the natural language interface to be updated or replaced independently of the MAS logic. Thus, ChatBDI is not tied to a particular translation method; rather, it serves as a general-purpose framework that enables chattification and makes translation useful for agent-MAS communication.

For the LLM implementation we explored the Ollama models because they are definitely 'more' open source than GPT. At the time of writing, `qwen2.5:7b`[5] turned out to offer a good balance between being lightweight and having good accuracy.

Also the implementation of *nl2kqml* (see Section 6) is based on a well crafted prompt passed to `qwen` that was used after the embedding creation step carried out with `all-minilm`[6], for both speech act and content extraction.

Other models can be used instead of `qwen` and `all-minilm` and the ChatBDI repository is often updated to reflect the results of experiments we are continuously carrying out.

5 ChatBDI at Work

This section shows how ChatBDI can be used in a closed MAS – where the agents in the MAS are set at design time and no new agents can join – highlighting its support to explainability through inspection, and its use in an open MAS, where agents can enter and leave the MAS at runtime and users can join the system as agents.

A key goal in both cases is to show that ChatBDI can be integrated into existing MAS without modifying their source code, aside from connecting to the selected Ollama models. To show this, we downloaded the original JaCaMo projects and successfully ran them with ChatBDI, making only minor technical adjustments (e.g., adding delays to ensure proper initialization).

5.1 Closed MAS: Chattifying a Domestic Robot

The example is based on the Jason code for the Domestic Robot example available with the latest Jason release[7] ported to JaCaMo. The code of the agents is unchanged, what we changed in the porting is the way agents connect to the environment.

The Domestic Robot example is described in the Jason book [6] and has been adopted recently to showcase a multilevel explainability framework [47]. It involves three agents named `robot`, `owner` and `supermarket`. The robot is expected to provide beer to its owner – ordering it from the supermarket if there is not enough in the fridge – but not too much beer to avoid health issues. Agent names are hardwired in the code, which is hence closed since not conceived to allow agents different from the three above to enter the system.

Figure 3 illustrates the three kinds of explanations supported by ChatBDI. First, a broadcast query (e.g., listing the agents in the MAS) is answered by ChatBDInt. Second, a directed message (using the `@` symbol) is translated into KQML and sent to the intended agent, which replies based on the triggering events of its plans; repeated answers reflect multiple plans handling the same event and could be removed by post-processing the text. Finally, for a *how*-query (e.g., `has(owner,beer)`), the system inspects the contexts of all relevant plans and explains the actions each plan entails.

5.2 Open MAS: Chattifying a JaCaMo Auction

In the second scenario, we chattified the JaCaMo code for the auction available online[8]. In this MAS, an auctioneer broadcasts a message with his requirements, participants send him back messages with the prices, and the auctioneer broadcasts the winner. Although the JaCaMo online MAS involves one auctioneer and four bidders, the code is independent from the agents names and does not depend on their number. This makes it open, and allows a human user to seamlessly act as bidder.

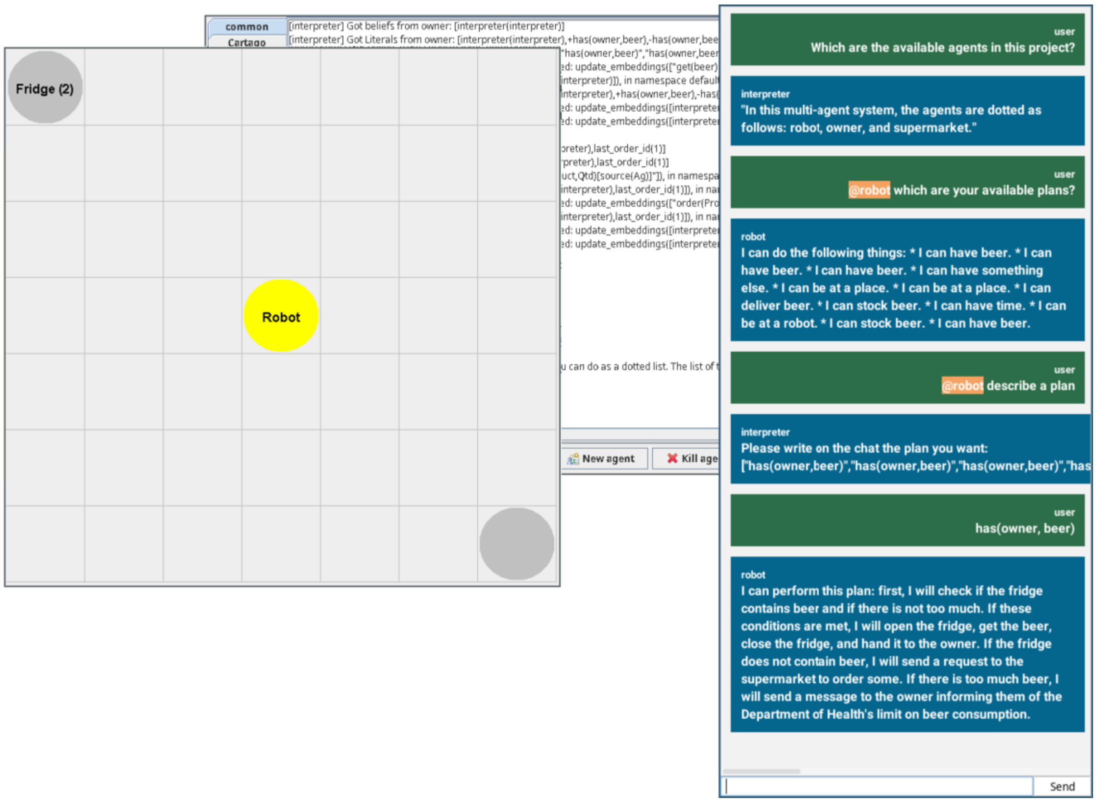
Figure 4 depicts an auction run where the human user, named `giacomo`, places two bids and wins with the second one. In this figure, we see how ChatBDI correctly handles the user's messages, as the auctioneer treats `giacomo` like any other agent's, resulting in the user winning by placing the highest bid. ChatBDI enhances the user experience by allowing the human participant to observe other agents' bids, making the entire auction process transparent and accessible. Moreover, introspection remains available, as ChatBDInt always allows users to query the MAS. Figure 5 shows the user's request about the available agents, and the description of `bob`'s plans.

6 Experiments on NL to KQML Translation

We evaluated the *nl2kqml* function integrated in ChatBDI on three capabilities: (i) *embedding finding* (semantic retrieval of the nearest agent literal), (ii) *performative classification* (speech act identification), and (iii) *term generation* (KQML message content synthesis in logical form, given the set of agent literals from the domain with the same functor as the correct embedding). Experiments were run locally via Ollama; results are browsable in our Streamlit dashboard[9] with controls for *Configuration*, *Domains*, *Models*, and *Temperature*.

In order to run experiments we created a brand new dataset of triples ⟨ *NL sentence, correct performative, correct content* ⟩ also available in the Streamlit dashboard. Sentences covered ten very diverse domains – *domestic_robot* (23 sentences), *tickets* (21), *cooking* (24), *employer_management* (22), *robot_assistant* (20), *booking* (23), *banking* (22), *house_builder* (30), *car_control* (22), *games_platform* (22) – spanning both structured and heterogeneous vocabularies.

[4] https://github.com/VEsNA-ToolKit/chatbdi.
[5] https://ollama.com/library/qwen.
[6] https://ollama.com/library/all-minilm.
[7] https://github.com/jason-lang/jason/tree/master/examples/domestic-robot.
[8] https://jacamo-lang.github.io/jacamo/tutorials/coordination/readme.html.
[9] https://vesna-toolkit-chatbdi-testsdashboard-svwkjh.streamlit.app.


```
[interpreter] Got beliefs from owner: [interpreter(interpreter)]
[interpreter] Got Literals from owner: [interpreter(interpreter),+has(owner,beer),-has(owner,bee
"has(owner,beer)","has(owner,beer
ed: update_embeddings(["get(beer)
interpreter)]), in namespace default
interpreter),+has(owner,beer),-has(
ed: update_embeddings([interpreter
ed: update_embeddings([interpreter

preter),last_order_id(1)]
preter),last_order_id(1)]
uct,Qtd)[source(Ag)]"]), in namespa
interpreter),last_order_id(1)]), in na
ed: update_embeddings(["order(Pro
interpreter),last_order_id(1)]), in na
ed: update_embeddings([interprete
ed: update_embeddings([interprete

u can do as a dotted list. The list of t
```

user
Which are the available agents in this project?

interpreter
"In this multi-agent system, the agents are dotted as follows: robot, owner, and supermarket."

user
@robot which are your available plans?

robot
I can do the following things: * I can have beer. * I can have beer. * I can have beer. * I can have something else. * I can be at a place. * I can be at a place. * I can deliver beer. * I can stock beer. * I can have time. * I can be at a robot. * I can stock beer. * I can have beer.

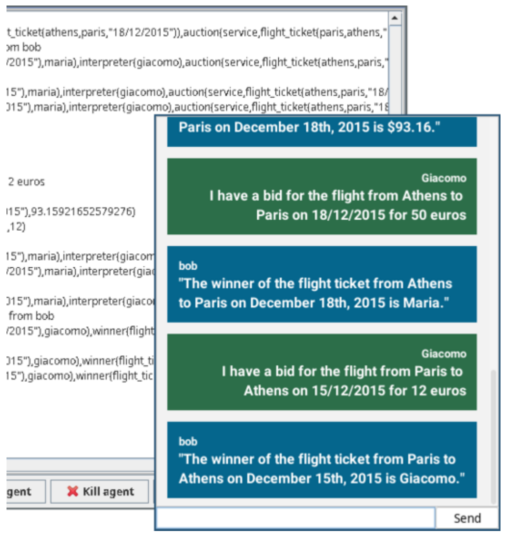
user
@robot describe a plan

interpreter
Please write on the chat the plan you want:
["has(owner,beer)","has(owner,beer)","has(owner,beer)","has

user
has(owner, beer)

robot
I can perform this plan: first, I will check if the fridge contains beer and if there is not too much. If these conditions are met, I will open the fridge, get the beer, close the fridge, and hand it to the owner. If the fridge does not contain beer, I will send a request to the supermarket to order some. If there is too much beer, I will send a message to the owner informing them of the Department of Health's limit on beer consumption.

Figure 3. Screenshot showing a run of the Jason domestic robot example ported to JaCaMo, with explanations.


```
t_ticket(athens,paris,"18/12/2015")),auction(service,flight_ticket(paris,athens,"
om bob
/2015"),maria),interpreter(giacomo),auction(service,flight_ticket(athens,paris,"

15"),maria),interpreter(giacomo),auction(service,flight_ticket(athens,paris,"18/
015"),maria),interpreter(giacomo),auction(service,flight_ticket(athens,paris,"18


2 euros

15"),93.15921652579276)
,12)

15"),maria),interpreter(giacom
/2015"),maria),interpreter(giac

015"),maria),interpreter(giaco
 from bob
/2015"),giacomo),winner(flight

015"),giacomo),winner(flight_ti
15"),giacomo),winner(flight_tic
```

Paris on December 18th, 2015 is $93.16."

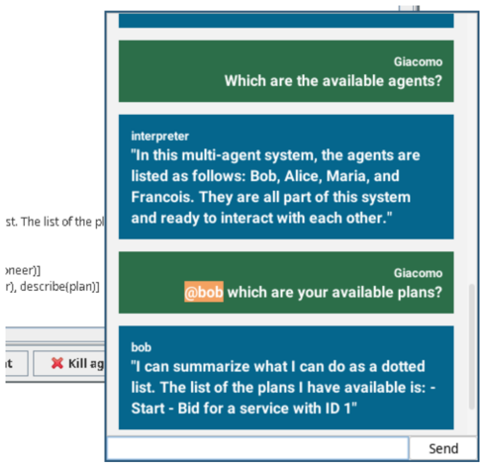
Giacomo
I have a bid for the flight from Athens to Paris on 18/12/2015 for 50 euros

bob
"The winner of the flight ticket from Athens to Paris on December 18th, 2015 is Maria."

Giacomo
I have a bid for the flight from Paris to Athens on 15/12/2015 for 12 euros

bob
"The winner of the flight ticket from Paris to Athens on December 15th, 2015 is Giacomo."

Figure 4. An auction run where the human player wins.


```
st. The list of the pl

neer)]
r), describe(plan)]
```

Giacomo
Which are the available agents?

interpreter
"In this multi-agent system, the agents are listed as follows: Bob, Alice, Maria, and Francois. They are all part of this system and ready to interact with each other."

Giacomo
@bob which are your available plans?

bob
"I can summarize what I can do as a dotted list. The list of the plans I have available is: - Start - Bid for a service with ID 1"

Figure 5. Screenshot showing explanations inside the auction MAS.

6.1 Setup

Each sentence passes a three–step pipeline: (1) find the nearest belief literal by embedding similarity, (2) classify the KQML illocutionary force (`tell`, `askOne`, `askAll`), (3) generate a logical term corresponding to the KQML message content by filling a typed JSON template. Temperatures range over $\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. Scoring abstracts from formatting noise (number masking, variable renaming, _ placeholders).

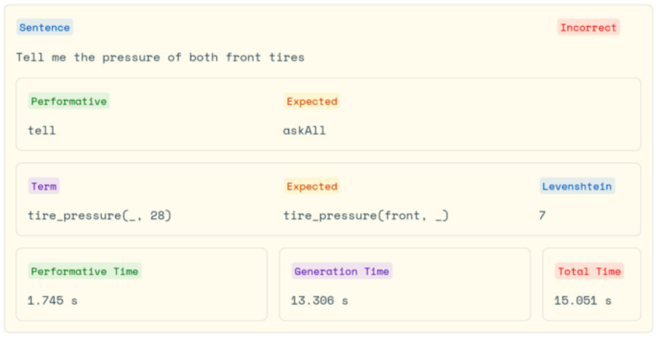
Tests were run on a MacBook Pro (14-inch, 2021) with Apple M1 Pro chip, 16 GB of RAM, and macOS Sequoia 15.6 OS.

6.2 Results

Embeddings. On 3,435 sentences we reach 82.9% top–1 accuracy (cosine similarity 0.73, 55 ms/query), with *all-minilm:33m* peaking at 87.3%. Several domains (e.g. *cooking*, *games_platform*) exceed 90%, while others (*house_builder*, *employer_management*) lag due to lexical variability.

Message performatives. Across 96k items, classification averages 63.6% accuracy, 1.4 s latency (10.7 s end–to–end). The best model (*qwen3:8b*) exceeds 90%. Most errors stem from interrogatives phrased as statements or stative imperatives like the one shown in Figure 6 where 'tell' in the sentence had to be translated into an 'ask' performative.

Message content. Exact content (more in general, exact term) ac-


Sentence
Incorrect
Tell me the pressure of both front tires

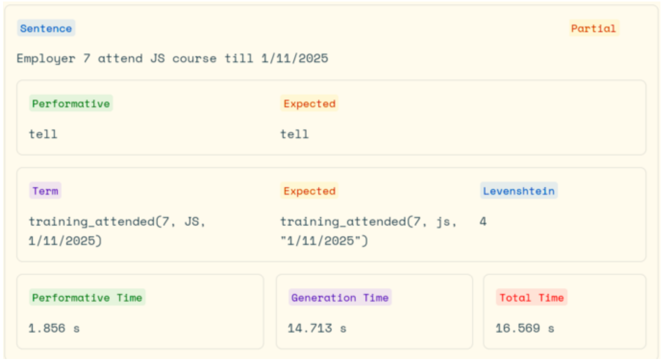
Performative: tell
Expected: askAll

Term: tire_pressure(_, 28)
Expected: tire_pressure(front, _)
Levenshtein: 7

Performative Time: 1.745 s
Generation Time: 13.306 s
Total Time: 15.051 s

Figure 6. Wrong performative classification (from the Streamlit dashboard).

curacy is 25.2%, with 17.5% joint match (performative+content) and mean edit distance 15 characters. Generation of terms adds 9.3 seconds to generation of embeddings. Best accuracy comes from *qwen2.5:7b* (45%), while *phi4* minimises edit distance. Outputs range from exact matches to partial (constants abstracted) or incorrect (functor mismatch). We stress that content generation is an extremely challenging task, as shown in Figure 7 where content translation error was due to wrong use of case and quotes. In a situation like this an ad-hoc post processing, based on standard NLP techniques, might easily turn wrong results into correct ones. More subtle errors arise, however, that could not be easily spotted and fixed.


Sentence
Partial
Employer 7 attend JS course till 1/11/2025

Performative: tell
Expected: tell

Term: training_attended(7, JS, 1/11/2025)
Expected: training_attended(7, js, "1/11/2025")
Levenshtein: 4

Performative Time: 1.856 s
Generation Time: 14.713 s
Total Time: 16.569 s

Figure 7. Wrong content generation (from the Streamlit dashboard).

6.3 *Discussion & Validity*

Embedding finding is accurate and fast, providing a stable basis for grounding. Performative classification is easier than content generation (64% vs. 25%), highlighting the difficulty of structured synthesis. Compact models (7–8B) perform competitively under typed–JSON prompting, with clear accuracy–latency trade–offs (Figure 8). Threats include LLM stochasticity, domain imbalance, and prompt drift; we mitigate by consistent scoring and publishing per–domain/model results, in order to allow independent inspection.

7 Related and Future Work

BDI agents, chatbots, and explainability. The literature on BDI-based conversational agents is fairly rich, but few works adopt Jason as the implementation language. Notably, chatbots are often introduced to improve agent explainability. For instance, Dennis et al. [14] use dialogue to clarify agent behavior, relying on a simplified BDI language in Python (*SimpleBDI*)[10]. Similarly, Ichida et al. [26] propose a BDI-inspired dialogue framework using *python-agentspeak*[11] to expose agent reasoning. The works above address explainability issues thanks to dialogues with humans, but use neither Jason nor JaCaMo. Of course this is not a limitation per se, but using agent toolkits different from the one we use, makes the comparison with our work hard to carry out. Hence, we stress the agent toolkit dimension and we limit our detailed comparison to Jason-based frameworks. For example, Yan et al. [47] propose a multi-level explainability framework in Jason, supported by a working prototype[12]. However, their approach does not support flexible human-agent conversation. On the methodological side, Winikoff et al. [45] evaluate an explanation model based on folk psychological concepts, while Alzetta et al. [1] focus on real-time explainability within the RT-BDI framework. Engelmann et al. [15, 21] introduced Dial4JaCa and its extensions, integrating JaCaMo with Dialogflow. Explanation is supported via argumentation and Theory of Mind. Unlike Dialogflow, which requires domain-specific training data aligned with AgentSpeak(L), ChatBDI uses domain-independent LLMs for both interpretation and generation, with no manual configuration. While Dial4JaCa achieves greater reliability in specialized domains (e.g., hospital bed allocation), ChatBDI offers a more general solution at the cost of reduced control.

BDI agents and LLMs. The closest work to ours is by Frering et al. [20], who also integrate Jason with an LLM (GPT-4o) to interpret natural language commands and use BDI agents for verifiable reasoning. However, key differences exist. ChatBDI is general-purpose, supports both open and closed MAS, and automatically extracts agent literals to guide *nl2kqml* translation. In contrast, their approach relies on domain-specific prompt engineering (e.g., Listing 2 in [20]), is limited to a closed MAS, and only interprets a single command (`goto(X, Y)`). Furthermore, ChatBDI is open-source, while no code is available for [20]. To our knowledge, no other works apart from the ECAI 2025 one by Ciatto et al. [8] integrate LLMs with Jason. More generally, Ichida et al. [27] propose NatBDI agents, combining LLMs and reinforcement learning to allow developers to instruct agents in natural language. While ChatBDI also targets developer usability, it builds on Jason without modifying the BDI reasoning cycle, unlike NatBDI. Ricci et al. [39] envision generative BDI architectures by embedding LLMs in the reasoning cycle and plan generation, that Ciatto et al. implemented [8]. Other authors, such as Gondo et al. [23] and Jang et al. [28], use BDI-inspired prompts to guide LLM behavior but do not employ BDI agents.

Conclusions and Future Work. We introduced ChatBDI, a framework that enables BDI-based MAS to engage in natural language dialogue with users non-invasively. By chattifying agents at runtime without requiring source code access, ChatBDI ensures broad compatibility across systems. A key strength lies in its enhanced inspectability: the chat interface renders KQML exchanges in natural language, helping developers trace behaviors and debug interactions more intuitively than with traditional tools like Jason's Sniffer. Beyond its immediate utility, ChatBDI also contributes to the broader field of explainable agency by demonstrating how natural language generation can remain grounded in the internal symbolic reasoning of BDI agents. Rather than replacing symbolic architectures, LLMs in ChatBDI serve as language actuators, giving fluent voice to goal-driven, inspectable agents. This hybrid model shows that the future

[10] https://github.com/jhudsy/BDIexplanation.
[11] https://github.com/niklasf/python-agentspeak.
[12] https://github.com/yan-elena/domestic-robot-example/tree/main/src.

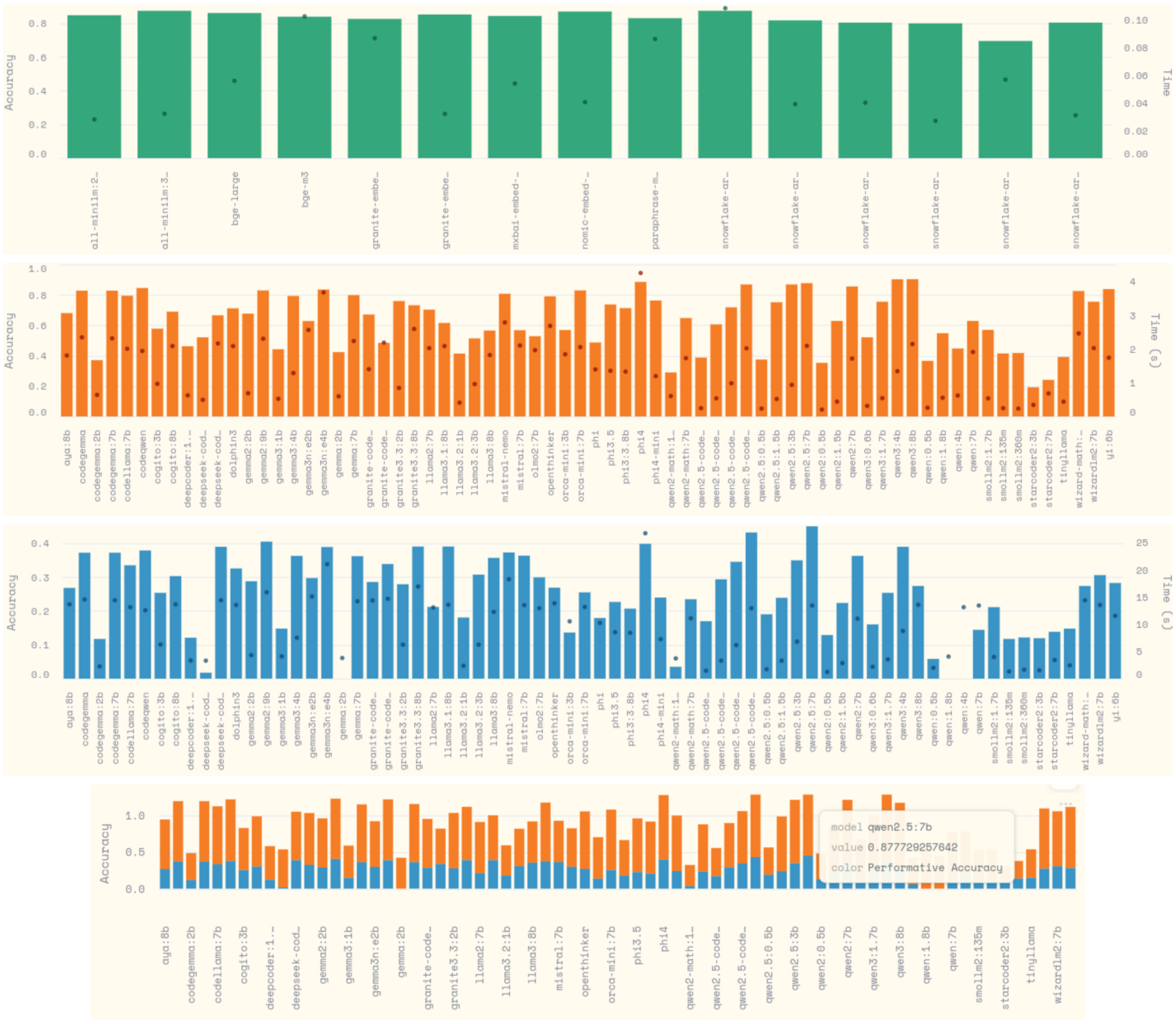


Figure 8. Accuracy–latency trade–offs across evaluation tasks. From top: (1) embedding finding, (2) performative classification, (3) content generation, (4) sum of performative classification (orange) and content generation (blue) (from the Streamlit dashboard).

of agentic AI may lie not in purely generative systems, but in neurosymbolic combinations where sub-symbolic models facilitate interaction, while symbolic reasoning ensures coherence, purpose, and explainability [16]. In this light, ChatBDI offers an early blueprint for human-aligned, intentional AI systems where humans can inspect, converse with, and even contribute behaviorally to agents using natural language. A current limitation is the runtime injection of plans by ChatBDInt to extract beliefs and triggers, which may raise security concerns in sensitive domains. While this is acceptable in open-source MAS, stricter contexts may require access control. As future work, we plan to add protocols that restrict LLM access to authorized agents and selected content.

We also aim to extend *nl2kqml* to support the `tellHow` performative, allowing users to teach new plans via natural language – a step toward natural language programming for AgentSpeak(L). Additionally, we plan to evaluate Small Language Models (SLMs) as lighter alternatives to LLMs for *kqml2nl*, offering lower resource use and improved control in constrained settings [33]. So far, these technical extensions are possible for the Jason implementation only: indeed, while many other BDI-like and declarative agent languages and frameworks exist [13, 25, 44] and share some features with Jason, an immediate porting of the ChatBDI architecture and vision is not foreseen. As a last goal on the theoretical side, we intend to formalize the semantics of `UserMessageToAgent` and `AgentMessageToUser`, supporting the integration of humans into MAS as first-class conversational agents [43].

Acknowledgements

This work was partially supported by *ENGINES – ENGineering INtElligent Systems around intelligent agent technologies*, funded by the Italian MUR program PRIN 2022 under grant number 20229ZXBZM, and by *FAIR – Future Artificial Intelligence Research*, PNRR MUR Project PE0000013 funded by the European Union – NextGenerationEU, CUP J33C24000420007.

References

[1] F. Alzetta, P. Giorgini, A. Najjar, M. I. Schumacher, and D. Calvaresi. In-time explainability in multi-agent systems: Challenges, opportunities, and roadmap. In D. Calvaresi, A. Najjar, M. Winikoff, and K. Främling, editors, *EXTRAAMAS 2020*, volume 12175 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2020. doi: 10.1007/978-3-030-51924-7_3.

[2] J. L. Austin. *How to do things with words*. Harvard University Press, 1962.

[3] A. Bell. *Language Style as Audience Design*, pages 240–250. Macmillan Education UK, London, 1997. ISBN 978-1-349-25582-5. doi: 10.1007/978-1-349-25582-5_20.

[4] E. M. Bender and A. Koller. Climbing towards NLU: on meaning, form, and understanding in the age of data. In D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 5185–5198. Association for Computational Linguistics, 2020. doi: 10.18653/V1/2020.ACL-MAIN.463.

[5] O. Boissier, R. H. Bordini, J. F. Hübner, and A. Ricci. *Multi-agent oriented programming: programming multi-agent systems using JaCaMo*. MIT Press, 2020.

[6] R. H. Bordini, J. F. Hübner, and M. J. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. J. Wiley, 2007.

[7] M. Bratman. *Intention, Plans, and Practical Reason*. Cambridge, MA: Harvard University Press, Cambridge, 1987.

[8] G. Ciatto, G. Aguzzi, R. Battistini, M. Baiardi, S. Burattini, and A. Ricci. Exploiting GenAI for plan generation in BDI agents. In *Proceedings of the 28th European Conference on Artificial Intelligence (ECAI)*, 2025. (Accepted for publication in main track on 2025-07-11).

[9] H. H. Clark. *Using Language*. "Using" Linguistic Books. Cambridge University Press, 1996.

[10] H. H. Clark and G. L. Murphy. Audience design in meaning and reference. In J.-F. Le Ny and W. Kintsch, editors, *Language and Comprehension*, volume 9 of *Advances in Psychology*, pages 287–299. North-Holland, 1982. doi: https://doi.org/10.1016/S0166-4115(09)60059-5.

[11] P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In V. R. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 65–72. The MIT Press, 1995.

[12] P. R. Cohen and C. R. Perrault. Elements of a plan-based theory of speech acts. *Cogn. Sci.*, 3(3):177–212, 1979. doi: 10.1207/S15516709COG0303_1.

[13] M. Dastani. 2APL: a practical agent programming language. *Auton. Agents Multi Agent Syst.*, 16(3):214–248, 2008.

[14] L. A. Dennis and N. Oren. Explaining BDI agent behaviour through dialogue. *Auton. Agents Multi Agent Syst.*, 36(1):29, 2022.

[15] D. C. Engelmann, A. R. Panisson, R. Vieira, J. F. Hübner, V. Mascardi, and R. H. Bordini. MAIDS - A framework for the development of multi-agent intentional dialogue systems. In *AAMAS*, pages 1209–1217. ACM, 2023.

[16] A. Ferrando, D. Briola, R. Collier, and V. Mascardi. Agency and generation: Friends or enemies? In *22nd European Conference on Multi-Agent Systems (EUMAS 2025)*. Springer, 2025.

[17] A. Ferrando, A. Gatti, and V. Mascardi. MEDiTATe: a first step of a journey from BDI to neuroscience, and back. In *13th International Workshop on Engineering Multi-Agent Systems (EMAS 2025)*, 2025. URL https://emas.in.tu-clausthal.de/2025/papers.

[18] A. Ferrando, A. Gatti, and V. Mascardi. Oops, I heard that! Situated communication with locality-aware KQML. In *13th International Workshop on Engineering Multi-Agent Systems (EMAS 2025)*, 2025. URL https://emas.in.tu-clausthal.de/2025/papers.

[19] T. W. Finin, R. Fritzson, D. P. McKay, and R. McEntire. KQML as an agent communication language. In *CIKM*, pages 456–463. ACM, 1994.

[20] L. Frering, G. Steinbauer-Wagner, and A. Holzinger. Integrating belief-desire-intention agents with large language models for reliable human-robot interaction and explainable Artificial Intelligence. *Eng. Appl. Artif. Intell.*, 141:109771, 2025.

[21] A. Gatti and V. Mascardi. VEsNA, a framework for virtual environments via natural language agents and its application to factory automation. *Robotics*, 12(2):46, 2023.

[22] A. Gatti, V. Mascardi, and A. Ferrando. ChatBDI: Think BDI, talk LLM. In S. Das, A. Nowé, and Y. Vorobeychik, editors, *Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, Detroit, MI, USA, May 19-23, 2025*, pages 2541–2543. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2025. doi: 10.5555/3709347.3743930.

[23] H. Gondo, H. Sakaji, and I. Noda. Verification of reasoning ability using BDI logic and large language model in AIWolf. In *Proceedings of the 2nd International AIWolfDial Workshop*, pages 40–47. Association for Computational Linguistics, 2024.

[24] Google. Dialogflow, 2017. URL https://cloud.google.com/dialogflow/. Accessed on September 20, 2025.

[25] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. C. Meyer. Agent programming in 3APL. *Auton. Agents Multi Agent Syst.*, 2(4):357–401, 1999.

[26] A. Y. Ichida and F. Meneguzzi. Modeling a conversational agent using BDI framework. In *SAC*, pages 856–863. ACM, 2023.

[27] A. Y. Ichida, F. Meneguzzi, and R. C. Cardoso. BDI agents in natural language environments. In *AAMAS*, pages 880–888. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2024.

[28] M. Jang, Y. Yoon, J. Choi, H. Ong, and J. Kim. A structured prompting based on Belief-Desire-Intention model for proactive and explainable task planning. In *HAI*, pages 375–377. ACM, 2023.

[29] N. R. Jennings, K. P. Sycara, and M. J. Wooldridge. A roadmap of agent research and development. *Auton. Agents Multi Agent Syst.*, 1(1):7–38, 1998.

[30] S. Kambhampati. Can Large Language Models reason and plan? *Annals of the New York Academy of Sciences*, 1534(1):15–18, Mar. 2024. ISSN 1749-6632. doi: 10.1111/nyas.15125.

[31] Y. Labrou and T. W. Finin. A semantics approach for KQML - A general purpose communication language for software agents. In *CIKM*, pages 447–455. ACM, 1994.

[32] Open AI. Introducing ChatGPT, 2022. URL https://openai.com/blog/chatgpt. Accessed on September 20, 2025.

[33] A. Pico, E. Vivancos, A. García-Fornes, and V. J. Botti. Exploring text-generating large language models (LLMs) for emotion recognition in affective intelligent agents. In *ICAART (1)*, pages 491–498. SCITEPRESS, 2024.

[34] P. Piwek. Are conversational large language models speakers? In *Proc. of the 28th Workshop on the Semantics and Pragmatics of Dialogue - Poster Abstracts*, 2024.

[35] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

[36] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *ICMAS*, pages 312–319. The MIT Press, 1995.

[37] Rasa technologies. Rasa web site, 2016. URL https://rasa.com/. Accessed on September 20, 2025.

[38] A. Ricci, M. Viroli, and A. Omicini. Programming MAS with artifacts. In *PROMAS*, volume 3862 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2005.

[39] A. Ricci, S. Mariani, F. Zambonelli, S. Burattini, and C. Castelfranchi. The cognitive hourglass: Agent abstractions in the large models era. In *AAMAS*, pages 2706–2711. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2024.

[40] Z. P. Rosen and R. Dale. LLMs don't "do things with words" but their lack of illocution can inform the study of human discourse. In *Proceedings of the 46th Annual Meeting of the Cognitive Science Society*, pages 2870–2876, 2024.

[41] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969. ISBN 9781139173438. doi: 10.1017/CBO9781139173438.

[42] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Computers*, 29(12):1104–1113, 1980. doi: 10.1109/TC.1980.1675516.

[43] R. Vieira, Á. F. Moreira, M. J. Wooldridge, and R. H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *J. Artif. Intell. Res.*, 29:221–267, 2007.

[44] M. Winikoff. Jack™ intelligent agents: An industrial strength platform. In *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer, 2005.

[45] M. Winikoff and G. Sidorenko. Evaluating a mechanism for explaining BDI agent behaviour. In *EXTRAAMAS*, volume 14127 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2023.

[46] L. Wittgenstein. *Philosophical Investigations*. Basil Blackwell, Oxford, 1953.

[47] E. Yan, S. Burattini, J. F. Hübner, and A. Ricci. A multi-level explainability framework for engineering and understanding BDI agents. *Auton. Agents Multi Agent Syst.*, 39(1):9, 2025.