Mitigating LLM Hallucinations in Code Generation for BDI Agents

Natália Mendes Goes[1], **Rafael C. Cardoso**[2], **Gleifer V. Alves**[1], **André P. Borges**[1]

Universidade Tecnológica Federal do Paraná (UTFPR)
Ponta Grossa, PR, Brasil

University of Aberdeen
Aberdeen, United Kingdom

nataliamendes@alunos.utfpr.edu.br, rafael.cardoso@abdn.ac.uk

{gleifer, apborges}@utfpr.edu.br

Abstract. *This paper presents MASPY-LLM, an autonomous code-generation pipeline designed to mitigate hallucinations in Large Language Models (LLMs) when programming Belief-Desire-Intention (BDI) agents within the Multi-Agent System for Python (MASPY) framework. The hybrid architecture integrates Retrieval-Augmented Generation (RAG) context injection, Actor-Critic orchestration, and algorithmic sanitisation to correct topological breaks and prevent context amnesia. To evaluate the approach, a case study focused on the contract-net negotiation protocol was conducted. The results demonstrate the pipeline's capability to compile syntactically valid code that adheres to MASPY's constraints, reducing the learning curve in modelling systems within the framework.*

Resumo. *Este artigo apresenta o MASPY-LLM, um pipeline autônomo para geração de código projetado para mitigar as alucinações de Large Language Models (LLMs) na programação de agentes Belief-Desire-Intention (BDI) no framework Multi-Agent System for Python (MASPY). A arquitetura híbrida integra injeção de contexto, Retrieval-Augmented Generation (RAG), orquestração ator-crítico e higienização algorítmica para corrigir quebras topológicas e previnir amnésia de contexto. Para avaliar a abordagem da ferramenta, instanciou-se um estudo de caso focado no protocolo de negociação contract-net. Os resultados demonstram a capacidade do pipeline em compilar códigos sintaticamente válido e aderente às restrições do MASPY, reduzindo a curva de aprendizado na modelagem de sistemas no framework.*

1. Introdução

A integração de *Large Language Models* (LLMs) [Shanahan 2024] impulsionou o desenvolvimento de sistemas agênticos [Park et al. 2023]. Estas entidades interpretam linguagem natural e atuam colaborativamente na resolução de problemas, sendo aplicadas tanto como desenvolvedores de *software* autônomos [Silva et al. 2025] quanto na simulação de ambientes econômicos e sociais [Park et al. 2023]. Na estruturação desses ecossistemas, utilizam-se Sistemas Multiagente (SMA) [Wooldridge 2009] baseados no paradigma *Belief-Desire-Intention* (BDI) [Bratman 1987], que garante um raciocínio deliberativo por meio de crenças, desejos e intenções [Cardoso and Ferrando 2021]. O *Multi-Agent System for Python* (MASPY) [Mellado et al. 2025] é um *framework* acadêmico que facilita a

programação desses agentes. Com sua adoção, torna-se viável utilizar agentes LLM para automatizar a geração de código estruturado no *framework*.

A interseção entre a programação de *frameworks* BDI restritos e a geração autônoma de código por LLMs apresenta desafios. Devido à ausência de conhecimento prévio sobre o *framework*, os LLMs assumem incorretamente que a geração deve seguir os paradigmas convencionais da linguagem *Python*, desconsiderando as regras arquiteturais estritas e as especificidades exigidas pelo MASPY. A aplicação direta de LLMs não supervisionados para gerar sistemas resulta em falhas topológicas e sintáticas. Neste escopo, a consistência topológica refere-se à correta inicialização, segregação e conexão dos componentes arquiteturais do sistema. Como o *framework* exige uma separação de heranças, um código topologicamente consistente garante que instâncias da classe *Agent* jamais sejam confundidas ou agrupadas indevidamente com instâncias de *Environment* ou *Channel* no momento do registro da simulação.

Por operarem com base em predições probabilísticas — um fenômeno frequentemente descrito como "papagaios estocásticos" [Bender et al. 2021] — os modelos de linguagem perdem o contexto de suas ações ao longo de múltiplos turnos [Silva et al. 2025]. Na geração de código BDI, isso se traduz em violações de ciclo de vida e, criticamente, na "amnésia de contexto": o LLM gera planos deliberativos baseados em crenças específicas, mas esquece de inicializar essas mesmas crenças nos construtores dos agentes, resultando em execuções paralisadas e instâncias em estado perpétuo de espera.

Para mitigar a imprevisibilidade desses modelos na geração de código, este trabalho apresenta o MASPY-LLM. Trata-se de um pipeline gerador autônomo estruturado em três camadas: (a) Injeção de Contexto com *Retrieval-Augmented Generation* (RAG) [Lewis et al. 2020] para fornecer a documentação oficial da ferramenta, (b) uma orquestração Ator-Crítico, que desacopla a geração criativa de uma revisão lógica rigorosa, e (c) *MaspySanitizer*, um módulo de higienização algorítmica que corrige dependências ausentes e erros de roteamento. Para avaliar a solução, o artigo apresenta um estudo de caso que instancia, de forma autônoma, um ecossistema de cooperação baseado no protocolo *contract-net* (leilões), validando a geração de cadeias de comunicação consistentes e livres de intervenção humana.

A motivação fundamental para o desenvolvimento deste projeto reside na necessidade de democratizar o acesso e transpor a íngreme curva de aprendizado associada à engenharia de *software* de SMAs. A modelagem estrutural de simulações complexas exige tempo e rigor sintático exaustivos por parte dos pesquisadores. Ao prover uma ferramenta que traduz linguagem natural em um código BDI consistente e com mitigações contra alucinações, o MASPY-LLM desloca o esforço humano da depuração de código para a concepção do design do ecossistema. Essa automação auxilia na prototipagem de negociações e interações sistêmicas, como também atua como um facilitador para futuras pesquisas aplicadas a domínios de grande escala.

O artigo está organizado da seguinte forma: a Seção 2 aborda a fundamentação e trabalhos relacionados; a Seção 3 detalha a arquitetura do MASPY-LLM; a Seção 4 discute os experimentos; e a Seção 5 apresenta as conclusões.

2. Fundamentação Teórica

Para a compreensão da arquitetura proposta neste artigo, esta seção aborda os conceitos fundamentais de SMA, o paradigma BDI, a ferramenta MASPY e os LLMs, ilustrando o estado da arte por meio de trabalhos recentes da literatura.

2.1. Sistemas Multiagente e a Arquitetura BDI

Sistemas Multiagente (SMA) coordenam entidades computacionais autônomas, proativas e interativas na resolução de problemas complexos e descentralizados [Wooldridge 2009]. Para modelar o comportamento destas de forma deliberativa (simulando processos de tomada de decisão), a comunidade de pesquisa utiliza amplamente o paradigma BDI [Cardoso and Ferrando 2021]. Esta arquitetura permite que os agentes representem seu conhecimento e raciocínio por meio de três componentes mentais: as crenças (*Beliefs*), que representam o conhecimento sobre o mundo; os objetivos que desejam alcançar (*Desires*); e os planos de ação com os quais se comprometem a realizá-los (*Intentions*) [Rao and Georgeff 1995].

2.2. Framework MASPY

O MASPY é um *framework Python* de código aberto para gerenciamento de SMAs baseados no paradigma BDI [Mellado et al. 2025]. Sua arquitetura fundamenta-se em quatro classes principais: `Agent`, `Environment`, `Communication` e `Admin`. O *framework* utiliza canais isolados e tem demonstrado robustez ao ser integrado como o motor lógico de simulações físicas e gráficas complexas, a exemplo do controle de tráfego urbano no SUMO e Traffic3D [Neres et al. 2025].

2.3. Large Language Models

Os LLMs [Ma et al. 2023] são redes neurais avançadas no campo do Processamento de Linguagem Natural [Wu et al. 2024]. Elas operam com unidades fundamentais chamadas *tokens* e aprendem a prever, iterativamente, qual sequência textual tem maior probabilidade de seguir um dado *prompt* de entrada [Shanahan 2024]. Apesar do sucesso na geração de texto, a dependência dessa predição probabilística acarreta limitações lógicas.

2.4. Sistemas Agênticos

A integração da capacidade de geração e de raciocínio dos LLMs às estruturas clássicas de agentes impulsionou o surgimento do que a literatura define como "sistemas agênticos" [Guo et al. 2024]. Diferentemente de assistentes virtuais puramente reativos, agentes baseados em LLMs são entidades proativas equipadas com arquiteturas cognitivas — que contêm módulos de memória, planejamento e percepção — que lhes permitem interpretar linguagem natural, utilizar ferramentas externas e atuar colaborativamente em um ambiente. A aplicação desses sistemas agênticos divide-se tradicionalmente em duas frentes, conforme delineado em taxonomias recentes da área [Guo et al. 2024]: agentes especialistas voltados à resolução de tarefas complexas de engenharia e agentes gerativos voltados à "simulação de mundo", usados para observar interações sociais, econômicas e de negociação descentralizada.

2.5. Trabalhos Relacionados

No domínio da resolução de tarefas, as arquiteturas cognitivas têm utilizado LLMs para atuarem como engenheiros de software autônomos. [Silva et al. 2025] propuseram um SMA baseado no framework LangGraph, no qual um orquestrador distribui subtarefas a agentes especialistas na geração de código. Abordagens similares, como o ChatDev [Qian et al. 2024], estruturam a comunicação em fases de design e de codificação. Contudo, o foco destas abordagens recai sobre a geração de código genérico e de scripts simples, sem abordar o desafio arquitetural de programar agentes sob o paradigma BDI.

Mais próximo do paradigma BDI, [Ciatto et al. 2025] investigam o uso de GenAI para a geração dinâmica de planos em agentes AgentSpeak(L). Os autores propõem a integração de uma *Plan Generation Procedure* (PGP) à arquitetura BDI, permitindo que o agente consulte um LLM quando precisa produzir novos planos para atingir determinados objetivos. O processo envolve a conversão do estado cognitivo e operacional do agente, incluindo crenças, objetivos, ações disponíveis e subplanos existentes, em um *prompt* estruturado, seguido da extração dos planos gerados para sua incorporação à biblioteca de planos do agente. O foco recai sobre a geração de planos em tempo de execução para agentes já especificados, enquanto a proposta deste trabalho atua em outro nível: o MASPY-LLM utiliza LLMs para gerar e revisar o código estrutural de sistemas multiagentes BDI em MASPY, combinando RAG, orquestração ator-crítico e sanitização algorítmica para produzir geração de código completa e reduzir erros de dependência, roteamento e ciclo de vida antes da execução do sistema.

Em simulação e negociação, [Junior et al. 2025] utilizaram o LangGraph e o LM Studio para modelar um cenário econômico competitivo baseado no jogo de RPG Gorim. Apesar do sistema conseguir instanciar os perfis de empresário e agricultor, os autores documentaram falhas severas de raciocínio decorrentes do fenômeno dos "papagaios estocásticos" [Bender et al. 2021]. O agente LLM frequentemente perdia o contexto temporal do ciclo de negociação, assumindo que a compra havia sido concluída logo após o envio de uma oferta e ignorando a necessidade estrutural de aguardar a resposta.

Esse fenômeno de amnésia e de perda de contexto na orquestração de diálogos é exatamente a limitação que este trabalho busca mitigar. Enquanto abordagens como a de [Junior et al. 2025] contornam o problema limitando os turnos de negociação e impondo bloqueios externos no ambiente, o MASPY-LLM atua na base estrutural. A arquitetura proposta utiliza a IA não como um personagem do diálogo BDI, mas como o programador autônomo do framework. Por meio da injeção de contexto e da sanitização algorítmica, o pipeline corrige deterministicamente as dependências ausentes e os erros de ciclo de vida mapeados na literatura antes da compilação do código, gerando sistemas complexos e topologicamente mais seguros a alucinações de controle.

3. Arquitetura do MASPY-LLM

O objetivo central do MASPY-LLM é traduzir instruções em linguagem natural em código MASPY estritamente funcional, mitigando a imprevisibilidade intrínseca dos LLMs. Para alcançar a geração de código determinístico e topologicamente seguro — ou seja, sem erros de conexão entre instâncias de *Agent, Environment* e *Channel* — a arquitetura foi concebida com um pipeline dividido em três camadas principais de processamento

sequencial, como ilustrado na Figura 1: injeção de contexto dinâmico, orquestração multimodelo e sanitização estrutural.

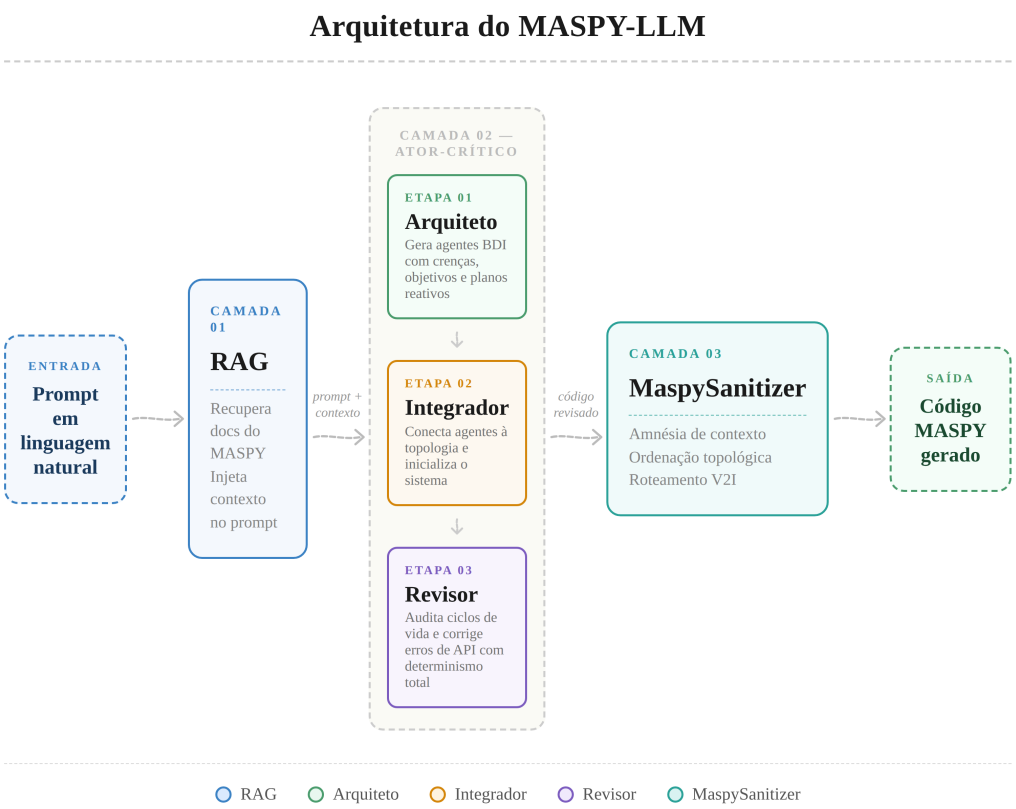

Figura 1. Visão geral da arquitetura do MASPY-LLM.

3.1. Injeção de Contexto via RAG e Mapeamento de Domínio

Para contornar a ausência de conhecimento pré-treinado dos LLMs sobre o framework MASPY, estruturou-se um pipeline de RAG utilizando o ecossistema *LangChain* e o banco de dados vetorial ChromaDB. A documentação do MASPY, as regras arquiteturais BDI e os exemplos de sintaxe validada foram convertidos em *embeddings* utilizando o modelo multilíngue bge-m3 [Chen et al. 2024a]. Este modelo foi selecionado por sua eficácia em conectar intenções do usuário, formuladas em português, a estruturas de código-fonte documentadas em inglês.

Durante a fase de recuperação de contexto, o sistema analisa a requisição do usuário e executa um mapeamento léxico do domínio, conforme ilustrado no Código 1. A detecção de entidades fundamentais — como *Agent, Communication, Belief, Goal* e *Environment* — não depende apenas da similaridade vetorial, mas de uma dupla verificação heurística. Isso permite que o sistema resgate exclusivamente os fragmentos mais relevantes do acervo, injetando-os dinamicamente no prompt de sistema e fornecendo uma base factual estrita que limita o espaço de alucinação inicial.

```
MASPY_ENTITIES = {
    "Communication": {
        "keywords": ["enviar", "receber", "negocia", "proposta", ...],
        "regex": [r"self\.send\s*\(", r"achieve\s*,\s*Goal", r"tell\s*,\s*Belief"]
    },
    "Belief_Goal": {
        "keywords": ["crenca", "belief", "objetivo", "goal", ...],
        "regex": [r"Belief\(", r"Goal\(", r"add_belief", r"add_goal"]
    },
```

```
    # ...
}

def detectar_intencao_na_pergunta(pergunta: str) -> list:
    """Estima quais entidades MASPY o usuário precisa, a partir do texto da query."""
    intencoes = set()
    for entity, rules in MASPY_ENTITIES.items():
        for kw in rules["keywords"]:
            if kw in pergunta.lower():
                intencoes.add(entity)
                break
    return list(intencoes)
```

Código 1. Mapeamento léxico.

3.2. Orquestração Multi-Modelo com Arquitetura Ator-Crítico

A fase de geração de código foi desacoplada para operar sob uma arquitetura de debate interno (*Generator-Evaluator*), executada localmente por meio do framework `Ollama`. Conforme ilustrado na Figura 2, o pipeline é composto por 3 estágios sequenciais — Arquiteto, Integrador e Revisor — cada um responsável por uma camada distinta na construção do código BDI.

O primeiro e segundo estágios compõem o *Ator* (Gerador), que opera sob o modelo `qwen2.5-coder:7b`, parametrizado com baixa temperatura para permitir leve síntese criativa. Como mostra a Etapa 01 da Figura 2 o agente Arquiteto é responsável pela geração das classes BDI: ele cria os agentes e ambientes com planos reativos e define as estruturas de *Belief*, *Goal* e *Percept*, inicializando os construtores (`__init__`) com as crenças inaugurais.

A transição para a Etapa 02 ocorre quando o Arquiteto entrega o conjunto de classes ao agente Integrador, que possui função exclusivamente estrutural: instanciar os agentes, conectá-los à topologia por meio de `Admin().connect_to()` e montar o bloco `__main__`, responsável por inicializar os objetivos do sistema.

Trecho do Prompt do Agente Arquiteto

4. O paradoxo do receptor: O agente que recebe uma ordem não pode exigir um estado que ele ainda não tem no gatilho.
6. Construtores (`__init__`): - Se um plano @pl exige um Belief (ex: Belief("estado", "vermelho")), você deve adicionar essa crença no `__init__` do agente usando `self.add(Belief(...))`, senão o agente ficará `Idle` para sempre!

O terceiro estágio, representado na Etapa 03 da Figura 2, corresponde ao Revisor Sênior, um LLM secundário (`llama3`) parametrizado com temperatura zero para garantir total determinismo lógico. Ao receber as classes BDI combinadas com o bloco principal, o Crítico aplica uma heurística de auditoria focada exclusivamente em falhas críticas documentadas na literatura sobre SMAs.

Entre as verificações realizadas estão a análise do ciclo de vida dos agentes, a inserção apropriada do comando `self.stop_cycle()` quando necessário e a validação do uso correto da API de Ambientes em contraste com a de Agentes, garantindo que os agentes utilizem as operações `add`/`rm` e que os ambientes utilizem a operação `change`.

3.3. Higienização Algorítmica

O mecanismo de geração automática foi complementado pelo MaspySanitizer, um módulo Python estruturado segundo o padrão de projeto *Chain of Responsibility*. Posicionado como a última camada de validação, o módulo atua por meio de uma análise baseada

Figura 2. Orquestração do MASPY-LLM.

em expressões regulares e aplica heurísticas corretivas destinadas a mitigar falhas recorrentes observadas no código produzido por LLMs.

A primeira heurística consiste na Prevenção de Amnésia de Contexto. Nesse processo, o sanitizador realiza uma varredura dos planos deliberativos em busca de dependências lógicas associadas a crenças e objetivos. Quando necessário, crenças inaugurais são automaticamente injetadas nos construtores. Os valores atribuídos a essas crenças são inferidos estaticamente a partir da pré-condição do plano ou, na ausência de contexto explícito, são inicializados com o tipo neutro para aguardar a percepção dinâmica do ambiente. De maneira análoga, caso o plano exija uma motivação interna, objetivos iniciais são injetados, garantindo a existência dos gatilhos necessários para o início da execução dos agentes e evitando estados permanentes de repouso.

A segunda heurística corresponde à Ordenação Topológica. Por meio da análise das assinaturas de herança dos objetos instanciados, o módulo reorganiza automaticamente os parâmetros da função `Admin().connect_to()`, assegurando a segregação entre instâncias derivadas de `Agent` e componentes de infraestrutura, como `Environment` e `Channel`. Essa validação previne exceções de tipagem durante a fase de inicialização do sistema.

Por fim, o módulo implementa um mecanismo de Roteamento Reflexivo (*Vehicle-to-Infrastructure*) [Park et al. 2023]. Essa heurística intercepta chamadas diretas a métodos de ambiente realizadas incorretamente por agentes e as converte dinamicamente para a sintaxe nativa de atuação encadeada do *framework*, como em `self.action(Ambiente).metodo()`. Dessa forma, preserva-se a conformidade arquitetural do código gerado sem exigir intervenções manuais do desenvolvedor.

4. Experimentos e Avaliação

Para avaliar a eficácia da arquitetura MASPY-LLM, adotou-se um protocolo de validação empírica cujo critério primário de sucesso é a capacidade do pipeline de converter instruções em linguagem natural em código funcional. A métrica de avaliação é dupla: ausência de exceções em tempo de execução *(Traceback Exceptions)* e consistência topológica das instâncias geradas. Os experimentos foram realizados em uma máquina equipada com processador AMD Ryzen 7 5700U with Radeon Graphics, 1.80 GHz e 32GB de RAM e sistema operacional Windows 11. O código-fonte completo do pipeline MASPY-LLM, incluindo os trechos apresentados nesta seção, encontra-se disponível em: `https://github.com/nataliamendesgoes/IC`.

4.1. Configuração do Cenário

O protocolo *Contract-Net* [Smith 1988] foi selecionado como *baseline* inicial por ser amplamente consolidado na literatura de agentes, exigindo encadeamento assíncrono de mensagens e troca de representações mentais. O escopo foca no `Initiator` delegando o objetivo `fazer_proposta` aos participantes, testando o *pipeline* simultaneamente contra amnésia de contexto e falhas de ordenação topológica.

Ao longo do experimento, foram conduzidas múltiplas iterações de *prompts*, variando de descrições estruturadas com exemplos de código a formulações puramente textuais. O objetivo foi identificar o formato de entrada que melhor orienta o pipeline. O *prompt* que produziu o artefato mais próximo do comportamento esperado foi estruturado da seguinte maneira:

Prompt de Avaliação

Crie um cenário de leilão com três agentes: 'Initiator', 'ParticipanteA' e 'ParticipanteB'. O Initiator inicia com o objetivo *iniciar_leilao*. Ao executar esse plano, o Initiator deve enviar ao ParticipanteA e ao ParticipanteB o objetivo *fazer_proposta*. Cada participante deve reagir ao objetivo *fazer_proposta*, gerar um valor aleatório entre 1 e 100, imprimir o valor gerado e enviar esse valor ao Initiator como crença 'proposta'. O Initiator deve reagir ao recebimento da crença 'proposta', imprimir quem enviou e qual foi o valor recebido. Não precisa criar canal.

Este escopo testa os limites do pipeline de geração, pois demanda a mitigação simultânea das falhas recorrentes documentadas na literatura: a amnésia de contexto, o encerramento prematuro de threads e os erros de ordenamento topológico na inicialização.

4.2. Artefato Gerado e Análise Inicial

A análise do código gerado, disponível no Código 2, demonstra claramente a atuação das camadas de contenção do MASPY-LLM contra as alucinações intrínsecas aos LLMs, frequentemente descritas na literatura como "papagaios estocásticos".

```
from maspy import *
import random

class Initiator(Agent):
    def __init__(self, agt_name=None):
        super().__init__(agt_name)
        self.add(Belief("estado", "iniciar_leilao"))
        self.add(Goal("iniciar_leilao"))

    @pl(gain, Goal("iniciar_leilao"))
    def iniciar_leilao(self, src, *args):
        self.print(f"Iniciando leilão")
        self.send("ParticipanteA_1", achieve, Goal("fazer_proposta"))
        self.send("ParticipanteB_1", achieve, Goal("fazer_proposta"))

    @pl(gain, Belief("proposta", Any), Goal("iniciar_leilao"))
    def receber_proposta(self, src, valor, *args):
        self.print(f"Recebido valor {valor} de {src}")
        if not hasattr(self, 'melhor_proposta') or valor > self.melhor_proposta:
            self.melhor_proposta = valor
            self.print(f"Novo melhor valor: {self.melhor_proposta}")

class ParticipanteA(Agent):
    def __init__(self, agt_name=None):
        super().__init__(agt_name)
        self.add(Belief("estado", "fazer_proposta"))
        self.add(Goal("fazer_proposta"))

    @pl(gain, Goal("fazer_proposta"))
    def fazer_proposta(self, src, *args):
        valor = random.randint(1, 100)
        self.print(f"ParticipanteA fez proposta: {valor}")
        self.send(src, tell, Belief("proposta", valor))

class ParticipanteB(Agent):
    def __init__(self, agt_name=None):
```

```
        super().__init__(agt_name)
        self.add(Belief("estado", "fazer_proposta"))
        self.add(Goal("fazer_proposta"))

    @pl(gain, Goal("fazer_proposta"))
    def fazer_proposa(self, src, *args):
        valor = random.randint(1, 100)
        self.print(f"ParticipanteB fez proposta: {valor}")
        self.send(src, tell, Belief("proposta", valor))

if __name__ == "__main__":
    Admin().console_settings(True)
    initiator = Initiator("Initiator")
    participante_a = ParticipanteA("ParticipanteA")
    participante_b = ParticipanteB("ParticipanteB")
    Admin().connect_to([initiator, participante_a, participante_b], [Channel()])
    initiator.stop_cycle()
    Admin().start_system()
```

Código 2. Artefato gerado por MASPY-LLM.

A análise do log do sistema mostra que o leiloeiro (*Initiator*) disparou corretamente o plano inicial e delegou o objetivo `fazer_proposta` aos participantes via canal. Consequentemente, os participantes geraram e transmitiram suas propostas. No entanto, o Initiator entrou em estado de repouso perpétuo (*Idle*) sem processar nenhuma proposta recebida, o que indica que o objetivo central do protocolo não foi alcançado.

4.3. Erros Identificados

A depuração do código gerado revelou categorias de falha que o pipeline não conseguiu corrigir de forma consistente, conforme Tabela 1. A partir destes dados, três falhas estruturais destacam-se pela complexidade de sua resolução autônoma:

Tabela 1. Erros identificados nas gerações do pipeline MASPY-LLM.

Erro Identificado	Categoria	Corrigido pelo pipeline?
Instanciação de Ambiente (classe inexistente)	Alucinação de API	Não
Tipo `fazer_proposta` no `@pl` e no método	Semântico recorrente	Não
`stop_cycle()` antes de `start_system()`	Ciclo de vida	Não
`Environment` na lista de agentes do `connect_to`	Topológico	Parcialmente
Condição composta incorreta no `@pl` do Initiator	Semântico	Não
Nomes duplicados nos agentes instanciados	Topológico	Não

- Amnésia de Plano Reativo: O plano receber_proposta do *Initiator* foi gerado com uma condição composta inexequível: @pl(gain, Belief(proposta, Any), Goal(iniciar_leilao). Como o objetivo iniciar_leilao já havia sido consumido no momento em que a crença proposta chegou pelo canal, a condição nunca pôde ser satisfeita. O agente Revisor não detectou essa incompatibilidade temporal estrita entre a precondição do plano e o ciclo de vida dos objetivos.

- Roteamento para Agente Inexistente: As mensagens enviadas durante o construtor (`__init__`) dos participantes, antes da inicialização completa do sistema, foram direcionadas a `self_1`, uma entidade não registrada. Isso ocorreu, pois o argumento src no contexto do construtor referenciava o próprio agente antes de sua conexão ao canal, uma falha topológica não interceptada pelo *MaspySanitizer*.
- Desvio Tipográfico Persistente: O método fazer_proposa do ParticipanteB manteve um erro ortográfico no nome do decorador `@pl`, dissociando o gatilho do plano de sua respectiva função. O *MaspySanitizer* não realizou a validação de consistência semântica entre as assinaturas, o que resultou na reprodução idêntica do erro em múltiplas iterações de geração independentes.

4.4. Discussão dos Resultados

Os experimentos demonstram que MASPY-LLM resolve os erros topológicos mais críticos, que o artefato foi compilado sem exceções de tipagem, que os agentes foram registrados adequadamente e que o fluxo inicial do protocolo foi executado com êxito. Destaca-se que a correção topológica para instâncias de ambiente na função de conexão atuou de forma parcial: embora o `MaspySanitizer` tenha segregado o objeto com sucesso para evitar a falha crítica de compilação, o LLM continuou gerando a chamada inicial de forma incorreta nas sucessivas iterações, evidenciando uma forte tendência do modelo a ignorar restrições estruturais que fogem ao padrão Python convencional. Contudo, vulnerabilidades semânticas, como incompatibilidades temporais nas condições dos planos e roteamento prematuro na inicialização, permaneceram sem correção.

Estes resultados evidenciam uma distinção entre dois níveis de mitigação das alucinações. No primeiro nível, os erros arquiteturais, como quebras de herança topológica e instanciação de classes inexistentes, foram tratados com sucesso pelas camadas algorítmicas do pipeline. Em contraste, o segundo nível, os erros semânticos de fluxo de execução, permanece fora do escopo da pipeline estática atual. Conclui-se que, enquanto a ferramenta resolve de forma assertiva as falhas base mapeadas, a precisão do fluxo de regras de negócios ainda demanda uma solução mais robusta.

5. Conclusão

A programação de SMAs-BDI exige um rigor arquitetural e sintático que contrasta fortemente com a natureza probabilística dos LLMs. A aplicação não supervisionada destas IAs na geração de sistemas resulta em graves violações do ciclo de vida, quebras de topologia e paralisia dos agentes devido à amnésia de contexto. Para mitigar essa barreira técnica e reduzir a curva de aprendizado associada ao framework MASPY, este artigo apresentou o MASPY-LLM.

Os experimentos validaram a eficácia da abordagem proposta e a arquitetura híbrida, que combina injeção de contexto via RAG, uma orquestração Ator-Crítico para auditoria lógica e o módulo *MaspySanitizer* para higienização algorítmica, mostrou-se altamente capaz de converter intenções em linguagem natural em simulações estruturalmente viáveis. O pipeline conseguiu instanciar um ecossistema distribuído, intervindo de forma determinística para evitar exceções de tipagem na ordenação topológica e garantir a compilação bem-sucedida do código-base.

Contudo, a análise aprofundada dos resultados evidenciou uma distinção na mitigação das alucinações. Enquanto o pipeline isola e resolve com precisão as falhas arqui-

teturais (como o registro de agentes e canais), as vulnerabilidades semânticas atreladas ao fluxo de execução (tais como incompatibilidades temporais nas precondições dos planos reativos e desvios tipográficos recorrentes) permanecem como limitações do sistema estático. Conclui-se que a versão atual da MASPY-LLM opera como um assistente de engenharia de software, abstraindo a complexidade da estruturação do código BDI e acelerando a prototipagem, mas a validação final do fluxo de regras de negócios ainda requer supervisão humana pontual.

Para trabalhos futuros, uma abordagem consistirá na evolução da arquitetura estática para um modelo dinâmico de validação em malha fechada (*self-debugging*) [Chen et al. 2024b] e também na condução de análises de desempenho por meio da substituição e do refinamento dos LLMs executados no Ollama.

Este trabalho tem financiamento parcial do projeto 444568/2024-7 (CNPq/MCTI/FNDCT Nº 22/2024, *Programa Conhecimento Brasil – Apoio a Projetos em Rede com Pesquisadores Brasileiros no Exterior*). Agradecemos à Universidade Tecnológica Federal do Paraná (UTFPR), *Campus* Ponta Grossa, pela infraestrutura laboratorial e pela concessão da bolsa de Iniciação Científica (PIBIC/PIBITI-UTFPR), bem como à Fundação Araucária e ao CNPq pelo apoio e fomento financeiro concedidos ao projeto no qual este trabalho está inserido.

Referências

Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 610–623.

Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA.

Cardoso, R. C. and Ferrando, A. (2021). A review of agent-based programming for multi-agent systems. *Computers*, 10(2):16.

Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D., and Liu, Z. (2024a). M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2318–2335, Bangkok, Thailand. Association for Computational Linguistics.

Chen, X., Lin, M., Schärli, N., and Zhou, D. (2024b). Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pages 8746–8825.

Ciatto, G., Aguzzi, G., Battistini, R., Baiardi, M., Burattini, S., and Ricci, A. (2025). Exploiting genai for plan generation in bdi agents. In *ECAI 2025*, volume 413 of *Frontiers in Artificial Intelligence and Applications*, pages 3495–3502. IOS Press.

Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., and Zhang, X. (2024). Large language model based multi-agents: A survey of progress and challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 8048–8057. International Joint Conferences on Artificial Intelligence Organization. Survey Track.

Junior, U. G., Born, M. B., Santos, A. C., Grossmann, R. B., Facklamm, J. V., de Castilhos, V. A., Alves, B. C., and de Aguiar, M. S. (2025). Sistemas multiagente e large language model: estudo de caso utilizando as ferramentas LM Studio e LangGraph. In *Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações (WESAAC)*, pages 250–261. SBC.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in neural information processing systems*, 33:9459–9474.

Ma, X., Fang, G., and Wang, X. (2023). LLM-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720.

Mellado, A. L. L., Borges, A. P., and Alves, G. V. (2025). MASPY: A Python-based framework for developing BDI multi-agent systems. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Computational Social Science: The PAAMS Collection*, pages 216–227. Springer-Verlag, Berlin, Heidelberg.

Neres, G. G., Cardoso, R. C., Borges, A. P., and Alves, G. V. (2025). Integração do SUMO e Traffic3D com o framework de agentes MASPY. In *Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações (WESAAC)*, pages 71–78. SBC.

Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., et al. (2024). Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 15174–15186.

Rao, A. S. and Georgeff, M. (1995). BDI Agents: From Theory to Practice. In *Proc. 1st Int. Conf. Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA.

Shanahan, M. (2024). Talking about large language models. *Communications of the ACM*, 67(2):68–79.

Silva, E., Santos, F. A., Thompson, P., and dos Reis, J. C. (2025). LLM-powered conversational multi-agent cognitive system for collaborative task solving. In *Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações (WESAAC)*, pages 59–70. SBC.

Smith, R. G. (1988). The contract net protocol: High-level communication and control in a distributed problem solver. In *Readings in distributed artificial intelligence*, pages 357–366. Elsevier.

Wooldridge, M. (2009). *An introduction to multiagent systems*. John wiley & sons.

Wu, L., Zheng, Z., Qiu, Z., Wang, H., Gu, H., Shen, T., Qin, C., Zhu, C., Zhu, H., Liu, Q., et al. (2024). A survey on large language models for recommendation. *World Wide Web*, 27(5):60.