AGENT BUILDER: UM FRAMEWORK PARA ENGENHARIA DE PROMPT E DESENVOLVIMENTO DE AGENTES USANDO MODELOS DE LINGUAGEM DE GRANDE ESCALA *

Luan Daros[†] Elder Francisco Fontana Bernardi[‡]

16 de dezembro de 2024

Resumo

Este trabalho apresenta o Agent Builder, um framework em TypeScript para desenvolvimento de aplicações baseadas em Modelos de Linguagem de Grande Escala (LLMs). O framework visa simplificar a integração com diferentes provedores de LLMs, oferecendo uma interface unificada e um conjunto de ferramentas para a construção de agentes inteligentes. O framework suporta saídas estruturadas, como JSON, utilizando schemas com validação. Ele permite a execução de funções externas e oferece flexibilidade por meio de interfaces abstratas. Comparado ao Langchain, o Agent Builder apresenta desempenho similar em acurácia, com vantagens em modularidade, extensibilidade e suporte à internacionalização. O código-fonte está disponível no GitHub sob licença MIT.

Palavras-chaves: Modelos de Linguagem de Grande Escala; Framework; TypeScript; Processamento de Linguagem Natural; Agentes Inteligentes.

*Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, na cidade de Passo Fundo, em 2024.

[†]Aluno do curso de Ciência da Computação, no Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense de Passo Fundo

[‡]Orientador, professor do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense de Passo Fundo.

1 INTRODUÇÃO

Modelos de Linguagem de Grande Escala (LLMs) têm demonstrado um potencial transformador em diversas áreas, desde a geração de texto criativo até a resolução de problemas complexos. Com sua capacidade de compreender e gerar linguagem natural, LLMs são amplamente aplicados em setores como medicina, direito, marketing e educação, promovendo avanços significativos em produtividade e eficiência (WEBER, 2024). Entretanto, a utilização eficaz desses modelos em aplicações práticas requer soluções que simplifiquem sua integração e forneçam mecanismos robustos para lidar com a complexidade inerente à interação com LLMs.

Estudos recentes indicam que LLMs, quando utilizados como componentes de software, podem substituir funções codificadas tradicionalmente e habilitar novos casos de uso. Por exemplo, frameworks baseados em LLMs têm sido aplicados na automação de tarefas, desenvolvimento de sistemas de diálogo e controle de sistemas complexos (JR et al., 2024; WEBER, 2024). No entanto, desafios como o custo de invocação, dependência de serviços externos e a necessidade de engenharia de prompts específicos ainda limitam o uso prático desses modelos (WEBER, 2024).

Pesquisas no campo da engenharia de sistemas têm destacado a importância de taxonomias para a compreensão e categorização de componentes baseados em LLMs, permitindo o desenvolvimento de arquiteturas modulares e reutilizáveis (WEBER, 2024). Além disso, frameworks especializados, como os apresentados em estudos de caso recentes, comprovam que a abstração de complexidades técnicas pode acelerar a adoção de LLMs em cenários industriais e técnicos (JR et al., 2024).

Este trabalho apresenta o Agent Builder, um framework em TypeScript projetado para facilitar o desenvolvimento de aplicações inteligentes baseadas em LLMs. O Agent Builder abstrai a complexidade da integração com diferentes provedores de LLMs, oferecendo uma interface unificada e um conjunto de ferramentas para construir agentes conversacionais, automatizar tarefas e integrar LLMs em fluxos de trabalho existentes. O framework promove a modularidade, extensibilidade e reutilização de código, permitindo que desenvolvedores se concentrem na lógica de suas aplicações, em vez dos detalhes da integração com os LLMs.

O objetivo geral deste trabalho é apresentar o Agent Builder e suas principais funcionalidades, demonstrando como ele simplifica o desenvolvimento de aplicações baseadas em LLMs.

Este trabalho está organizado da seguinte forma: na Seção 2, é apresentado o referencial teórico que fundamenta o estudo; na Seção 3, são discutidos os trabalhos relacionados. A Seção 4 descreve a arquitetura proposta e as funcionalidades desenvolvidas, enquanto a Seção 5 aborda um estudo de caso prático, avaliando o desempenho e a usabilidade do framework. Por fim, a Seção 6 reúne as conclusões e direções para trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 PROCESSAMENTO DE LINGUAGEM NATURAL

O Processamento de Linguagem Natural (NLP, do inglês *Natural Language Processing*) é uma subárea da inteligência artificial que se concentra na interação entre compu-

tadores e humanos por meio da linguagem natural. Segundo Nadkarni, Ohno-Machado e Chapman (NADKARNI; OHNO-MACHADO; CHAPMAN, 2011), o NLP combina princípios de inteligência artificial e linguística para permitir que os computadores compreendam e manipulem a linguagem humana de maneira significativa.

2.1.1 Principais tarefas de NLP

- **Tokenização**: Divisão do texto em unidades menores, como palavras ou frases.
- **Análise Sintática**: Estudo da estrutura gramatical das frases.
- **Reconhecimento de Entidades Nomeadas (NER)**: Identificação de entidades específicas, como nomes de pessoas, organizações e locais.
- **Análise de Sentimentos**: Determinação da polaridade emocional de um texto.
- **Tradução Automática**: Conversão de texto de um idioma para outro.
- **Compreensão de Leitura e *Question Answering***: Capacidade de entender o texto e responder a perguntas com base nele.

2.1.2 NLP estatístico

Com o crescimento exponencial dos dados textuais e a complexidade inerente da linguagem natural, os métodos tradicionais baseados em regras manuais tornaram-se inadequados. Segundo Nadkarni, Ohno-Machado e Chapman (NADKARNI; OHNO-MACHADO; CHAPMAN, 2011), a vasta dimensão, a natureza irrestrita e a ambiguidade da linguagem natural levaram à superação das abordagens baseadas em regras manuais, culminando na ascensão do NLP estatístico durante a década de 1980. Essa abordagem utiliza técnicas de aprendizado de máquina para criar regras probabilísticas, que são mais robustas e adaptáveis. Ao aprender com grandes volumes de dados reais, esses métodos são capazes de capturar os casos mais comuns, melhorando a precisão e degradando de forma mais graciosa com entradas desconhecidas ou errôneas.

2.2 APRENDIZADO DE MÁQUINA

O aprendizado de máquina é uma área da inteligência artificial cujo objetivo é desenvolver técnicas computacionais que permitam a aquisição automática de conhecimento. Segundo Monard e Baranauskas (MONARD; BARANAUSKAS, 2003), um sistema de aprendizado é um programa de computador que toma decisões baseadas em experiências acumuladas por meio da solução bem-sucedida de problemas anteriores. Esses sistemas possuem características particulares e comuns que possibilitam sua classificação quanto à linguagem de descrição, modo, paradigma e forma de aprendizado utilizado.

2.3 MODELOS DE LINGUAGEM DE GRANDE ESCALA

Os modelos de linguagem de grande escala têm revolucionado a forma como interagimos com a tecnologia, permitindo a geração de textos complexos e a realização de diversas tarefas de processamento de linguagem natural (NLP). Um dos principais avanços nesse campo foi a introdução da arquitetura Transformer, que substitui os tradicionais modelos recorrentes e convolucionais por mecanismos de atenção, conforme detalhado por Vaswani et al. (VASWANI et al., 2017).

2.4 ARQUITETURA TRANSFORMER

A arquitetura Transformer, introduzida por Vaswani et al. (VASWANI et al., 2017), revolucionou o campo do NLP ao substituir as abordagens baseadas em redes recorrentes (RNNs) e convolucionais (CNNs) por um mecanismo de atenção auto-regressiva, que melhora a eficiência e a capacidade de modelar dependências globais em sequências. Essa abordagem se mostrou altamente paralelizável, acelerando o treinamento e tornando-a ideal para lidar com grandes volumes de dados e tarefas complexas.

O Transformer é composto por duas partes principais: codificador (encoder) e decodificador (decoder), que trabalham em conjunto em tarefas de tradução ou geração de texto.

Os LLMs, como o GPT-3 e o BERT, baseados na arquitetura Transformer, têm mostrado capacidades impressionantes em tarefas de *NLP*. Eles são capazes de realizar aprendizado de poucos exemplos (*few-shot learning*) e gerar texto com coerência e contexto avançado. Além disso, a capacidade de paralelização do Transformer permite um treinamento mais rápido e eficiente, tornando possível o uso de grandes volumes de dados para melhorar o desempenho do modelo.

2.5 ENGENHARIA DE PROMPT

A engenharia de prompt é uma habilidade essencial para interagir eficazmente com modelos de linguagem grande (*LLMs*), como o ChatGPT. Segundo White et al. (WHITE et al., 2023), os prompts são instruções fornecidas a um *LLM* para impor regras, automatizar processos e garantir qualidades específicas na saída gerada. Eles podem ser considerados uma forma de programação que personaliza as interações e saídas dos *LLMs*.

2.6 AGENTES DE INTELIGÊNCIA ARTIFICIAL

Agentes de inteligência artificial (IA) são sistemas capazes de utilizar *LLMs* para raciocinar sobre problemas, criar planos de ação e executar essas ações com a ajuda de diversas ferramentas. Esses agentes são compostos por módulos principais, incluindo um núcleo de agente, módulo de memória, ferramentas e módulo de planejamento (NVIDIA, 2023).

Agentes de IA desempenham um papel crucial em diversas aplicações modernas, desde a geração de código até a execução de tarefas complexas de raciocínio. Conforme Liu et al. (LIU et al., 2023), a combinação de múltiplos agentes *LLM* pode melhorar significativamente o desempenho em tarefas complexas por meio da colaboração e da otimização dinâmica dos agentes.

2.7 DESIGN PATTERNS

Design patterns são soluções reutilizáveis para problemas comuns de design de software. Eles representam melhores práticas desenvolvidas por programadores experientes e são essenciais para criar sistemas robustos e flexíveis. Gamma et al. (GAMMA et al., 1994) categorizam esses padrões em três tipos principais:

- **Padrões Criacionais**: Envolvem a criação de objetos, permitindo maior flexibilidade e reutilização de código. Exemplos incluem *Singleton*, *Factory* e *Builder*;

- **Padrões Estruturais**: Lidam com a composição de classes e objetos para formar estruturas maiores e mais complexas. Exemplos incluem *Adapter*, *Composite* e *Decorator*;
- **Padrões Comportamentais**: Focam na interação e responsabilidade entre objetos. Exemplos incluem *Observer*, *Strategy* e *Command*.

2.8 TYPESCRIPT

O TypeScript é uma linguagem de programação desenvolvida pela Microsoft, que se baseia no JavaScript, mas com a adição de recursos de tipagem estática opcional e ferramentas avançadas para desenvolvimento. Essa linguagem é frequentemente utilizada para construir aplicações escaláveis e de grande porte, oferecendo uma experiência de desenvolvimento mais robusta e segura em comparação com o JavaScript puro.

Desde sua introdução, o TypeScript tornou-se uma das linguagens mais populares no ecossistema de desenvolvimento web. Conforme relatórios da comunidade, como o Stack Overflow Developer Survey (STACK OVERFLOW, 2023), o TypeScript é amplamente elogiado por sua robustez e facilidade de uso, sendo adotado em grandes projetos de código aberto e sistemas corporativos de alta complexidade.

3 TRABALHOS RELACIONADOS

LangChain é uma *framework* para desenvolvimento de aplicativos baseados em *LLMs*, conforme descrito por Topsakal (TOPSAKAL, 2023). A biblioteca oferece uma estrutura modular que permite a integração de *LLMs* com diversas ferramentas e fontes de dados, tornando o desenvolvimento de aplicações mais flexível e escalável. Entre as funcionalidades suportadas estão a recuperação de documentos, a geração de texto e a execução de tarefas complexas baseadas em *prompts*. Um dos principais benefícios do LangChain é sua capacidade de combinar múltiplas funcionalidades de *LLMs* em uma aplicação coesa, permitindo a automação de fluxos de trabalho e a personalização das interações com os usuários (LANGCHAIN, 2024).

O OpenRouter, por sua vez, é uma plataforma que oferece uma interface unificada para interagir com diversos modelos de linguagem de grande escala (*LLMs*). Seu principal objetivo é simplificar o acesso a múltiplos modelos de IA, permitindo que desenvolvedores integrem facilmente diferentes *LLMs* em suas aplicações. Uma de suas ferramentas mais relevantes é o *OpenRouter Runner*, um mecanismo de inferência monolítico que funciona como uma solução robusta para a implantação de diversos modelos de código aberto (OPENROUTER, 2024).

Além disso, o estudo de Weber (WEBER, 2024) apresenta uma taxonomia detalhada para a análise e descrição de aplicações integradas com *LLMs*. Esse trabalho destaca como componentes baseados em *LLMs* podem ser projetados para desempenhar tarefas específicas em sistemas de software. A taxonomia proposta facilita a compreensão das integrações de *LLMs*, abordando tanto os desafios quanto as soluções práticas para sua aplicação em diferentes contextos.

Outro trabalho relevante é o estudo de Lima Junior et al. (JR et al., 2024), que investiga o uso de *LLMs* na construção de casos de teste em engenharia de software. Os autores destacam o potencial dos *LLMs* para automatizar etapas importantes do processo

de teste, bem como os desafios encontrados, como custos de invocação e inconsistências nos resultados.

4 DESENVOLVIMENTO

4.1 FUNCIONALIDADES

O Agent Builder oferece um conjunto de funcionalidades que simplificam o desenvolvimento de aplicações baseadas em LLMs. Em resumo, o framework provê:

- **Abstração de Modelos de Linguagem:** Integração simplificada com diversos provedores de modelos (e.g., OpenAI, Google AI, OpenRouter) através de uma interface unificada.
- **Engenharia de Prompt Simplificada:** Construção de prompts dinâmicos e reutilizáveis com o sistema de templates, facilitando a experimentação e o gerenciamento de prompts.
- **Parsers:** Análise e estruturação da saída do LLM com suporte a diversos formatos de dados, incluindo JSON, através de parsers customizáveis.
- **Integração com Ferramentas Externas:** Execução de funções e serviços externos, permitindo que os agentes interajam com outros sistemas e realizem ações além da geração de texto.
- **Tratamento de Erros e Retentativas:** Mecanismos para lidar com erros de comunicação com os LLMs e outras falhas, com suporte a retentativas configuráveis, aumentando a robustez das aplicações.
- **Internacionalização:** Suporte a múltiplos idiomas, facilitando a adaptação do framework a diferentes contextos e usuários.
- **Monitoramento:** Registro detalhado das interações com o LLM e outras informações relevantes, auxiliando na depuração e no monitoramento do desempenho da aplicação.

4.2 ARQUITETURA

O Agent Builder foi projetado com uma arquitetura modular e extensível, centrada no conceito de um `Agent`. Cada `Agent` encapsula a lógica e os componentes necessários para interagir com um LLM e executar tarefas específicas. A arquitetura é composta por quatro módulos principais, que trabalham em conjunto para fornecer o fluxo de trabalho:

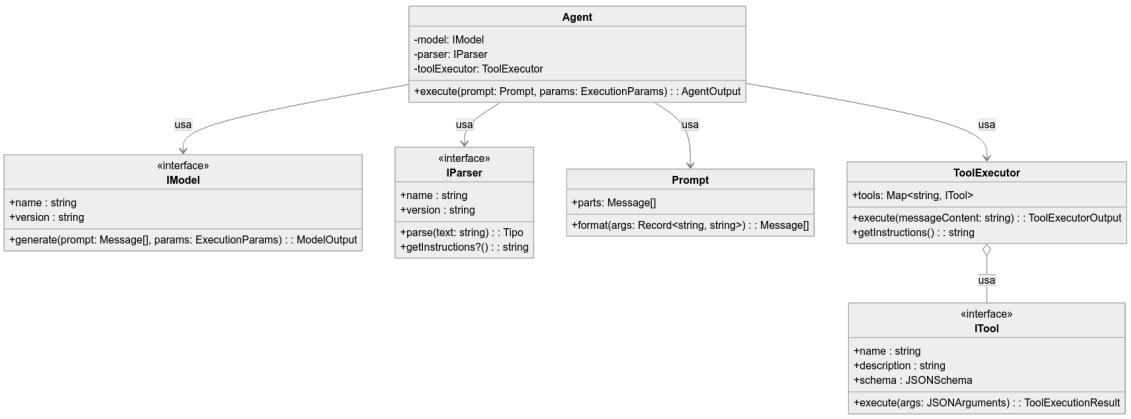
- **Model**: Este módulo representa a interface com o Modelo de Linguagem de Grande Escala (LLM). Ele abstrai as particularidades de diferentes provedores de LLMs, como OpenAI e Google, permitindo que o Agent Builder suporte uma variedade de modelos sem a necessidade de modificar a lógica principal. A interface `IModel` define os métodos de interação, como a geração de texto a partir de um prompt.
- **Template**: O módulo Template facilita a construção de prompts dinâmicos e reutilizáveis. Ele utiliza um sistema de templates que permite a inserção de variáveis e placeholders, tornando a geração de prompts mais eficiente e menos propensa a erros. A classe `Prompt` oferece métodos para formatar os templates com os valores desejados antes de enviá-los ao modelo.

- **Parser**: O módulo Parser é responsável por interpretar e estruturar a saída do LLM. Ele pode lidar com diferentes formatos de saída, desde texto simples até estruturas de dados complexas como JSON. A interface `IParser` define o método `parse`, que converte a saída textual do LLM em um formato estruturado. O Agent Builder inclui parsers predefinidos, como `PlainTextParser` e `JSONSchemaParser`, e permite a criação de parsers personalizados para atender às necessidades específicas de cada aplicação.
- **Tools**: Este módulo permite a integração de funcionalidades externas ao agente. As ferramentas podem ser qualquer função ou serviço que possa ser executado pelo framework, permitindo que o agente acesse informações externas, execute ações ou interaja com outros sistemas. A classe `ToolExecutor` gerencia e executa as ferramentas definidas pelo usuário, fornecendo uma interface consistente para acessá-las. A inclusão deste módulo possibilita a criação de agentes mais sofisticados e capazes de realizar tarefas complexas.

A interação entre esses módulos é orquestrada pela classe `Agent`, que recebe o modelo, o parser e executor de ferramentas como parâmetros. A classe `Agent` expõe o método `execute`, que aceita um `Prompt` ou uma lista de mensagens e retorna a saída do LLM, juntamente com os dados estruturados pelo parser e outras informações relevantes.

A Figura 1 apresenta um diagrama de classes da arquitetura do framework, ilustrando a interação entre os seus componentes principais.

Figura 1 – Arquitetura do Agent Builder



Fonte: **do autor**.

Essa arquitetura modular permite que os desenvolvedores personalizem e estendam o framework de acordo com suas necessidades, escolhendo os modelos, templates, parsers e ferramentas mais adequados para cada aplicação.

4.3 IMPLEMENTAÇÃO

O Agent Builder foi implementado utilizando TypeScript, escolha motivada pela sua tipagem estática, que contribui para um código mais robusto e manutenível, e por sua crescente popularidade no ecossistema JavaScript, facilitando a integração com outras

bibliotecas e frameworks, especialmente no desenvolvimento web. O framework foi projetado com foco em modularidade e extensibilidade, permitindo que desenvolvedores adicionem novos modelos, parsers e ferramentas de forma simples.

4.3.1 Tecnologias utilizadas

As principais tecnologias utilizadas na implementação do framework são:

- **TypeScript:** Linguagem principal de desenvolvimento, oferecendo tipagem estática e recursos modernos de JavaScript. (MICROSOFT, 2024)
- **mocha:** Framework para testes unitários, garantindo a qualidade e confiabilidade do código. (MOCHA.JS, 2024)
- **i18next:** Biblioteca para internacionalização, permitindo o suporte a múltiplos idiomas. A flexibilidade oferecida pelo i18next torna o framework acessível a um público mais amplo, adaptando-se às necessidades linguísticas de diferentes usuários. A utilização de arquivos JSON para armazenar as traduções simplifica o processo de adicionar novos idiomas e gerenciar as strings traduzidas. (I18NEXT, 2024)
- **Ajv (Another JSON Schema Validator):** Biblioteca para validação de schemas JSON, garantindo que os dados recebidos e enviados estejam em conformidade com a estrutura esperada. (AJV.ORG, 2024)

4.3.2 Integração com modelos de linguagem (Models)

Os Modelos de Linguagem de Grande Escala (LLMs) são o núcleo do Agent Builder. O framework fornece uma interface abstrata e implementações concretas para interagir com diferentes provedores e tipos de LLMs, permitindo flexibilidade e facilitando a adaptação a novas tecnologias.

A comunicação com os LLMs é padronizada pela interface `IModel`[1]. Essa interface define o contrato que todos os modelos de linguagem devem seguir, garantindo interoperabilidade e simplificando a troca de modelos sem a necessidade de modificar a lógica principal do agente. A interface `IModel` tipicamente define métodos para enviar prompts e receber respostas, gerenciar o contexto da conversa e configurar parâmetros específicos do modelo.

A interação com os LLMs é baseada em um sistema de mensagens, simulando uma conversa. Cada mensagem possui um atributo ‘role‘ que define sua função na conversa. Os seguintes papéis são suportados:

- **system:** Mensagens de sistema contêm instruções especiais para o LLM, como a persona que ele deve adotar ou o formato da resposta desejada.
- **user:** Mensagens do usuário representam a entrada do usuário para o LLM.
- **assistant:** Mensagens do assistente representam as respostas geradas pelo LLM.
- **tool:** Mensagens de ferramenta contêm os resultados da execução de ferramentas externas, permitindo que o LLM acesse informações e funcionalidades externas.

[1] Definido no arquivo: `src/core/interfaces.ts`.

Essa estrutura de mensagens proporciona uma maneira organizada e flexível de gerenciar o contexto da conversa e integrar informações de diferentes fontes.

O Agent Builder inclui implementações prontas para três provedores de LLMs populares:

- **OpenAI (`OpenAIModel`[2]):** Permite acesso aos modelos da família GPT.
- **Google AI (`GoogleAIModel`[3]):** Integração com os modelos da família Gemini.
- **OpenRouter (`OpenRouterModel`[4]):** Oferece acesso a uma ampla variedade de modelos disponíveis na plataforma OpenRouter, incluindo modelos de código aberto e proprietários.

A arquitetura do Agent Builder facilita a integração com outros LLMs. Desenvolvedores podem criar suas próprias implementações da interface `IModel` para suportar novos provedores ou modelos específicos. Essa extensibilidade garante que o framework possa se adaptar às mudanças rápidas no cenário de LLMs.

4.3.3 Templates (Prompt)

O Agent Builder oferece um sistema flexível de templates para a construção de prompts, permitindo a criação de prompts dinâmicos e reutilizáveis. A classe `Prompt`[5]. facilita a definição de templates com placeholders, que são substituídos por valores concretos no momento da execução.

Os placeholders em templates são definidos utilizando a notação de chaves duplas: `{{nome_do_parâmetro}}`. É utilizado expressões regulares para localizar os placeholders no template e substituí-los pelos valores correspondentes, fornecidos no momento da execução.

4.3.4 Saídas estruturadas (Parser)

A capacidade de gerar saídas estruturadas, como JSON, a partir de LLMs é fundamental para integrar esses modelos em pipelines de dados e aplicações que exigem formatos específicos. O Agent Builder facilita essa tarefa combinando engenharia de prompt, parsers customizáveis, um sistema de schemas embutido e o método Generate and Organize (G&O) (LI; RAMPRASAD; ZHANG, 2024).

Prompts bem elaborados são essenciais para orientar o LLM na geração de saídas no formato desejado. Instruções claras, exemplos e a definição explícita do schema de saída aumentam significativamente a chance de o LLM produzir o resultado esperado.

O Agent Builder incorpora um sistema de schemas que permite ao usuário definir a estrutura desejada da saída em JSON Schema. Ao fornecer um schema, o framework gera automaticamente instruções para o LLM, explicando o formato esperado. Essas instruções são combinadas com o prompt do usuário, fornecendo ao LLM um guia completo.

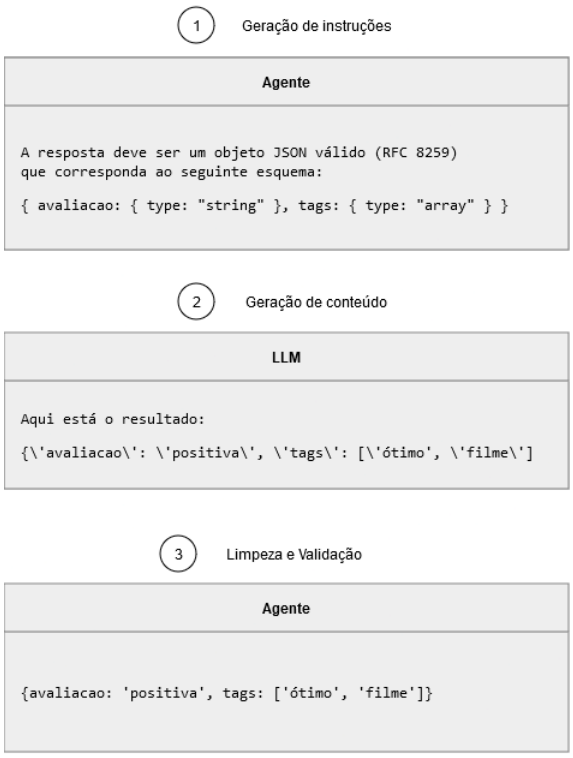
Após a geração do texto pelo LLM, o Agent Builder valida a saída contra o schema fornecido. Isso garante a consistência dos dados e permite que a aplicação trate eventuais

[2] Definido no arquivo: `src/models/open-ai/index.ts`.
[3] Definido no arquivo: `src/models/google-ai/index.ts`.
[4] Definido no arquivo: `src/models/open-router/index.ts`.
[5] Definido no arquivo: `src/core/prompt.ts`.

inconsistências. Se a saída não corresponder ao schema, o parser lança um erro, indicando os problemas encontrados.

A Figura 2 ilustra as etapas de uma geração estruturada.

Figura 2 – Exemplo de Parsing



Fonte: **do autor**.

Para cenários mais complexos ou LLMs com capacidade de contexto limitada, o método G&O (LI; RAMPRASAD; ZHANG, 2024) se mostra particularmente útil. O Agent Builder implementa esse método no agente especializado `JSONConversionAgent`[6]. Este agente permite que o LLM foque inicialmente na geração do conteúdo em linguagem natural, sem se preocupar com a formatação. Em seguida, um segundo prompt é utilizado para organizar a resposta na estrutura JSON desejada. Essa abordagem simplifica a tarefa do LLM, especialmente em modelos com janelas de contexto menores, resultando em saídas estruturadas mais precisas.

Essa combinação de engenharia de prompt, validação de schema e o método G&O oferece um mecanismo robusto e flexível para gerar e processar saídas estruturadas a partir de LLMs, simplificando a integração desses modelos em diversas aplicações.

4.3.5 Execução de funções externas (Tools)

A capacidade de executar funções externas, ou "ferramentas", enriquece significativamente as funcionalidades dos agentes no Agent Builder. As ferramentas permitem que os agentes interajam com sistemas externos, recuperem informações, façam chamadas a APIs,

[6] Definido no arquivo: `src/agents/json-agent.ts`.

executem comandos de sistema e, em geral, realizem ações que vão além das capacidades de um LLM.

A execução das ferramentas é gerenciada pela classe `ToolExecutor`[7]. Este componente desempenha três papéis principais:

1. **Instruções para o LLM:** O `ToolExecutor` gera instruções claras para o LLM, descrevendo as ferramentas disponíveis, seus parâmetros e como solicitá-las. Isso permite que o próprio LLM decida quando e como usar uma ferramenta.
2. **Detecção de Chamadas de Ferramentas:** O `ToolExecutor` analisa a saída do LLM, buscando por um padrão específico que indica a intenção de usar uma ferramenta. Esse padrão, por convenção, utiliza o formato '/tool {nome_da_ferramenta} /args {parâmetros_json}'. Uma expressão regular é usada para detectar esse padrão na saída do LLM.
3. **Execução e Retorno:** Ao identificar uma chamada de ferramenta, o `ToolExecutor` extrai o nome da ferramenta e os parâmetros (em formato JSON). Em seguida, executa a função correspondente à ferramenta, passando os parâmetros extraídos. O resultado da execução da ferramenta é então retornado ao LLM, permitindo que ele processe e utilize essa informação em suas respostas subsequentes.

As ferramentas são implementadas utilizando a interface `ITool`. Esta interface define a estrutura básica de uma ferramenta, exigindo a implementação dos seguintes elementos:

- **name:** Nome da ferramenta (string).
- **description:** Descrição da ferramenta (string).
- **parameters:** Um schema JSON Schema que define a estrutura dos parâmetros de entrada da ferramenta.
- **execute:** A função que implementa a lógica da ferramenta, recebendo os parâmetros de entrada e retornando o resultado.

A Figura 3 ilustra o fluxo de execução de um agente com o uso de ferramentas. O LLM gera uma resposta que inclui uma chamada de ferramenta. O `ToolExecutor` identifica a chamada, executa a ferramenta correspondente e retorna o resultado ao LLM para processamento posterior.

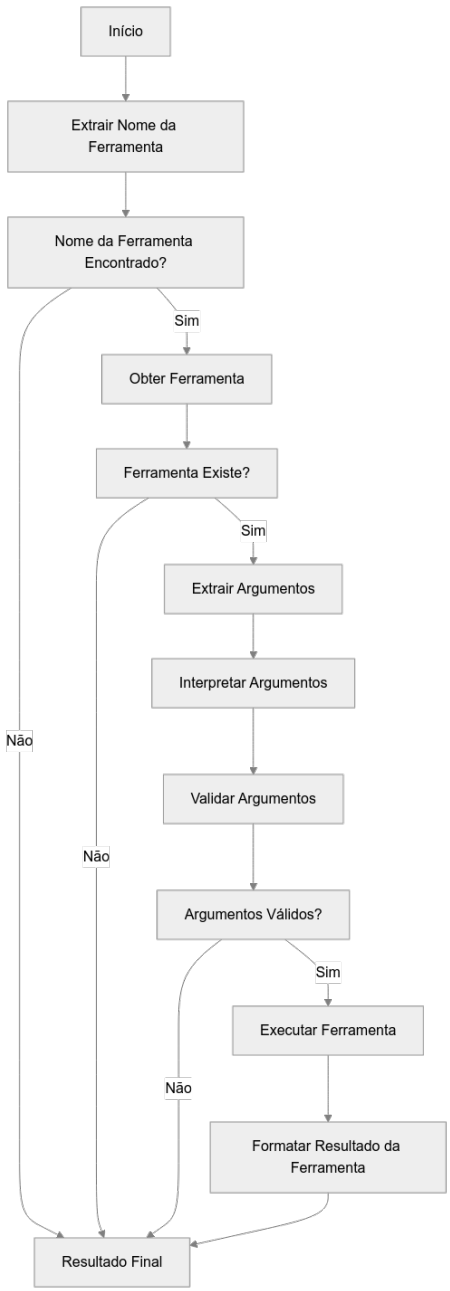
4.3.6 Fluxo de execução de um agente

O processo de execução de um agente se inicia com a inicialização e culmina com o retorno de um objeto contendo a saída estruturada do LLM, juntamente com metadados relevantes. A figura 4 ilustra o fluxograma desse processo.

As etapas detalhadas do fluxo de execução são as seguintes:

[7] Definido no arquivo: `src/core/tool-executor.ts`

Figura 3 – Fluxo de Execução com Ferramentas

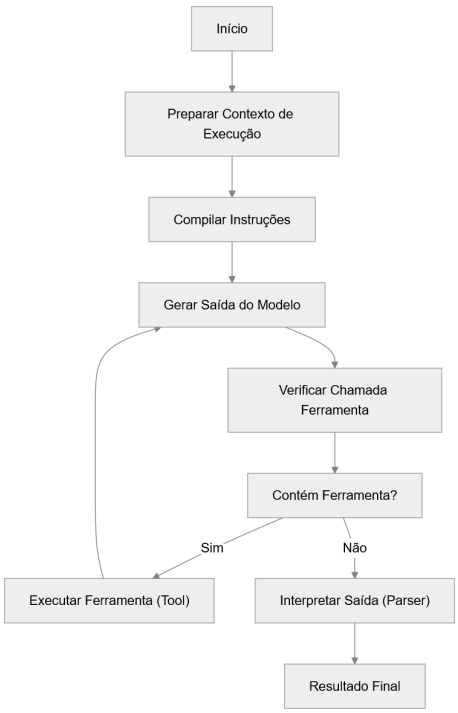


Fonte: **do autor**.

1. **Inicialização do Agente:** O processo começa com a criação de uma instância da classe `Agent`. Nesta etapa, o usuário fornece os componentes essenciais: um modelo de linguagem (`Model`), um parser (`Parser`) e, opcionalmente, um executor de ferramentas (`ToolExecutor`).

2. **Chamada do Método 'execute()':** O usuário inicia a execução do agente chamando o método `execute()` da instância da classe `Agent`. Como argumentos, este método recebe o prompt e parâmetros adicionais de execução.

Figura 4 – Fluxograma de Execução do Agente



Fonte: **do autor**.

3. **Formatação do Prompt (Template):** Se o prompt for um objeto `Prompt`, o módulo `Template` entra em ação, substituindo as variáveis e placeholders dentro do template pelos valores fornecidos pelo usuário.

4. **Geração de Texto (Model):** O prompt formatado é então enviado para o modelo de linguagem especificado através da interface `Model`.

5. **Execução de Ferramentas (Opcional):** Caso um executor de ferramentas tenha sido fornecido na inicialização e a saída do modelo de linguagem indique a necessidade de utilizar uma ferramenta, o `Agent` executa a ferramenta correspondente.

6. **Análise da Saída (Parser):** Após a geração de texto pelo modelo (e a eventual execução de ferramentas), o módulo `Parser` processa a saída, convertendo-a de texto bruto para um formato estruturado definido pelo usuário.

7. **Retorno (AgentOutput):** Finalmente, o agente retorna um objeto `AgentOutput`. Este objeto contém a saída bruta do modelo, os dados estruturados pelo parser e metadados da execução, como tempo de processamento, tokens utilizados e informações sobre as ferramentas executadas.

4.4 PUBLICAÇÃO E DISPONIBILIDADE

O código-fonte foi publicado no GitHub[8] sob a licença MIT (OPEN SOURCE INITIATIVE, 2020), permitindo o uso, modificação e distribuição livre. O repositório contém o código completo, histórico de commits e recursos para contribuir com o projeto.

[8] Disponível em: <https://github.com/ldaros/agent-builder>

A documentação completa, incluindo guias de uso, exemplos e referência da API, está disponível na página do projeto no GitHub. A documentação visa facilitar a adoção por desenvolvedores, fornecendo informações claras e concisas sobre suas funcionalidades e como utilizá-las.

Para simplificar a integração em projetos JavaScript/TypeScript, a versão compilada da biblioteca foi publicada no NPM (Node Package Manager)[9]. Desenvolvedores podem instalar o Agent Builder facilmente em seus projetos utilizando qualquer gerenciador de pacotes compatível com o NPM.

5 AVALIAÇÃO

5.1 ESTUDO DE CASO: CLASSIFICAÇÃO DE SENTIMENTOS EM AVALIAÇÕES DE FILMES

Para avaliar a eficácia e usabilidade do framework em um cenário real, foi desenvolvido um sistema de análise de sentimentos aplicado a avaliações de filmes. Este estudo de caso utiliza o *Movie Review Dataset* (MAAS et al., 2011), um conjunto de dados amplamente utilizado para tarefas de classificação de sentimentos.

O objetivo da aplicação é classificar avaliações de filmes como positivos ou negativos e gerar tags relevantes associadas ao conteúdo da review. A saída do sistema deve ser estruturada em formato JSON, conforme o schema definido. O desempenho foi avaliado em termos de acurácia na classificação e comparado com o framework Langchain construído usando linguagem de programação Python, utilizando o mesmo modelo de linguagem (GPT-4o Mini) para garantir uma comparação justa. A usabilidade de ambos os frameworks também foi considerada na análise.

O Langchain foi escolhido como ponto de comparação por ser um dos frameworks mais populares na área de desenvolvimento de aplicações baseadas em agentes, além de apresentar funcionalidades similares ao Agent Builder.

O dataset contém 50.000 reviews de filmes, divididas igualmente entre classes positivas e negativas. Para este estudo de caso, foi utilizado um subconjunto balanceado composto pelas primeiras 200 reviews do conjunto de teste (100 positivas e 100 negativas).

O código e prompts utilizados nos testes está disponível no repositório oficial do projeto no GitHub[10].

5.2 RESULTADOS E DISCUSSÃO

Para calcular a acurácia, as previsões geradas pelos sistemas foram comparadas com os rótulos reais presentes no dataset. A acurácia foi calculada usando a seguinte fórmula:

$$\text{Acurácia} = \frac{\text{Número de Previsões Corretas}}{\text{Total de Exemplos Avaliados}}$$

Ambas as implementações, utilizando o Agent Builder e Langchain, alcançaram uma acurácia de 98% na classificação de sentimentos das reviews. Essa similaridade no desempenho é esperada, visto que ambos utilizaram o mesmo modelo de linguagem (GPT-4 Mini) e um conjunto de dados relativamente pequeno.

[9] Disponível em: <https://www.npmjs.com/package/agent-builder>
[10] Disponível em: <https://github.com/ldaros/agent-builder-framework-comparison>

Apesar da performance similar em termos de acurácia, algumas diferenças importantes foram observadas entre os frameworks:

- **Robustez e Maturidade:** O Langchain, sendo um framework mais maduro e com maior tempo de desenvolvimento, apresenta um conjunto mais amplo de ferramentas e integrações, o que o torna mais robusto para uma variedade maior de casos de uso.
- **Suporte a Internacionalização:** O Agent Builder demonstrou maior facilidade de uso para lidar com textos em português, com suporte nativo a internacionalização.
- **Modularidade e Extensibilidade:** A arquitetura orientada a objetos do Agent Builder proporciona maior modularidade e extensibilidade em comparação com o Langchain. A adição de novos modelos, parsers e ferramentas é simplificada pela estrutura do framework.
- **Leveza e Dependências:** A utilização de recursos padrão da linguagem TypeScript e uma menor quantidade de dependências externas tornam o Agent Builder mais leve que o Langchain, o que pode ser vantajoso em ambientes com recursos limitados.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O Agent Builder demonstrou ser um framework promissor para o desenvolvimento de agentes de IA, com desempenho comparável ao Langchain em tarefas de classificação de sentimentos. Sua arquitetura modular, suporte a internacionalização e foco na simplicidade o tornam uma alternativa atraente, especialmente para projetos que exigem flexibilidade, extensibilidade e suporte a diferentes idiomas. Embora o Langchain ofereça maior robustez e um ecossistema mais amplo no momento, o Agent Builder apresenta um forte potencial para crescimento e desenvolvimento futuro. Para projetos em estágios iniciais ou com requisitos específicos de internacionalização e modularidade, o Agent Builder pode ser a escolha ideal.

Como trabalhos futuros, visa-se expandir as funcionalidades do Agent Builder através da integração com outros provedores de LLMs, aprimorando a flexibilidade e a abrangência do framework. Além disso, pretende-se refinar os prompts embutidos para otimizar o desempenho dos agentes, ampliar o suporte a idiomas adicionais para internacionalização completa, e adicionar suporte a formatos de dados estruturados mais complexos, aumentando a versatilidade e a aplicabilidade do framework em diferentes contextos. Finalmente, planeja-se aprimorar a documentação e os exemplos de uso, facilitando a adoção e o aprendizado por parte da comunidade.

Referências

AJV.ORG. *AJV: Another JSON Schema Validator*. 2024. Disponível em: <https://ajv.js.org/>. Acesso em 30 Novembro 2024. Citado na página 8.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1994. Citado na página 4.

I18NEXT. *i18next: Internationalization Framework for JavaScript*. 2024. Disponível em: <https://www.i18next.com/>. Acesso em 30 Novembro 2024. Citado na página 8.

JR, R. L. et al. A case study on test case construction with large language models: Unveiling practical insights and challenges. In: *Anais do XXVII Congresso Ibero-Americano em Engenharia de Software*. Porto Alegre, RS, Brasil: SBC, 2024. p. 388–395. Disponível em: <https://sol.sbc.org.br/index.php/cibse/article/view/28465>. Citado 2 vezes nas páginas 2 e 5.

LANGCHAIN. *Introduction to LangChain*. 2024. Disponível em: <https://python.langchain.com/v0.2/docs/introduction/>. Acesso em 30 Novembro 2024. Citado na página 5.

LI, Y.; RAMPRASAD, R.; ZHANG, C. A simple but effective approach to improve structured language model output for information extraction. In: AL-ONAIZAN, Y.; BANSAL, M.; CHEN, Y.-N. (Ed.). *Findings of the Association for Computational Linguistics: EMNLP 2024*. Miami, Florida, USA: Association for Computational Linguistics, 2024. p. 5133–5148. Disponível em: <https://aclanthology.org/2024.findings-emnlp.295>. Citado 2 vezes nas páginas 9 e 10.

LIU, Z. et al. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv*, 2023. Disponível em: <http://arxiv.org/abs/2310.02170v1>. Citado na página 4.

MAAS, A. L. et al. Learning word vectors for sentiment analysis. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, 2011. p. 142–150. Disponível em: <http://www.aclweb.org/anthology/P11-1015>. Citado na página 14.

MICROSOFT. *TypeScript: JavaScript with Syntax for Types*. 2024. Disponível em: <https://www.typescriptlang.org/>. Acesso em 30 Novembro 2024. Citado na página 8.

MOCHA.JS. *Mocha: The JavaScript Test Framework*. 2024. Disponível em: <https://mochajs.org/>. Acesso em 30 Novembro 2024. Citado na página 8.

MONARD, M. C.; BARANAUSKAS, J. A. Conceitos sobre aprendizado de máquina. In: *Sistemas Inteligentes para Engenharia*. São Carlos: Universidade de São Paulo, 2003. Disponível em: <https://dcm.ffclrp.usp.br/~augusto/publications/2003-sistemas-inteligentes-cap4.pdf>. Citado na página 3.

NADKARNI, P. M.; OHNO-MACHADO, L.; CHAPMAN, W. W. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, v. 18, n. 5, p. 544–551, 2011. Disponível em: <https://academic.oup.com/jamia/article/18/5/544/829676>. Citado na página 3.

NVIDIA. *Introduction to LLM Agents*. 2023. Disponível em: <https://developer.nvidia.com/blog/introduction-to-llm-agents/>. Acesso em 30 Novembro 2024. Citado na página 4.

OPEN SOURCE INITIATIVE. *The MIT License*. 2020. Disponível em: <https://opensource.org/licenses/MIT>. Acesso em 30 Novembro 2024. Citado na página 13.

OPENROUTER. *OpenRouter Runner*. 2024. Disponível em: <https://github.com/OpenRouterTeam/openrouter-runner>. Acesso em 30 Novembro 2024. Citado na página 5.

STACK OVERFLOW. *Stack Overflow Developer Survey 2023*. 2023. Disponível em: <https://survey.stackoverflow.co/2023/>. Acesso em 30 Novembro 2024. Citado na página 5.

TOPSAKAL, O. *Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast*. 2023. Disponível em: <https://www.researchgate.net/publication/372669736_Creating_Large_Language_Model_Applications_Utilizing_LangChain_A_Primer_on_Developing_LLM_Apps_Fast>. Citado na página 5.

VASWANI, A. et al. Attention is all you need. In: *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017)*. Long Beach, CA, USA: [s.n.], 2017. Disponível em: <https://arxiv.org/abs/1706.03762>. Citado 2 vezes nas páginas 3 e 4.

WEBER, I. Large language models as software components: A taxonomy for llm-integrated applications. *arXiv preprint arXiv:2406.10300*, 2024. Disponível em: <https://arxiv.org/abs/2406.10300>. Citado 2 vezes nas páginas 2 e 5.

WHITE, J. et al. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv*, 2023. Disponível em: <http://arxiv.org/abs/2302.11382v1>. Citado na página 4.