Relatório Agente 1746

Lucas Miguel

September 2025

1 Introdução

Este documento detalha as decisões de arquitetura e design tomadas durante a construção do "Agente 1746", uma aplicação de IA conversacional. O objetivo do projeto foi criar um assistente autônomo capaz de interagir em linguagem natural para consultar, analisar e responder a perguntas sobre o dataset de chamados públicos do 1746 da Prefeitura do Rio de Janeiro, hospedado no Google BigQuery.

Este relatório serve como uma documentação técnica do projeto, explicando não apenas o que foi construído, mas, fundamentalmente, por que cada componente foi projetado de sua maneira específica. O foco da análise está na construção do grafo de execução e na implementação do sistema de memória persistente com SQLite, que são os pilares da funcionalidade e robustez do agente. O sistema final é modular, com memória e orquestrado pelo LangGraph, com uma interface web interativa via Chainlit.

2 Arquitetura Geral

O sistema é um agente autônomo baseado em um grafo de estados. Cada nó no grafo representa uma capacidade específica, permitindo um fluxo de trabalho complexo e condicional. A figura abaixo apresenta o grafo construído para orquestrar o fluxo do agente.

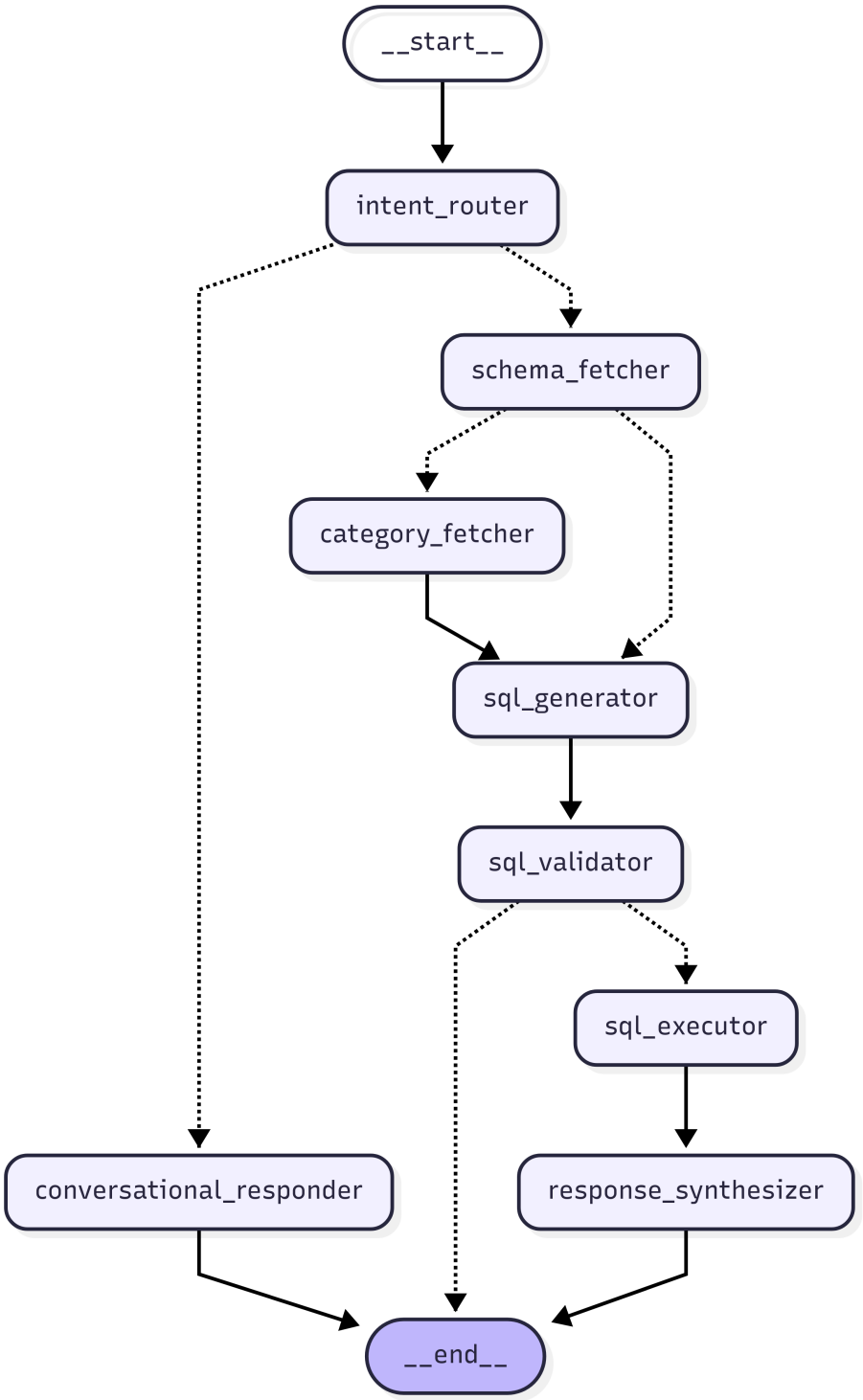

Figure 1: Diagrama da Arquitetura do Grafo de Execução do Agente.

3 Anatomia do Agente

Cada nó no grafo é uma unidade funcional com uma responsabilidade clara.

3.1 `intent_router` (Roteador de Intenção)

Responsabilidade: Classificar a intenção do usuário e direcionar o fluxo de trabalho.

Justificativa: Atua como o "porteiro" do agente. Sem ele, o sistema tentaria gerar SQL para entradas conversacionais (ex: "Olá"), resultando em falhas e desperdício de recursos. Ao separar as intenções (`chat`, `sql_direct`, `sql_contextual`), o roteador otimiza tanto o custo quanto a experiência do usuário.

Ao identificar saudações, agradecimentos ou perguntas genéricas, o roteador imediatamente desvia o fluxo para o caminho chat, para que o usuário seja respondido pelo nosso querido assistente flameguista. Isso é crucial não apenas para a otimização de recursos, evitando o acionamento desnecessário do caro pipeline de análise de dados, mas também para a experiência do usuário, garantindo que interações sociais sejam recebidas com uma resposta contextualmente apropriada e amigável, em vez de uma falha técnica ou uma mensagem de "não encontrei dados".

Uma vez que uma pergunta é classificada como uma solicitação de dados, o roteador realiza uma segunda camada de triagem, mais sutil, para determinar a complexidade da análise necessária. O caminho sql_direct funciona como uma "via expressa" para consultas inequívocas, tipicamente aquelas com filtros claros baseados em datas ou números. Para essas perguntas, o agente pode gerar um SQL preciso apenas com o conhecimento do esquema da tabela, e o roteador otimiza o processo ao pular etapas de investigação adicionais. Em contraste, o caminho sql_contextual é a rota investigativa, ativada para perguntas com filtros de texto ambíguos, onde o vocabulário do usuário pode não corresponder à terminologia exata do banco de dados. Este fluxo aciona o category_fetcher, um passo deliberado de descoberta de dados que equipa o LLM com o conhecimento necessário para resolver ambiguidades e aumentar a taxa de sucesso da consulta.

3.2 `schema_fetcher` (Buscador de Esquema)

Responsabilidade: Obter a estrutura das tabelas do BigQuery em tempo real.

Justificativa: Permite muito mais precisão no SQL gerado pelo agente. Sem este nó, o sistema seria forçado a operar com base em um esquema de banco de dados estático e pré-definido no prompt, uma abordagem frágil. O schema_fetcher age como os "olhos" do agente, conectando-se diretamente à fonte da verdade, o BigQuery, para obter o esquema exato das tabelas em tempo real. Este passo inicial de descoberta de dados elimina a classe mais comum de erros na geração de SQL: a alucinação de nomes de colunas.

Além da precisão imediata, o schema_fetcher introduz um desacoplamento crucial entre a lógica do agente e a estrutura física do banco de dados, tornando

o sistema resiliente à evolução natural dos dados.

3.3 `category_fetcher` (Buscador de Categorias)

Responsabilidade: Buscar os valores únicos de colunas de texto importantes.

Justificativa: Resolve o "problema do vocabulário" entre o usuário e o banco de dados. Ao fornecer ao LLM uma lista de valores válidos para tipo, subtipo e categoria, as principais colunas categóricas do banco, nós o tornamos mais resiliente a possíveis ambiguidades que apenas o esquema do banco não é suficiente para resolver, aumentando as chances de uma consulta funcional mesmo nessas condições.

O nó `schema_fetcher` fornecia ao LLM o **esquema** do banco de dados, informando-o que existiam colunas como `tipo` e `subtipo`, ambas do tipo `STRING`. No entanto, o esquema não dizia nada sobre o **conteúdo** dessas colunas.

Dessa forma, sempre existiam ambiguidades em caso de filtros textuais. Assim, para esses casos, o nó é executado e realiza um SELECT DISTINCT dessas colunas que causavam mais confusão, como um analista de dados certamente faria.

Essas informações são injetadas no contexto do sql generator, melhorando consideravelmente a query gerada pelo agente para esses casos de filtros textuais.

3.4 `sql_generator` (Gerador de SQL)

Responsabilidade: Converter a pergunta do usuário e o contexto acumulado em uma consulta SQL válida.

Justificativa: atua como o motor central que converte a intenção do usuário em uma consulta SQL executável. Arquiteturalmente, sua função mais importante é servir como a convergência para toda a inteligência coletada pelos nós anteriores.

É alimentado com um contexto que inclui o histórico da conversa, o esquema preciso da tabela obtido pelo schema_fetcher e as categorias de texto válidas do category_fetcher.

Busca mitigar a imprevisibilidade do LLM através de um sistema baseado em regras explícitas, em vez de um raciocínio de formato livre. O prompt define um conjunto de "REGRAS ESSENCIAIS" que funcionam como barreiras de proteção para o comportamento do LLM.

3.5 `sql_validator` (Validador de SQL)

Responsabilidade: Atuar como um firewall, inspecionando o SQL gerado antes da execução.

Justificativa: A função principal do nó sql_validator é servir como um ponto de controle de segurança obrigatório entre a geração da consulta e sua execução. O sql_generator utiliza um Modelo de Linguagem, cuja saída é inerentemente probabilística e não pode ser completamente garantida. Portanto,

o validador existe para mitigar os riscos de o LLM gerar uma consulta indesejada, seja por erro de interpretação ou em resposta a uma entrada maliciosa do usuário (prompt injection). Este nó intercepta o SQL gerado e o verifica contra um conjunto de regras de segurança, garantindo que apenas consultas seguras prossigam para o banco de dados.

A implementação deste nó utiliza lógica determinística, através de expressões regulares e comparação de listas, em vez de uma nova chamada a um LLM. Essa decisão foi tomada para garantir que o processo de validação seja previsível, rápido e imune às mesmas vulnerabilidades que visa prevenir. O nó executa duas verificações principais: primeiro, procura por palavras-chave que modificam dados, como UPDATE ou DELETE, para assegurar a natureza de somente leitura do agente. Segundo, extrai os nomes das tabelas da consulta e os valida contra uma lista de permissões explícita, impedindo o acesso a dados não autorizados.

3.6 `sql_executor` (Executor de SQL)

Responsabilidade: Isolar a lógica de interação com o BigQuery.

Justificativa: Este nó funciona como a interface exclusiva entre o ambiente de raciocínio do agente e a infraestrutura de dados externa, o Google BigQuery. Sua responsabilidade é única e bem definida: receber uma string de consulta SQL, executá-la no banco de dados e retornar os resultados de forma estruturada.

3.7 `response_synthesizer` (Sintetizador de Resposta)

Responsabilidade: Traduzir os dados brutos em uma resposta em linguagem natural.

Justificativa: Experiência do usuário. Apresentar dados brutos anularia o propósito de um agente conversacional. Este nó completa o ciclo, garantindo que a saída seja amigável.

Ele não apenas formata, mas também interpreta o resultado da consulta no contexto da pergunta original para formular uma frase coerente e útil, preenchendo a lacuna entre uma ferramenta de extração de dados e um verdadeiro assistente analítico.

Além disso, possui um prompt seguro contra prompt injection, fundamental para prevenção contra instruções mal intencionadas.

3.8 `conversational_responder` (Assistente Rubro-Negro)

Responsabilidade: Lidar com interações que não requerem análise de dados.

Justificativa: Garante que o agente possa responder a uma saudação de forma apropriada, tornando a interação mais fluida e clubista, garantindo uma otimização de recursos.

Sem este nó, o agente seria forçado a tratar um "Olá, tudo bem?" com a mesma seriedade de um pedido de relatório trimestral. Este nó, portanto, é a primeira linha de defesa contra o desperdício de recursos.

Além da eficiência, o conversational_responder encapsula toda a personalidade do agente. Enquanto os outros nós são analistas de dados sérios e focados, este é o componente que veste a camisa do mais querido, usa emojis específicos e se permite ser um pouco clubista.

4 O Sistema de Memória Conversacional com SQLite

A capacidade de manter uma conversa contextual é o que eleva o agente de uma ferramenta para um assistente.

A implementação de memória no agente utiliza o SqliteSaver do LangGraph para resolver o problema de interações sem estado. Por padrão, cada chamada a um LLM é independente, o que impede a continuidade em um diálogo. A memória persistente é necessária para que o agente entenda perguntas de acompanhamento e mantenha o contexto ao longo de uma conversa. O SqliteSaver foi escolhido por ser um checkpointer que salva o estado da conversa a cada turno, garantindo a continuidade da interação.

A opção mais simples, o InMemorySaver, foi considerada, mas descartada em favor do SqliteSaver por razões de robustez e persistência.

O InMemorySaver armazena o histórico da conversa diretamente na memória RAM do processo em que a aplicação está rodando. Embora seja extremamente rápido por não ter operações de disco, sua principal desvantagem é a volatilidade. Qualquer interrupção no processo, seja o script terminando, o servidor sendo reiniciado ou uma falha inesperada, resulta na perda completa e irrecuperável de todas as conversas. Isso o torna inadequado para qualquer caso de uso que exija continuidade de sessão. A memória existiria apenas durante uma única execução contínua do script, e o agente "esqueceria" as interações a cada reinicialização, o que anularia o propósito de ter um histórico persistente.

Em contraste, o SqliteSaver foi escolhido por fornecer persistência em disco. Ele armazena o estado de todas as conversas em um arquivo físico (agent_memory.sqlite), desacoplando a memória do ciclo de vida da aplicação. Esta é uma característica fundamentalmente mais robusta por dois motivos principais. Primeiro, ele garante a continuidade da sessão: um usuário pode fechar a interface e retornar dias depois e, ao fornecer o mesmo thread_id, sua conversa será perfeitamente restaurada. Segundo, ele torna o histórico do agente auditável e independente, pois o arquivo de banco de dados pode ser acessado, analisado e feito backup externamente, mesmo com a aplicação desligada. A escolha pelo SQLite foi, portanto, uma decisão deliberada para garantir que a memória do agente fosse uma característica confiável e permanente da arquitetura, e não um estado temporário e frágil.

Esta arquitetura permite que o agente mantenha inúmeras conversas parale-

las, garantindo que o histórico de uma sessão não interfira nas outras, o que é um requisito para aplicações multi-usuário ou multi-sessão.

5 Interface com Chainlit

A decisão de utilizar Chainlit como a interface para o agente foi motivada pela necessidade de prover um ponto de acesso interativo e amigável para o usuário, superando as limitações de um script de linha de comando. A escolha recaiu sobre este framework por ele ser especializado na criação de aplicações de IA conversacional.

Para aprimorar a usabilidade e o controle, foi introduzida a capacidade do usuário de gerenciar ativamente a memória da conversa através da troca do thread_id diretamente na interface. Implementada por meio de um comando de barra (/thread novo_id), essa funcionalidade transforma o sistema de memória de um mecanismo passivo para uma ferramenta interativa. Ela permite que um usuário mantenha múltiplas linhas de análise em paralelo, alternando entre diferentes contextos de conversa sem perder o histórico de nenhuma delas. O funcionamento é demonstrado na imagem abaixo.

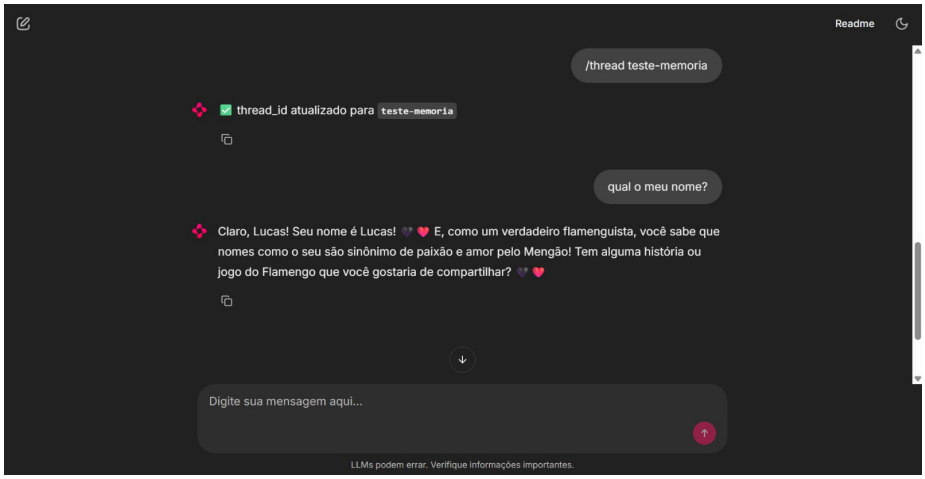

Figure 2: Interface Chainlit e Gerenciamento de Memória

Além disso, foi introduzido o pacote pt-BR.json nas configurações .chainlit traduzindo todos os elementos estáticos da interface, como botões e mensagens de status, alinhando a linguagem da interface com a do agente. Também foi implementado um chainlit.md para um guia de uso do agente customizado que pode ser acessado diretamente pelo usuário.

6 Conclusão

O desenvolvimento do Agente 1746 resultou em um sistema que vai além de um simples tradutor de linguagem natural para SQL. Através de uma arquitetura modular orquestrada pelo LangGraph, foi construído um agente dotado de capacidades de raciocínio contextual, memória persistente e camadas de segurança essenciais.

Igualmente importante, a priorização da experiência do usuário, através da interface interativa e nativa com Chainlit, da capacidade de manter conversas contextuais e de uma persona amigável, transforma o que poderia ser uma ferramenta complexa em um assistente acessível e agradável de usar.

O processo de desenvolvimento foi uma jornada particularmente gratificante, desde ajustes no prompt flamenguista até a resolução iterativa de problemas complexos. Ver o agente evoluir a cada etapa, tornando-se mais seguro, inteligente e divertido, foi o aspecto mais envolvente do trabalho.