Text-to-Insight: Um Sistema Multi-Agente para Consulta a Bancos de Dados Relacionais via Linguagem Natural*

Caio Petroncini[1], Cauã Sathler[2], João Pedro Calixto[1], Jonas Melo[3], Julio Cesar Fuganti[1], Luiz Correia dos Santos[1], Moacir Ponti[1]

[1] Instituto de Ciências Matemáticas e de Computação (ICMC) – USP – São Carlos, SP
[2]Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG
[3]Centro Universitário de Brasília (UniCEUB) – Brasília, DF

{caiopetroncini, jpcalixto, luizgcorreiadosantos, ponti}@usp.br
cauasathlerufmg@gmail.com, jonashonorato4@gmail.com
juliocesarfuganti@gmail.com

Resumo. *O TextToInsight é uma biblioteca Python de código aberto que converte perguntas em linguagem natural em consultas SQL seguras para bancos de dados relacionais. Utilizando uma arquitetura de grafos de agentes via LangGraph, o sistema orquestra módulos especializados: Planejador, Agente de Código, Executor e Crítico. O fluxo conta com um ciclo de autocorreção iterativa, onde o Crítico fornece feedback estruturado ao Planejador em caso de erros. Além da execução automatizada, a biblioteca oferece visualizações de dados e suporte a Human-in-the-Loop para a resolução de ambiguidades complexas.*

Palavras-chave: Text-to-SQL, Interfaces em Linguagem Natural, Agentes LLM, LangGraph, Autocorreção Iterativa.

1. Introdução

A linguagem SQL permanece como principal interface para exploração de bancos relacionais, mas seu domínio ainda representa uma barreira significativa para profissionais não-técnicos. Nos últimos anos, modelos de linguagem de grande porte (LLMs) impulsionaram avanços substanciais em Text-to-SQL, com métodos recentes atingindo altas acurácias de execução nos benchmarks Spider e BIRD. Entretanto, esses resultados foram obtidos majoritariamente em cenários controlados, com esquemas relativamente simples e pipelines baseados em uma única chamada ao modelo, sem mecanismos explícitos de verificação, crítica ou recuperação de erros [3].

Essa limitação torna-se evidente em ambientes mais próximos de aplicações reais. O benchmark Spider 2.0 Lite introduz bancos de larga escala com schemas extensos e consultas analíticas complexas, incluindo funções de janela, múltiplos JOINs e raciocínio multi-etapa. Nesse cenário, o desempenho dos métodos atuais degrada acentuadamente, com os antigos melhores sistemas no SPIDER 1.0 reportando apenas 5,7% de acurácia de execução no SPIDER 2.0 [2]. Esses resultados sugerem que melhorar apenas a qualidade

*Este projeto acadêmico de graduação é vinculado à Rede de Avanço em Inteligência Artificial (RAIA), grupo de pesquisa e extensão do ICMC-USP.

da primeira geração de SQL pode não ser suficiente para lidar com consultas complexas e ambiguidades semânticas em ambientes reais.

Neste trabalho, descrevemos o **Text-to-Insight**, uma biblioteca Python de código aberto[1] desenvolvida como uma primeira tentativa de construir um sistema end-to-end de consulta a bancos relacionais via linguagem natural. O sistema organiza a tarefa em etapas de planejamento, introspecção de schema, geração de SQL, execução segura e geração de visualizações gráficas, orquestradas via LangGraph. Como estratégia inicial de verificação, incorporamos um agente Crítico que avalia o resultado após a execução e aciona novas tentativas quando detecta problemas, uma abordagem intuitiva que avaliamos empiricamente neste trabalho.

Este artigo relata essa primeira iteração: o que foi construído, como foi avaliado e, principalmente, o que os experimentos revelaram sobre os limites da abordagem. O código está disponível publicamente e o projeto está em desenvolvimento ativo.

2. Trabalhos Relacionados

A pesquisa em Text-to-SQL evoluiu rapidamente com a adoção de LLMs e benchmarks mais desafiadores. O Spider [1] consolidou-se como principal referência da área ao introduzir generalização entre bancos distintos e schemas não vistos durante o treinamento. Nesse cenário, abordagens recentes passaram a explorar estratégias de prompting contextualizado para geração direta de SQL. Métodos como DAIL-SQL [3] e C3 [4] mostram que enriquecimento de contexto, seleção de exemplos e engenharia de prompt podem produzir consultas de alta qualidade em uma única interação com o modelo.

Uma segunda linha de pesquisa busca decompor o problema em etapas intermediárias especializadas. DIN-SQL [5] segmenta explicitamente o raciocínio em múltiplas fases de decomposição e refinamento, enquanto MAC-SQL [6] utiliza múltiplos agentes cooperativos para geração e verificação de consultas. Mais recentemente, CHESS [9] e Spider-Agent [2] avançaram o uso de arquiteturas multi-agente e *code agents* para lidar com schemas complexos e bancos de larga escala. Em paralelo, trabalhos como CodeS [10] exploram treinamento especializado de modelos open-source para aumentar robustez e desempenho em benchmarks Text-to-SQL.

Apesar desses avanços, a maioria dos sistemas ainda opera em fluxo linear: a consulta é gerada uma única vez e o processo é encerrado. O Spider 2.0 [2] evidencia a limitação desse paradigma ao mostrar forte degradação de desempenho em cenários analíticos mais realistas. O Text-to-Insight explora esse mesmo espaço de problema, investigando se um laço de autocorreção guiado por um agente Crítico independente é capaz de recuperar erros de geração sem depender de um gabarito externo. O sistema incorpora ainda execução segura em modo somente-leitura e persistência de estado via LangGraph [8], e sua avaliação empírica busca entender os limites e o potencial dessa abordagem.

3. Arquitetura do Sistema

O Text-to-Insight é implementado como uma biblioteca Python com duas interfaces: uma API programática (`InsightEngine`) e uma CLI. O núcleo é um grafo de agentes compilado pelo LangGraph, composto por nós especializados conectados por arestas fixas e condicionais (Figura 1).

[1]Código-fonte disponível em: `https://github.com/gruporaia/TextToInsight`

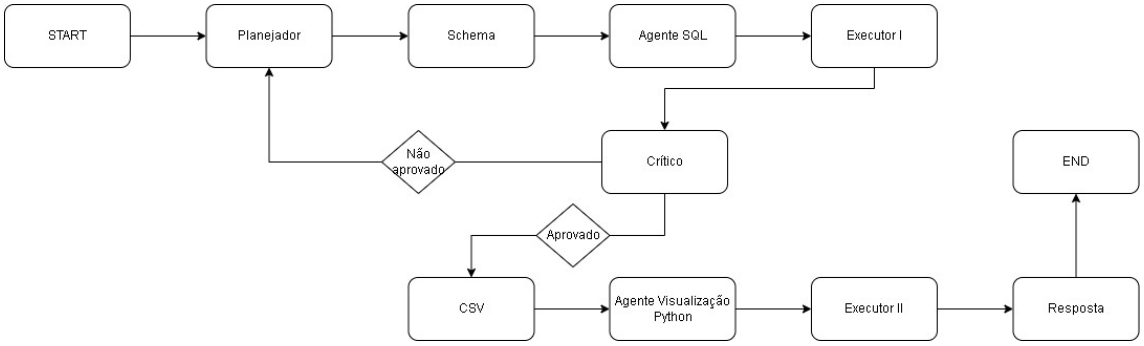

Figura 1. Fluxo simplificado do grafo Text-to-Insight

3.1. Descrição dos Componentes

O fluxo de execução baseia-se na transição de estado entre os nós do grafo. Cada componente detalhado a seguir desempenha um papel específico na progressão da pergunta até o artefato final:

Planejador. Analisa a pergunta, o schema e o feedback de iterações anteriores para decidir a próxima ação: obter schema, gerar SQL, revisar estratégia ou acionar o Human-in-the-Loop (interrompendo o pipeline para solicitar esclarecimentos diretamente ao usuário via terminal).

Introspecção de Schema. A CLI gerencia a integração local estabelecendo conexão com o arquivo SQLite via driver nativo. Em modo somente-leitura, extrai metadados reais via PRAGMA: tabelas, colunas com tipos e chaves estrangeiras.

Agente SQL. Gera a consulta via LLM. O prompt é estruturado concatenando a pergunta, as instruções do sistema (restrição a `SELECT` e `WITH`), o DDL das tabelas relevantes e o histórico de erros anteriores.

Executor I. Valida a SQL por expressão regular, bloqueando comandos destrutivos (`INSERT`, `DELETE`, `DROP`, etc.) e múltiplas *statements*. Executa em modo *read-only*.

Crítico. Avalia via LLM se o resultado responde à pergunta original, produzindo veredito e feedback estruturado. Erros de execução causam reprovação automática.

Autocorreção. Caso a query SQL seja reprovada, o fluxo retorna ao Planejador com feedback. O Agente recebe o histórico completo de tentativas e motivos de reprovação. Ciclo limitado a 3 tentativas.

Salvar CSV. O resultado da consulta é exportado para um arquivo CSV com *timestamp*, garantindo rastreabilidade.

Agente Visualização Python. Uma aresta condicional avalia via LLM se os dados justificam uma visualização; quando acionado, gera código `matplotlib` a partir da pergunta, SQL e amostra dos dados.

Executor II. O código é executado em subprocesso isolado (limite de 30s) e o gráfico salvo como PNG em `graphs/`; falhas não interrompem o pipeline.

Resposta. Ao fim da execução, o nó de Resposta agrega a SQL gerada, os resultados e os feedbacks do Crítico em uma resposta em linguagem natural.

3.2. Estado e Telemetria

O estado compartilhado do pipeline é modelado como um *TypedDict* em Python, com campos obrigatórios (*pergunta* e *db_path*) e campos operacionais populados durante o fluxo: schema extraído, query gerada, histórico de feedback do Crítico, contador de tentativas e resultados de execução. O sistema rastreia automaticamente o consumo de tokens e métricas de execução ao longo do ciclo de vida da consulta.

4. Avaliação Experimental

4.1. Configuração

O sistema foi avaliado em dois benchmarks. No Spider 1.0 [1], utilizamos uma amostra de 500 perguntas (seed=42) com o modelo `gpt-4o-mini`, HITL desativado e limite de 3 tentativas. No Spider 2.0 Lite [2], realizamos dois testes com 50 perguntas cada: um com `gpt-4o-mini` e outro com `gpt-5-mini`, para isolar o efeito do modelo em schemas mais complexos.

Para cada pergunta, o pipeline: (1) obtém os resultados gold do benchmark; (2) invoca o Text-to-Insight; (3) executa a query gerada; e (4) compara os conjuntos de resultados. Três métricas foram utilizadas: **Exact Match** (comparação column-level, ignorando nomes de colunas), **F1 Score** (precision/recall sobre colunas de resultado) e **SQL Similarity** (similaridade textual entre SQLs normalizadas, aplicável apenas quando o gold fornece SQL).

4.2. Resultados

A Tabela 1 apresenta os resultados nos dois benchmarks.

Tabela 1. Resultados da avaliação nos dois benchmarks.

Métrica	Spider 1.0 4o-mini (N=500)	Spider 2.0 4o-mini (N=50)	Spider 2.0 5-mini (N=50)
Taxa de aprovação (Crítico)	91,6%	48,0%	86,0%
Exact Match Rate	79,2%	20,0%	50,0%
F1 Score médio	0,787	0,209	0,602
SQL Similarity médio	0,784	—	—

No Spider 1.0, 82,8% das perguntas foram resolvidas na primeira tentativa; as demais 86 acionaram o laço de autocorreção, com 44 convergindo em tentativas subsequentes e 42 falhando nas três. Dos 104 casos sem exact match, as categorias mais frequentes são diferenças em colunas do SELECT (89 casos), DISTINCT (23) e subconsultas (20). A maioria envolve divergências estruturais, mas alguns casos refletem erros semânticos, como escopo incompleto de contagem, uso incorreto de agregação ou caminhos de JOIN errados.

A Tabela 2 compara o sistema com e sem Crítico nos três cenários de teste. No Spider 1.0 com `gpt-4o-mini`, o efeito é neutro: 78,8% de EM sem Crítico contra 79,2% com Crítico. No Spider 2.0 com `gpt-4o-mini`, o laço produz ganho de 10 pontos percentuais (de 10% para 20%), pois a taxa de acerto na primeira tentativa é baixa

e o Crítico tem mais oportunidades de recuperar erros. Com `gpt-5-mini` no Spider 2.0, onde o modelo acerta 52% já na primeira tentativa, o efeito volta a ser neutro. Das 86 queries que o Crítico reprovou em alguma tentativa, incluindo 43 em que a primeira query já era correta (falsos negativos do Crítico) , 16 melhoraram (EM=False→True), 29 permaneceram corretas (EM=True→True), 27 permaneceram incorretas (EM=False→False) e 14 pioraram (EM=True→False), com saldo de $+2$.

A matriz de confusão (Tabela 3) ajuda a entender por que o efeito do Crítico é menor do que o esperado. Das 396 queries que tiveram exact match correto, o Crítico aprovou 377 (sensibilidade de 95,2%); das 104 queries incorretas, reprovou apenas 23 (especificidade de 22,1%). O Crítico tende a aprovar a maioria do que vê, o que faz com que a maior parte das queries erradas nunca chegue ao laço. E mesmo nas 86 que entraram em correção, o saldo é pequeno: 16 melhoraram e 14 pioraram após a reescrita, sugerindo que, sem um sinal externo de corretude, a reescrita não tem direção preferencial.

Como Gerador e Crítico usam o mesmo modelo, queries que pareceram plausíveis na geração tendem a parecer plausíveis também na avaliação. O feedback do Crítico muitas vezes aponta problemas específicos de lógica ou particularidades dos dados que não aparecem no esquema; ainda assim, seu veredito depende de uma avaliação subjetiva da qualidade da query, problema difícil que MAC-SQL [6] e CHESS [9] tentam contornar gerando múltiplos candidatos e comparando os resultados de execução para buscar consenso. Soma-se a isso o fato de que o laço, na forma atual, não decompõe a tarefa em sub-etapas: embora o Agente receba o histórico das tentativas anteriores e o feedback do Crítico, ele regenera a query inteira a cada iteração, sem trabalhar incrementalmente sobre partes do problema. Tentativa e erro guiada por feedback continua útil, mas não oferece a mesma estrutura de raciocínio de técnicas que quebram o problema explicitamente em passos, como chain-of-thought ou DIN-SQL [5].

Tabela 2. Ablação do Crítico: comparação entre benchmarks e modelos.

Cenário	EM sem Crítico	EM com Crítico	Saldo
Spider 1.0, `gpt-4o-mini` (N=500)	78,8%	79,2%	$+2$
Spider 2.0, `gpt-4o-mini` (N=50)	10,0%	20,0%	$+5$
Spider 2.0, `gpt-5-mini` (N=50)	52,0%	50,0%	-1

Tabela 3. Matriz de confusão do Crítico no Spider 1.0 (N=500, acurácia 80%).

	EM Correto	EM Incorreto
Crítico Aprovou	377 (VP)	81 (FP)
Crítico Reprovou	19 (FN)	23 (VN)

No Spider 2.0 Lite, a diferença entre modelos é mais pronunciada do que o efeito do Crítico: `gpt-4o-mini` atinge 20% de EM contra 50% do `gpt-5-mini`, uma diferença de 30 pontos percentuais atribuível à qualidade do modelo. A análise qualitativa revelou falhas em funções de janela, CTEs complexas e schema linking em schemas extensos, indicando que o gargalo principal é a capacidade de raciocínio multi-etapa do modelo base, não o mecanismo de correção.

5. Exemplo de Uso

A seguir, apresentamos uma execução completa do Text-to-Insight sobre o dataset Olist [7], ilustrando o pipeline para uma pergunta analítica que envolve agregações sobre quatro tabelas distintas.

Pergunta: *"Quais são as 10 categorias de produtos com maior receita total em pedidos entregues, e qual é a nota média de avaliação dos clientes para cada uma?"*

O sistema obteve o schema, gerou a seguinte SQL na primeira tentativa e obteve aprovação do Crítico:

```
WITH RevenueAndRatings AS (
   SELECT
      p.product_category_name,
      SUM(oi.price)   AS total_revenue,
      AVG(r.review_score) AS average_rating
   FROM orders o
   JOIN order_items oi ON o.order_id = oi.order_id
   JOIN products   p  ON oi.product_id = p.product_id
   JOIN order_reviews r ON o.order_id = r.order_id
   WHERE o.order_status = 'delivered'
   GROUP BY p.product_category_name
)
SELECT product_category_name, total_revenue, average_rating
FROM RevenueAndRatings
ORDER BY total_revenue DESC
LIMIT 10;
```

O Agente de Visualização Python avaliou os dados retornados e gerou automaticamente um gráfico de barras com receita total e nota média por categoria.

Resposta: *"As 10 categorias de produtos com maior receita total em pedidos entregues foram identificadas. A categoria Beleza e Saúde lidera, com receita total de aproximadamente R$ 1.221.790,24 e nota média de avaliação de 4,19. Outras categorias como Relógios e Presentes e Cama, Mesa e Banho também apresentaram receitas significativas e notas acima de 4."*

6. Conclusão

Os resultados deste trabalho reforçam uma limitação importante dos sistemas atuais de Text-to-SQL: gerar uma consulta sintaticamente plausível é significativamente mais fácil do que verificar sua correção semântica em cenários reais. Embora benchmarks clássicos como Spider indiquem desempenho próximo da saturação, os experimentos no Spider 2.0 Lite mostram que schemas extensos, planejamento multi-etapa e ambiguidades analíticas ainda permanecem desafios em aberto.

O laço de autocorreção investigado neste trabalho apresentou comportamento ambíguo. Em cenários com alta taxa de erro inicial, o mecanismo consegue recuperar parte das falhas e melhorar o exact match final. Entretanto, quando gerador e avaliador compartilham o mesmo modelo e contexto, o Crítico tende a herdar os mesmos pontos cegos do sistema que tenta corrigir. Isso sugere que autocorreção puramente baseada em julgamento semântico do LLM possui limitações estruturais, especialmente na ausência de sinais externos de verificação.

Apesar dessas limitações, acreditamos que arquiteturas iterativas e reflexivas continuam sendo uma direção promissora para Text-to-SQL em ambientes reais. Os resultados apontam que avanços futuros provavelmente dependerão menos de melhorar marginalmente a primeira geração de SQL e mais da capacidade do sistema de validar, decompor e revisar seu próprio raciocínio. Nesse sentido, o Text-to-Insight fornece uma base aberta e modular para exploração de mecanismos de autocorreção, consenso entre agentes e validação semântica em consultas analíticas complexas. Trabalhos futuros incluem a exploração de candidatos múltiplos com comparação cruzada de resultados; técnicas de decomposição de queries para lidar com funções de janela e CTEs complexas; e RAG baseado em grafos para schemas extensos.

7. Uso de IA Generativa

Os autores utilizaram ferramentas de IA generativa (Claude e ChatGPT) para revisão de linguagem, apoio a estruturação de trechos do texto e auxílio no desenvolvimento do código para o projeto.

Referências

[1] Yu, T. et al. (2018). Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. *Proceedings of EMNLP 2018*, 3911–3921.

[2] Lei, F. et al. (2024). Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. *arXiv preprint arXiv:2411.07763*.

[3] Gao, D. et al. (2024). DAIL-SQL: Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proceedings of the VLDB Endowment*, 17(5), 1132–1145.

[4] Dong, X. et al. (2023). C3: Zero-shot Text-to-SQL with ChatGPT. *arXiv preprint arXiv:2307.07306*.

[5] Pourreza, M. e Rafiei, D. (2024). DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. *Advances in Neural Information Processing Systems*, 36.

[6] Wang, B. et al. (2025). MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. *Proceedings of the 31st International Conference on Computational Linguistics (COLING 2025)*, 540–557.

[7] Olist (2018). Brazilian E-Commerce Public Dataset by Olist. Disponível em: `https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce`.

[8] LangGraph Documentation (2024). LangGraph: Build Stateful Multi-Actor Applications. Disponível em: `https://langchain-ai.github.io/langgraph/`.

[9] Talaei, S., Pourreza, M., Chang, Y.-C., Mirhoseini, A., and Saberi, A. (2024). CHESS: Contextual Harnessing for Efficient SQL Synthesis. *arXiv preprint arXiv:2405.16755*.

[10] Li, H., Zhang, J., Liu, H., Fan, J., Zhang, X., Zhu, J., Wei, R., Pan, H., Li, C. e Chen, H. (2024). CodeS: Towards Building Open-Source Language Models for Text-to-SQL. *Proceedings of the ACM on Management of Data*, 2(3), 1–28.