Perspectivas de Desenvolvimento de Assistentes Conversacionais com LLMs: Um Estudo de Caso

Annia L. Sebold[1], **Samuel Ferraz**[1]

[1]Universidade Federal do Mato Grosso do Sul (UFMS)
79.070-900 – Mato Grosso do Sul – MS – Brasil

{annia_sebold}{samuel.ferraz}@ufms.br

Resumo. *Assistentes conversacionais baseados em Large Language Models (LLMs) tornaram-se cada vez mais comuns, mas projetá-los de forma confiável e manutenível não é simples, pois envolve decisões arquiteturais em múltiplas dimensões. O objetivo deste trabalho é propor uma avaliação sistemática dos passos e desafios enfrentados no desenvolvimento iterativo de um assistente conversacional para controle financeiro pessoal integrado ao Telegram, construído ao longo de três versões progressivas: um sistema monolítico, uma arquitetura baseada em múltiplos fluxos de trabalho com extração de entidades por LLM e uma arquitetura multi-agente com ferramentas atômicas. As versões foram comparadas por critérios qualitativos da engenharia de software e por métricas quantitativas coletadas com o módulo N8N Evaluations, sob o padrão LLM-as-Judge. Ao propormos uma discussão sobre os ganhos e perdas de cada versão, este trabalho constitui um relato prático do desenvolvimento de um assistente conversacional baseado em LLMs, do protótipo à versão final, constituindo uma referência para projetos semelhantes.*

Abstract. *Conversational assistants based on Large Language Models (LLMs) have become increasingly common, but designing them in a reliable and maintainable way is not straightforward, as it involves architectural decisions across multiple dimensions. This work reports the iterative development of Finbot, a conversational assistant for personal finance management integrated with Telegram, built through three progressive versions: a monolithic system, a multi-workflow architecture with LLM-based entity extraction, and a multi-agent architecture with atomic tools. The versions were compared using qualitative software engineering criteria and quantitative metrics collected with the N8N Evaluations module under the LLM-as-Judge paradigm. Finally, the work discusses the gains and trade-offs of each version and constitutes a practical report on the development of an LLM-based conversational assistant, from prototype to final version, which may serve as a reference for similar projects.*

1. Introdução

Nos últimos anos, os *Large Language Models* (LLMs) deixaram de ser apenas objetos de pesquisa acadêmica para se tornarem componentes centrais de produtos amplamente utilizados. LLMs são modelos de IA generativa, uma classe de sistemas capazes de gerar conteúdo novo como texto, código ou imagens a partir de padrões aprendidos durante o treinamento. Um dos exemplos mais visíveis dessa adoção são os assistentes conversacionais, sistemas de *software* que interagem com usuários por meio de linguagem natural

para realizar tarefas ou responder perguntas. O ChatGPT, um assistente conversacional baseado em LLM lançado em novembro de 2022, atingiu 100 milhões de usuários em dois meses, o ritmo de adoção mais rápido já registrado entre aplicações de software [Hu 2023]. O fenômeno não foi isolado: 78% das organizações relataram uso de IA em 2024, comparado a 55% no ano anterior, e o investimento privado global em IA generativa chegou a US$ 33,9 bilhões no mesmo período [Stanford HAI 2025]. Hoje, assistentes conversacionais baseados em LLMs estão presentes em aplicações que vão do atendimento ao cliente à automação de tarefas pessoais.

Apesar da rápida adoção, projetar assistentes conversacionais baseados em LLMs de forma confiável e manutenível não é simples, pois envolve decisões arquiteturais em múltiplas dimensões: estratégias de raciocínio e planejamento, padrões de orquestração (agente único *versus* multi-agente, coordenação centralizada *versus* descentralizada) e formas de interação com ferramentas externas via *tool calling* [Kapoor et al. 2026]. Entre uma demonstração funcional e um sistema robusto em produção há uma distância considerável: em tarefas com várias etapas encadeadas, falhas em etapas intermediárias se acumulam ao longo da execução, e o não-determinismo inerente aos LLMs dificulta a avaliação e a depuração sem protocolos padronizados. Relatos práticos que documentem a evolução arquitetural real de sistemas com LLMs, com métricas concretas e o registro das decisões de cada etapa, têm valor particular para a comunidade de desenvolvimento de software.

Para explorar esses desafios na prática, este trabalho relata o desenvolvimento do *Finbot*, um assistente conversacional para controle financeiro pessoal integrado ao Telegram. O domínio financeiro foi escolhido por oferecer um problema real e bem delimitado: o cartão de crédito figura como principal causa de endividamento das famílias brasileiras [CNC 2026], e mais da metade dos consumidores do país não controla os gastos mensais com esse instrumento [CNDL et al. 2023], em parte pela complexidade das ferramentas tradicionais de controle. O domínio financeiro é, portanto, o campo de aplicação, enquanto o foco do trabalho é a construção e a evolução arquitetural do sistema.

Este trabalho investiga os *trade-offs* arquiteturais no desenvolvimento iterativo de um assistente conversacional baseado em LLMs, considerando três abordagens progressivas: um sistema monolítico, uma arquitetura baseada em múltiplos fluxos de trabalho com extração de entidades por LLM, e uma arquitetura multi-agente com ferramentas atômicas. Analisa-se também o impacto de decisões de infraestrutura e de escolha de provedor de modelo. O objetivo é documentar e comparar as três versões arquiteturais do *Finbot* por meio de métricas objetivas e de lições qualitativas extraídas de cada etapa, identificando os ganhos e perdas em cada evolução. As ferramentas e os provedores utilizados são apresentados e justificados ao longo do texto, e o resultado esperado é um relato prático do desenvolvimento de um assistente conversacional com LLM, do protótipo à versão final, que sirva de referência para projetos semelhantes.

2. Fundamentação Teórica

Este capítulo apresenta a base conceitual necessária para entender as escolhas feitas no desenvolvimento do Finbot.

2.1. LLMs e IA Generativa

A IA generativa é uma classe de modelos de aprendizado profundo capazes de criar conteúdo original como texto, imagens, áudio, vídeo ou código, em resposta a uma solicitação do usuário. Esses modelos aprendem padrões a partir de grandes volumes de dados durante o treinamento e usam esse conhecimento para gerar novos conteúdos [IBM 2024a]. Dentro dessa classe, os LLMs (*Large Language Models*) são especializados em linguagem natural, treinados para prever e gerar sequências de texto.

A base arquitetural dos LLMs é o *Transformer* [Vaswani et al. 2017], que substituiu as redes neurais recorrentes ao processar sequências de forma paralela por meio de mecanismos de *self-attention*. Como ilustra a Figura 1, o *Transformer* é composto por um codificador (*encoder*) e um decodificador (*decoder*), cada um formado por camadas empilhadas de atenção multi-cabeça (*multi-head attention*) e redes neurais densas (*feed-forward*). O mecanismo de atenção mapeia cada *token* da sequência de entrada a um conjunto de consultas, chaves e valores, e calcula para cada posição um peso que indica o quanto ela deve "prestar atenção" às demais. Isso permite que o modelo capture relações contextuais de longo alcance de forma simultânea, independentemente da distância entre os *tokens* na sequência, o que viabilizou treinamento em escala muito maior do que as arquiteturas recorrentes permitiam.

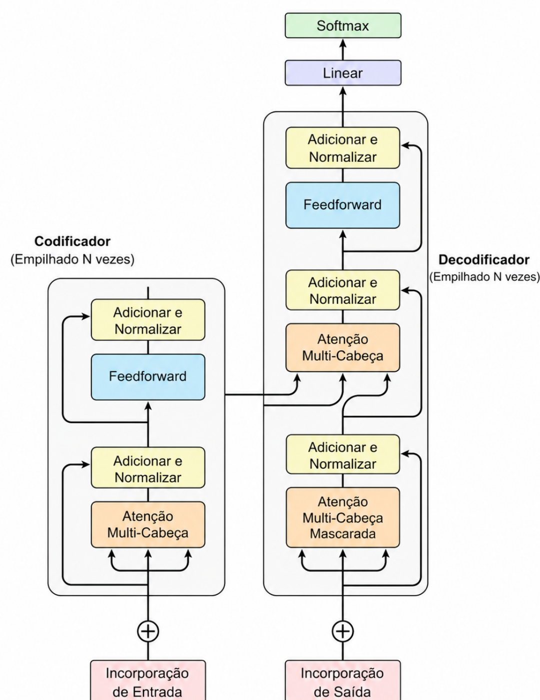

Figura 1. Arquitetura do *Transformer* com codificador (esquerda) e decodificador (direita). Fonte: [Vaswani et al. 2017].

Modelos como GPT [OpenAI 2023], Claude [Anthropic 2024] e Gemini [Google and DeepMind 2023] derivam dessa arquitetura e são treinados em volumes massivos de dados textuais. A interação com esses modelos se dá por meio de *prompts*,

instruções em linguagem natural fornecidas como entrada que orientam o comportamento e o conteúdo da resposta gerada pelo modelo. Durante o treinamento, o modelo ajusta seus parâmetros, que são os pesos internos que determinam como cada entrada é transformada em saída ao longo das camadas da rede. O número de parâmetros é uma das principais dimensões de escala de um LLM. Trabalhos demonstraram empiricamente que o desempenho desses modelos melhora de forma previsível, segundo uma lei de potência, à medida que se aumentam o número de parâmetros, o tamanho do conjunto de dados e o poder computacional empregado no treinamento [Kaplan et al. 2020]. Isso ajuda a explicar por que os maiores modelos atuais, com parâmetros na ordem de bilhões, exibem maior capacidade de seguir instruções, raciocinar sobre problemas e gerar respostas coerentes, ainda que demandem mais computação.

No entanto, LLMs não são inerentemente confiáveis: sem mecanismos externos de verificação, podem produzir afirmações plausíveis mas incorretas, fenômeno conhecido como alucinação [Kapoor et al. 2026], o que motiva o uso de ferramentas externas e bases de dados como âncoras de confiabilidade, que desloca o papel do LLM de gerador de texto para orquestrador de ações verificáveis.

2.2. *AI Agents* e *Tool Calling*

Um agente de IA baseado em LLM é um sistema que acopla um modelo de linguagem a um laço de execução capaz de observar um ambiente, planejar ações, invocar ferramentas e atualizar a memória com base nos resultados obtidos [Kapoor et al. 2026]. Nesse paradigma, o modelo de linguagem funciona como o núcleo de raciocínio do agente, enquanto o laço de execução é o que efetivamente orquestra a interação com o ambiente. Em conjunto, o agente deixa de apenas gerar texto e passa a traduzir intenções do usuário em procedimentos executáveis sobre sistemas externos, como bancos de dados, APIs e interfaces de mensagens: o LLM decide qual ação tomar, e o laço a executa e devolve o resultado ao modelo.

O framework ReAct [Yao et al. 2023] estabeleceu as bases desse paradigma ao combinar explicitamente raciocínio e ação em um ciclo intercalado. A cada etapa, o agente primeiro articula um pensamento interno sobre o que fazer e por quê, depois executa uma ação sobre uma ferramenta externa e então obser-va o resultado antes de prosseguir. Esse ciclo de *Thought* $\rightarrow$ *Action* $\rightarrow$ *Observation*, ilustrado na Figura 2, torna as etapas intermediárias inspecionáveis e permite que o agente corrija o plano diante de uma falha ou de um resultado inesperado.

O mecanismo que viabiliza a interação com ferramentas é o *tool calling* (ou *function calling*), em que o modelo propõe chamadas a ferramentas com parâmetros estruturados, que são validadas e executadas externamente antes que o resultado retorne ao contexto do agente [Kapoor et al. 2026]. Quando múltiplos agentes especializados colaboram sob a coordenação de um orquestrador central, temos o padrão multi-agente hierárquico, no qual o orquestrador recebe a intenção do usuário, decompõe a tarefa e a delega ao agente mais adequado, que dispõe de suas próprias ferramentas atômicas para executar ações específicas.

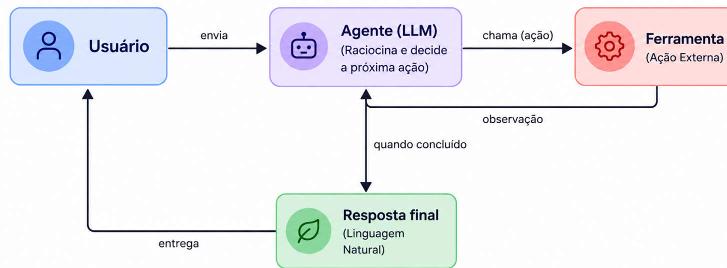

Figura 2. Ciclo de funcionamento de um *AI Agent* com *tool calling*: o agente raciocina sobre a mensagem do usuário, decide chamar uma ferramenta externa, observa o resultado e repete o ciclo até produzir a resposta final. Fonte: própria da autora.

2.3. Avaliação de Agentes com *LLM-as-Judge*

Avaliar a qualidade das respostas de sistemas baseados em LLMs é um desafio reconhecido na literatura [Zheng et al. 2023], já que métricas tradicionais de correspondência exata, como *BLEU* e *ROUGE*, originalmente desenvolvidas para tarefas de tradução automática e sumarização, não capturam aspectos semânticos importantes em saídas abertas em linguagem natural. Uma abordagem amplamente adotada é o padrão *LLM-as-Judge*, em que um segundo modelo de linguagem funciona como avaliador automático das respostas produzidas pelo sistema sob análise.

Esse padrão foi proposto e sistematizado por [Zheng et al. 2023], no qual foi demonstrado que LLMs usados como juízes alcançam concordância superior a 80% com avaliadores humanos especialistas em tarefas abertas. Também foram identificadas limitações do método, como viés de posição, de verbosidade e de auto-aprimoramento, propondo estratégias para mitigá-las. Em assistentes conversacionais, o padrão é aplicado para avaliar se a resposta gerada é correta, relevante e bem formatada, sem anotação humana caso a caso, o que torna a avaliação escalável durante o desenvolvimento. Na prática, conforme ilustrado na Figura 3, é definido um conjunto de entradas de teste com respostas esperadas e configurado um modelo juiz que pontua cada resposta do agente segundo critérios como correção, relevância e formato, permitindo comparar versões distintas do sistema de forma objetiva e reproduzível.

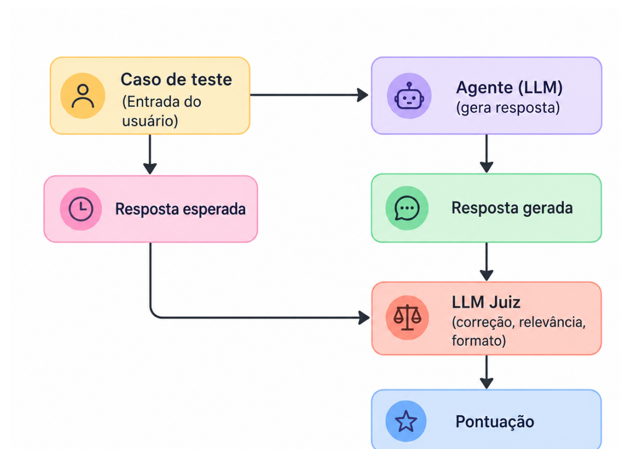

Figura 3. Fluxo de avaliação com o padrão *LLM-as-Judge*: a resposta gerada pelo agente é comparada à resposta esperada por um modelo juiz, que emite uma pontuação segundo critérios de correção, relevância e formato. Fonte: própria da autora.

2.4. Ferramentas para Orquestração de Sistemas com LLMs

Construir um sistema baseado em LLMs exige orquestrar o fluxo de informação entre o modelo, as ferramentas externas e a lógica de controle da aplicação. Existem diferentes abordagens para isso, dispostas num espectro que vai do desenvolvimento inteiramente em código à composição majoritariamente visual. Em uma ponta estão as implementações que usam diretamente as APIs dos provedores de modelos, sem abstrações intermediárias: oferecem controle total, mas exigem que o desenvolvedor implemente manualmente grande parte das coisas, do gerenciamento de *prompts* à coordenação de ferramentas. Mais adiante no espectro estão os *frameworks* de código, como o LangChain [IBM 2024b] e sua extensão LangGraph [LangChain 2024], que fornecem abstrações de alto nível para modelos, memória, ferramentas e fluxos de execução, reduzindo o esforço de implementação em relação ao uso direto das APIs, mas mantendo a orquestração expressa em código [ZenML 2024]. Na ponta oposta estão as plataformas *low-code* e *no-code*, que privilegiam velocidade de desenvolvimento e acessibilidade por meio de interfaces visuais e nós pré-configurados, o que não significa ausência total de código, já que muitas delas permitem inserir lógica customizada em pontos específicos. A escolha entre essas abordagens é um compromisso entre flexibilidade, curva de aprendizado e velocidade, e foi um dos pontos avaliados neste trabalho. As subseções a seguir detalham as duas ferramentas consideradas, o N8N [n8n-io 2024] e o LangChain, situadas em pontos distintos desse espectro.

Plataformas *low-code* e *no-code* são caracterizadas pelo uso de interfaces visuais e componentes pré-construídos para criar aplicações com o mínimo de código manual. O crescimento desse paradigma reflete a demanda por mais velocidade de desenvolvimento: em 2021, o Gartner projetou que, até 2025, 70% das novas aplicações empresariais usariam tecnologias *low-code* ou *no-code* [Gartner 2021]. No caso de sistemas com LLMs, essas plataformas ganham relevância adicional por permitir que equipes de perfil técnico variado construam e iterem rapidamente sobre fluxos de orquestração de agentes sem implementar toda a infraestrutura do zero.

O N8N é uma plataforma de automação de fluxos de trabalho de código aberto que combina uma interface visual de construção com a possibilidade de escrever código JavaScript ou Python em qualquer etapa do fluxo [n8n-io 2024]. Sua unidade fundamental é o *workflow*, um grafo de nós (*nodes*) conectados que representam ações, transformações ou integrações com serviços externos. Fluxos podem ser acionados por eventos externos via *webhooks*, por agendamento ou por outros fluxos, compondo hierarquias de sub-fluxos reutilizáveis. O N8N oferece suporte nativo a agentes de IA, com nós dedicados a modelos de linguagem, memória conversacional e ferramentas chamáveis por agentes, o que o torna adequado ao desenvolvimento de sistemas conversacionais baseados em LLMs. A Figura 4 ilustra um *workflow* construído no N8N e seus elementos centrais: nós retangulares conectados por arestas que representam o fluxo de dados, cada um executando uma ação específica como o disparo do fluxo por mensagem recebida, a interação com um modelo de linguagem, a consulta a um banco de dados ou o envio da resposta ao usuário. Diferentemente do LangGraph mostrado no Código 1, no N8N a estrutura do fluxo é definida visualmente e o comportamento de cada nó é configurado por formulários, embora ainda seja possível inserir código personalizado por meio de nós de código quando necessário.

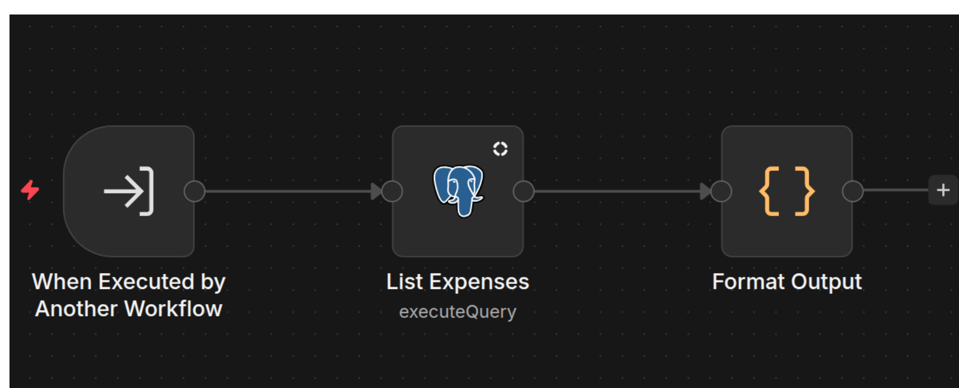

Figura 4. Construção de um fluxo no N8N por meio de sua interface visual baseada em nós. Fonte: própria da autora.

O LangChain, por sua vez, é um *framework* de código aberto desenvolvido em Python e JavaScript, lançado em outubro de 2022, que rapidamente se tornou uma das principais referências para construção de aplicações com LLMs [IBM 2024b]. Sua abordagem é modular: fornece abstrações padronizadas para modelos, memória, ferramentas, recuperação de documentos e encadeamento de operações, permitindo conectar qualquer LLM a fontes de dados externas e fluxos de trabalho complexos [Topsakal and Akinci 2024].

O Código 1 ilustra essa abordagem com um trecho extraído da prova de conceito do Finbot construída em LangGraph, contendo dois nós responsáveis pelo cadastro de um cartão de crédito. Cada nó é declarado como uma função Python que recebe o estado compartilhado do fluxo (`FinBotState`) e retorna uma atualização desse estado, possivelmente acompanhada de uma instrução de roteamento. O primeiro nó, `add_card_node`, verifica se o usuário já possui cartões cadastrados e, em caso negativo, retorna um objeto `Command` que indica ao grafo qual será o próximo passo da conversa por meio do campo `step`, além de adicionar ao histórico a mensagem a ser enviada ao usuário. O segundo nó,

add_card_nickname_node, processa a última mensagem recebida do usuário, atualiza o estado com o nome do banco emissor e solicita o apelido do cartão. Essa estrutura exemplifica a natureza centrada em código do LangChain: a topologia do fluxo, o tratamento do estado, a montagem do histórico de mensagens e o roteamento entre etapas são definidos de forma programática, em contraste com a configuração visual de nós e arestas adotada pelo N8N.

Essa diferença define o principal *trade-off* entre as duas ferramentas. O LangChain oferece mais flexibilidade e controle fino sobre cada parte do sistema, mas tem curva de aprendizado mais alta e exige manter mais código. O N8N permite prototipar mais rápido, com sua interface visual e seus nós pré-configurados, embora ofereça menos granularidade de controle. Para equipes que precisam iterar rápido sobre arquiteturas de agentes sem perder a possibilidade de inserir lógica customizada quando preciso, o N8N é uma alternativa viável ao desenvolvimento tradicional com frameworks de código.

O N8N está disponível em duas modalidades de implantação. Na *self-hosted*, o sistema é instalado e operado em infraestrutura própria, como um servidor virtual privado (VPS), com controle total sobre dados, configurações e custos operacionais, mas com maior responsabilidade de manutenção e configuração. Na *cloud*, a plataforma é gerenciada pela própria N8N, o que elimina a administração de infraestrutura em troca de um custo mensal fixo e de restrições sobre o número de execuções e fluxos ativos [n8n-io 2024].

Por fim, o N8N oferece um módulo nativo de avaliação de fluxos de IA, o *N8N Evaluations*, que permite definir conjuntos de casos de teste com entradas e saídas esperadas e executá-los de forma automatizada para medir o desempenho dos agentes ao longo do desenvolvimento. O módulo integra-se diretamente ao padrão *LLM-as-Judge* descrito na seção anterior, deixando que um modelo separado avalie as respostas geradas e produza métricas objetivas para comparação entre versões.

```
1  def add_card_node(state: FinBotState):
2      existing_cards = state.get("cards", [])
3
4      if not existing_cards:
5          bot_response = """
6          Percebi que voce ainda nao tem nenhum cartao registrado.
7          Vamos cadastrar seu cartao de credito!
8          Qual o nome do banco emissor do cartao?
9          (ex: Nubank, Itau, Santander...)
10         """
11         return Command(
12             goto="__end__",
13             update={
14                 "step": "add_card_nickname_node",
15                 "messages": state["messages"] + [SystemMessage(
                       content=bot_response)],
16             },
17         )
18
19 def add_card_nickname_node(state: FinBotState):
```

```
20      messages = state["messages"]
21      last_message = messages[-1].content
22      state["issuing_bank"] = last_message.strip()
23
24      card_name = state.get("card_name")
25      if not card_name:
26          bot_response = """
27          Otimo! Agora, como voce gostaria de chamar esse cartao?
28          Pode ser um apelido para facilitar, como
29          "Cartao Pessoal", "Nubank Black", etc.
30          """
31          return {
32              "messages": state["messages"] + [SystemMessage(
                    content=bot_response)],
33              "original_intent": None,
34          }
```

Código 1. Trecho de definição de nós no LangGraph, extraído da prova de conceito do Finbot com LangGraph.

2.5. Mensageria, Persistência e Gerenciamento de Estado

Esta seção discorre sobre as tecnologias e APIs que deram suporte ao desenvolvimento das três versões do Finbot.

Para a interação com o usuário, adotou-se o Telegram como plataforma de mensagens. O Telegram oferece uma API pública para criação de bots, a *Telegram Bot API* [Telegram 2024b], que permite a qualquer usuário criar agentes automatizados integrados ao aplicativo. A API suporta dois modelos de recebimento de atualizações: o *polling*, em que o bot consulta periodicamente os servidores do Telegram por novas mensagens, e o *webhook*, em que o Telegram envia uma requisição HTTP POST ao servidor do bot a cada nova mensagem [Telegram 2024a]. O modelo de *webhook* é preferido em produção por ser orientado a eventos: elimina consultas desnecessárias e reduz a latência de resposta. Cada mensagem recebida é serializada em JSON e entregue ao servidor configurado, responsável por processá-la e devolver uma resposta ao usuário pela API.

Sistemas conversacionais precisam de uma camada de persistência durável, e a primeira decisão não é qual banco usar, mas qual modelo de dados adotar. Bancos orientados a documentos, como o MongoDB, armazenam registros em formato semiestruturado (tipicamente JSON) e oferecem esquema flexível, o que se ajusta bem a dados heterogêneos ou que evoluem com frequência, inclusive aos próprios *payloads* JSON entregues por *webhooks*. Bancos relacionais, como o PostgreSQL, organizam os dados em tabelas com esquema fixo e relações explícitas, oferecendo garantias transacionais e integridade referencial. No caso do Finbot, o domínio é fortemente estruturado e relacional: cartões pertencem a usuários, faturas pertencem a cartões e despesas pertencem a faturas, e a consistência entre essas entidades, como o recálculo do total de uma fatura a cada despesa, é um requisito central. Por isso optou-se pelo modelo relacional: embora o dado de entrada chegue como *JSON* via *webhook*, ele é convertido para uma representação estruturada assim que entra no sistema, e as garantias de integridade do banco relacional pesaram mais do que a flexibilidade de esquema que um banco de documentos ofereceria. Sendo assim,

adotou-se o PostgreSQL, um sistema de gerenciamento de banco de dados relacional de código aberto reconhecido pela robustez, pela conformidade com o padrão SQL e pelo suporte a tipos de dados avançados. Sobre o PostgreSQL, plataformas como o Supabase oferecem uma camada de *backend* que inclui banco de dados gerenciado, autenticação, APIs geradas automaticamente e assinaturas em tempo real, posicionando-se como alternativa de código aberto a plataformas proprietárias como o Firebase [Supabase 2024], reduzindo o esforço de configuração e ainda oferecendo uma camada gratuita adequada a desenvolvimento e prototipagem.

Sistemas conversacionais com múltiplos turnos precisam manter contexto ao longo da conversa. Quando o usuário envia algo como "antes de cadastrar, corrige o valor para R$ 50", o sistema só responde corretamente se souber o que foi dito antes. Isso é necessário porque os LLMs são, em sua maioria, sem estado: cada chamada à API é independente e não retém informações das interações anteriores. É preciso, então, armazenar explicitamente o histórico de mensagens e o estado da sessão entre uma interação e a próxima [Redis 2024b]. Aqui há um *trade-off* entre tempo de resposta e durabilidade dos dados, e a escolha depende de *quais* dados estão em jogo. O estado conversacional é, por natureza, efêmero: representa apenas em que ponto de um fluxo o usuário se encontra naquele momento, tem tempo de expiração curto e pode ser perdido sem prejuízo real, no pior caso, o usuário reinicia uma interação em andamento, sem que nenhum dado financeiro persistido seja afetado. Para esse tipo de dado, a baixa latência importa mais que a durabilidade, e justifica-se o uso de um armazenamento em memória. Adotou-se o Redis, um armazenamento de dados em memória de código aberto que opera com latência inferior a um milissegundo em leituras e escritas [Redis 2024a], cumprindo dois papéis: manter a memória conversacional (o histórico recente de mensagens) e persistir o estado da sessão entre turnos. Os dados que de fato exigem confiabilidade (cartões, faturas e despesas) permanecem no banco relacional, de modo que a volatilidade do Redis nunca coloca em risco informação crítica do usuário.

3. Metodologia

Este trabalho adota uma abordagem de estudo de caso com desenvolvimento iterativo, no qual o Finbot foi construído em três versões arquiteturais progressivas. O objetivo não é apenas documentar o produto final, mas registrar as decisões tomadas em cada iteração, os problemas identificados e as motivações que levaram à evolução da arquitetura.

A escolha do domínio financeiro tem motivação concreta. Em 2024, o Brasil ultrapassou a marca de 220 milhões de cartões de crédito ativos, superando o próprio número de habitantes, e cerca de 53 milhões de pessoas estavam endividadas nessa modalidade [IstoÉ Dinheiro 2026]. Tarefas como registrar despesas, consultar faturas e editar lançamentos costumam exigir o uso de aplicativos dedicados ou planilhas de controle, com interfaces pouco intuitivas e registro manual. Um assistente conversacional integrado a uma plataforma de mensagens já utilizada no dia a dia, como o Telegram, simplifica essas tarefas ao permitir que sejam realizadas por meio de linguagem natural, sem que o usuário precise abrir um aplicativo específico ou aprender uma nova interface. Ou seja, é um domínio socialmente relevante e que também serve bem ao propósito técnico do trabalho: reúne tarefas que envolvem extração de informação estruturada, manutenção de estado entre turnos e execução de ações sobre dados persistidos.

O Finbot é um assistente conversacional integrado ao Telegram cujo objetivo é permitir que o usuário organize seus cartões de crédito e faturas em linguagem natural. As funcionalidades implementadas nas três versões são organizadas em três grupos:

- **Gestão de cartões:** cadastro de cartão de crédito com nome, limite, data de fechamento e data de vencimento.
- **Gestão de despesas:** registro de despesas únicas ou parceladas com as informações relevantes à transação; edição e remoção de despesas cadastradas.
- **Gestão de faturas:** geração de relatório de fatura por cartão e aviso automático e programado para o vencimento.

O banco de dados relacional, implementado no Supabase com PostgreSQL, conta com regras de integridade e lógica de negócio definidas diretamente por funções no banco, como validações e cálculos automáticos associados às despesas parceladas.

A avaliação do sistema ao longo das versões se dá por dois conjuntos de critérios. Os qualitativos olham aspectos arquiteturais e de experiência de desenvolvimento: **manutenibilidade**, que avalia a facilidade de modificar, depurar e estender o sistema; **acoplamento**, que mede o grau de dependência entre os componentes do fluxo; **delegação à LLM**, que indica o quanto a lógica de decisão e de extração é confiada ao modelo em vez de controlada por código; e **adequação funcional**, que verifica se o assistente cumpre seu objetivo, isto é, se executa corretamente as ações solicitadas de forma natural, sem respostas engessadas ou fluxos que ignorem a intenção expressa na mensagem.

Os critérios quantitativos são medidos pelo módulo N8N *Evaluations*, configurado nos principais fluxos do sistema. Para cada caso de teste, o módulo registra métricas de execução: total de *tokens* consumidos, *tokens* de entrada, *tokens* de saída e tempo de execução do fluxo e aciona um LLM juiz para avaliar a qualidade da resposta em relação à esperada. Os critérios usados pelo juiz são detalhados na subseção de avaliação. Essas métricas permitem comparar as versões de forma objetiva, sobretudo quanto a custo operacional, eficiência e qualidade das respostas de cada abordagem.

3.1. Modelagem do Banco de Dados

A estrutura central do banco de dados do Finbot tem seis tabelas, ilustradas na Figura 5. A tabela `users` armazena os usuários do sistema, identificados pelo `chat_id` do Telegram. A tabela `cards` representa os cartões de crédito cadastrados por cada usuário, com nome, limite, dia de fechamento e dia de vencimento. A tabela `invoices` representa as faturas de cada cartão, organizadas por ciclo mensal no formato `YYYY-MM`, com datas de fechamento e vencimento definidas na criação e status que pode ser `open`, `closed`, `paid`, `late` ou `partially_paid`. A tabela `expenses` registra as despesas associadas a um cartão e a uma fatura, suportando compras únicas e parceladas por meio dos campos `installment_number`, `installment_total` e `purchase_id`, que agrupa as parcelas de uma mesma compra. As tabelas `conversation_state` e `conversation_state_invoices` armazenam o estado conversacional da sessão, registrando a etapa atual do fluxo, o campo esperado e um rascunho em JSON com os dados coletados até o momento, mecanismo usado especialmente na V1 para persistir contexto entre turnos. Todas as tabelas possuem um `id` gerado automaticamente por UUID, um `created_at` preenchido na criação e um `updated_at` atualizado a cada modificação.

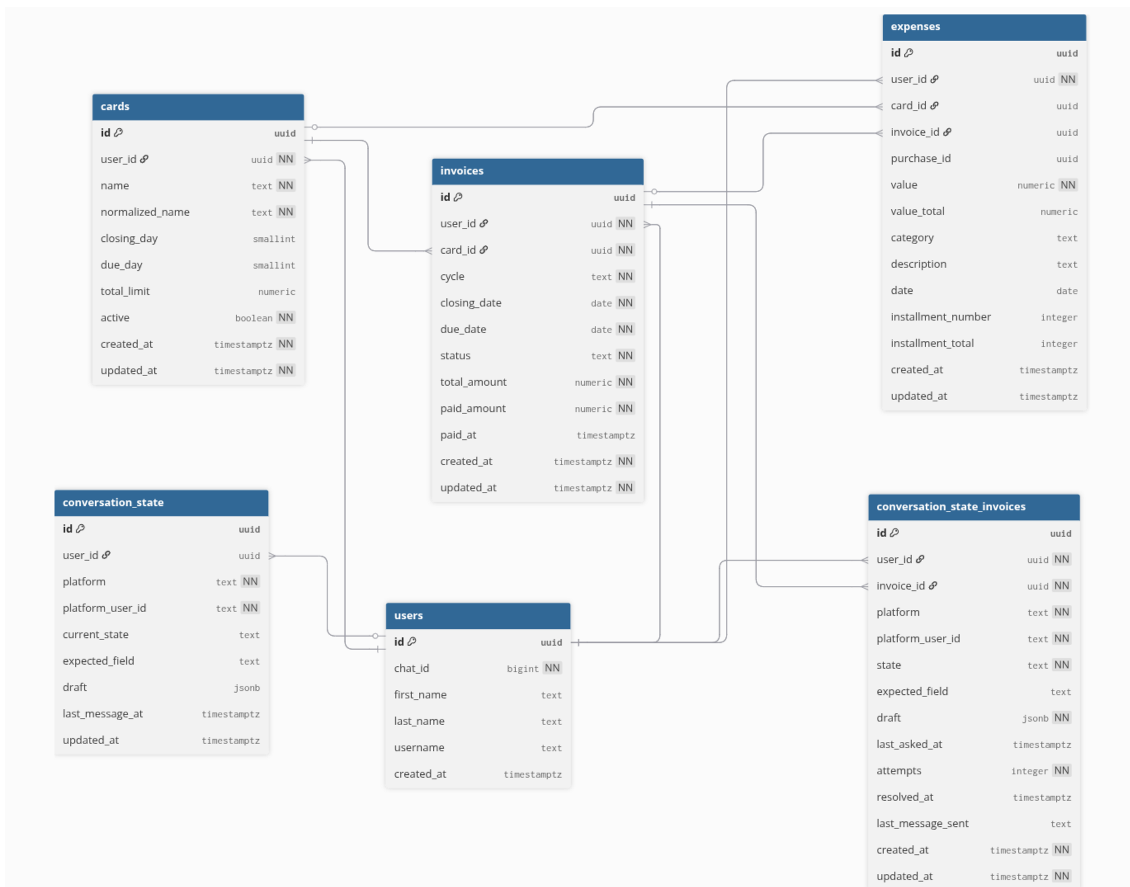

Figura 5. Diagrama entidade-relacionamento do banco de dados do Finbot. Fonte: própria da autora.

A lógica de negócio mais relevante está encapsulada em funções e gatilhos no próprio banco. A função `get_or_create_invoice` determina a qual fatura uma despesa pertence, com base na data da compra e no dia de fechamento do cartão: se a compra ocorreu após o fechamento do ciclo corrente, ela é alocada na fatura do mês seguinte, caso contrário, é alocada no ciclo atual. A fatura correspondente é criada se ainda não existir. Dessa forma, a responsabilidade de descobrir o ciclo da fatura fica no banco, e não no agente. O recálculo do total da fatura é gerenciado por um gatilho (*trigger*) na tabela `expenses`, conforme ilustrado na Figura 6: a função `trg_expenses_recalc_invoice` é acionada a cada inserção, atualização ou remoção de despesa e chama `recalc_invoice_total` para recalcular o campo `total_amount` da fatura afetada com a soma das despesas vinculadas. O total da fatura permanece consistente sem que nenhuma camada da aplicação precise gerenciar esse cálculo.

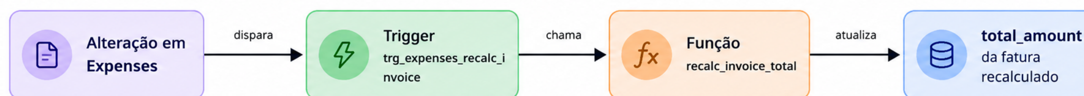

Figura 6. Fluxo automático de recálculo do total da fatura no banco de dados: a cada inserção, atualização ou remoção de uma despesa, o gatilho `trg_expenses_recalc_invoice` dispara a função `recalc_invoice_total`, que soma as despesas vinculadas e atualiza o campo `total_amount` da fatura. Fonte: própria da autora.

3.2. Prova de Conceito — LangGraph ou N8N

Antes de iniciar o desenvolvimento do Finbot, foi realizada uma prova de conceito para avaliar qual ferramenta de orquestração seria mais adequada ao projeto. A primeira abordagem testada usou o LangGraph[1], extensão do LangChain voltada à construção de agentes com fluxo de execução baseado em grafos de estados. Nessa prova de conceito, foi implementado um protótipo em Python com integração ao Groq como provedor de LLM, organizado em módulos separados para o grafo de execução, os nós de processamento e os estados da conversa.

Embora o LangGraph ofereça controle minucioso sobre o fluxo, ficou evidente que chegar a algo minimamente funcional exigia implementar manualmente uma série de componentes que, em uma plataforma como o N8N, já vêm prontos: integração com serviços externos como Telegram e bancos de dados, gerenciamento de memória conversacional, definição explícita de estados, configuração de nós e roteamento condicional. Numa abordagem baseada em código, o desenvolvedor constrói e mantém cada uma dessas peças, o que traz mais flexibilidade, mas concentra esforço em camadas de integração e estruturação que não são o objeto deste trabalho. A escolha implica abrir mão de parte do controle granular dos *frameworks* de código, mas para os objetivos definidos, a troca se mostrou favorável.

A partir dessa constatação, foi testado o N8N como alternativa. Os primeiros testes foram feitos localmente, com fluxos simples de integração com o Telegram. Depois, o ambiente foi migrado para um servidor com exposição pública, o que permitiu testar o sistema em condições mais próximas de produção. Na prática, o N8N levava a resultados funcionais com muito menos configuração inicial: a integração com o Telegram, o acionamento por *webhook* e a conexão com um LLM foram feitos visualmente em poucos nós, sem código de infraestrutura. Com base nessa comparação prática, o N8N foi escolhido como plataforma para as três versões do Finbot.

3.3. Infraestrutura — *Self-hosted* ou *Cloud*

Como descrito na Seção 3, o N8N pode rodar em duas modalidades: *self-hosted*, em infraestrutura própria, ou *cloud*, em ambiente gerenciado pela N8N. A escolha entre elas afeta diretamente custo, esforço de configuração e controle sobre os dados, fatores especialmente relevantes em um projeto acadêmico com orçamento restrito.

[1] Repositório disponível em: `https://github.com/fintrackbot/finbot-langgraph`.

A modalidade *cloud* oferece a vantagem da simplicidade: o ambiente vem configurado e mantido, sem administração de servidor. O problema é o custo: é um serviço pago por assinatura, com planos baseados no número de execuções e nos recursos disponíveis, um custo recorrente difícil de justificar para um protótipo acadêmico. A opção *self-hosted* elimina esse custo de assinatura ao rodar em infraestrutura própria e ainda oferece controle total sobre dados e configurações, porém exigindo a configuração e a manutenção do servidor.

Por isso, optou-se pela modalidade *self-hosted* ao longo de todo o desenvolvimento. No início, o N8N rodou na própria máquina de desenvolvimento, suficiente para os primeiros testes de integração. Depois, o sistema foi hospedado em um servidor, com acesso externo provido por um túnel Cloudflare, o que permitiu que o *webhook* do Telegram alcançasse a aplicação sem endereço IP público fixo nem configurações complexas de rede. Essa configuração deu controle completo sobre o ambiente, sem as restrições de execução dos planos *cloud*.

3.4. Provedores de LLM

Um provedor de LLM é um serviço que oferece acesso a modelos de linguagem de grande escala por meio de uma API, abstraindo a infraestrutura de execução do modelo (servidores, GPUs, balanceamento de carga) e permitindo que aplicações consumam o modelo como uma camada externa. A escolha do provedor influencia diretamente a variedade de modelos disponíveis, o custo por interação e a facilidade de experimentar com diferentes opções ao longo do desenvolvimento. Ao longo do desenvolvimento do Finbot, dois provedores foram utilizados em momentos distintos: o Groq, na fase inicial e o OpenRouter, adotado a partir da V2.

O Groq foi o provedor utilizado na prova de conceito e nos primeiros experimentos. Sua principal característica é a alta velocidade de inferência, obtida por uma arquitetura de hardware dedicada (*Language Processing Unit*, ou LPU), além de uma camada gratuita adequada à prototipagem [Groq 2024]. Nessa fase, o modelo utilizado foi o `llama-3.3-70b-versatile`, da família Llama [Meta AI 2024] da Meta, que oferecia bom equilíbrio entre velocidade e qualidade dentro do catálogo disponível. Contudo, este catálogo se restringe a modelos de código aberto, como os das famílias Llama e DeepSeek [DeepSeek 2024], o que limita a possibilidade de testar e comparar diferentes famílias de modelos durante o desenvolvimento. Conforme o sistema avançou, surgiu o interesse de experimentar com uma variedade maior de modelos, o que ia além do que o catálogo do Groq oferecia.

Isso motivou a migração para o OpenRouter, um *gateway* que unifica o acesso a modelos de diferentes provedores, incluindo modelos proprietários, como os das famílias GPT, Claude e Gemini, por uma única interface compatível com o padrão da API da OpenAI [OpenRouter 2024]. A mudança trouxe duas vantagens centrais. Uma foi o acesso a uma gama bem maior de modelos, permitindo escolher o mais adequado a cada tarefa e comparar o comportamento do sistema sob modelos diferentes sem mexer na integração. A outra foi a flexibilidade de roteamento, que viabiliza trocar de modelo com mínima alteração na configuração dos fluxos. Em compensação, usar modelos proprietários pelo OpenRouter envolve custo por *tokens* consumidos, o que tornou o monitoramento de gastos um fator relevante nas versões mais avançadas, como se discute na análise de resultados.

3.5. Configuração das Avaliações no N8N Evaluations

A coleta das métricas quantitativas foi feita pelo módulo *N8N Evaluations*, que permite configurar conjuntos de casos de teste e executá-los de forma automatizada sobre os *workflows* do sistema. Cada caso é definido em uma planilha com a entrada do usuário, o estado inicial esperado da sessão, o conteúdo esperado da resposta, o estado final esperado e, quando necessário, o histórico de mensagens anteriores. Ao executar a avaliação, o módulo dispara o *workflow* para cada linha da planilha, captura a resposta gerada e aciona um modelo configurado como juiz para verificar se ela atende aos critérios definidos. A Figura 7 mostra como esse mecanismo é representado dentro de um *workflow* no N8N: o nó *Eval Trigger* dispara o fluxo durante a execução dos casos de teste, enquanto um nó condicional decide quais etapas devem ser ignoradas no modo de avaliação, como a persistência de informações ou o envio da mensagem ao usuário final, que não são necessárias para medir o comportamento do agente.

O comportamento do modelo juiz é determinado por um *prompt* de avaliação que estrutura a decisão em critérios objetivos. Em vez de pedir um julgamento livre, o *prompt* apresenta ao juiz três blocos de informação: a entrada que o agente recebeu, a saída que ele produziu e os critérios esperados do caso de teste, e exige uma resposta em *JSON* com sinalizadores booleanos separados para o estado e para o conteúdo da resposta. Essa separação evita que uma resposta correta no conteúdo, mas com estado final errado (ou vice-versa), seja considerada um acerto. O Código 2 reproduz o *prompt* utilizado. Note-se que o critério de conteúdo é avaliado semanticamente: o juiz é instruído a julgar a intenção e a completude da resposta, e não a correspondência exata de palavras, o que é justamente a vantagem do padrão *LLM-as-Judge* sobre métricas como BLEU e ROUGE.

```
1  Voce e um avaliador de qualidade de agentes de IA. Analise a
      resposta do agente e decida se ela cumpre o comportamento
      esperado.
2
3  === ENTRADA QUE O AGENTE RECEBEU ===
4  message: {{ message }}
5  state_string: {{ state_string }}
6  history: {{ history }}
7
8  === SAIDA DO AGENTE ===
9  response: {{ response }}
10 state_string retornado: {{ state_string }}
11
12 === CRITERIOS DO CASO DE TESTE ===
13 expected_contains: {{ expected_contains }}
14 expected_state_prefix: {{ expected_state_prefix }}
15 description: {{ description }}
16
17 === SUA TAREFA ===
18 1. state_ok (booleano): o state_string retornado COMECA com
      expected_state_prefix?
19 2. response_ok (booleano): a response cobre semanticamente o que
       expected_contains descreve? Nao exija palavras exatas,
      avalie intencao e completude.
```

```
20 3. passed (booleano): true somente se AMBOS state_ok e
      response_ok forem true.
21 4. reasoning (string curta): explique em 1-2 frases por que
      passou ou falhou.
22
23 Responda SOMENTE com JSON valido:
24 { "passed": bool, "state_ok": bool, "response_ok": bool, "
      reasoning": "..." }
```

Código 2. Prompt utilizado pelo modelo juiz na avaliação automatizada das respostas dos agentes, sob o padrão *LLM-as-Judge*.

A Tabela 1 apresenta um exemplo de caso de teste utilizado para avaliar o agente de despesas (*Expense Agent*).

Tabela 1. Exemplo de caso de teste do agente de despesas. Fonte: própria da autora.

Campo	Valor
Descrição	Todos os campos de uma vez - despesa simples
Histórico de mensagens	[{role: user, content: cadastrar despesa}, {role: assistant, content: Me informe valor, data, cartão e categoria.}]
Mensagem do usuário	"70 reais, ontem, Itaú Black, Borracharia"
Estado inicial esperado	`expense_flow | Aguardando campos.`
Conteúdo esperado na resposta	Confirma cadastro da despesa com valor 70, cartão Itaú Black e categoria Borracharia
Estado final esperado	`idle`

Nesse caso, o agente recebe a mensagem em um estado de fluxo no qual aguarda os dados da despesa. A avaliação verifica se, ao final, a resposta gerada contém a confirmação do cadastro com os valores extraídos corretamente e se o estado da sessão retornou para `idle`, indicando que o fluxo foi concluído. O conjunto completo de casos de teste utilizados em cada versão do sistema é apresentado no Apêndice B.

Os indicadores registrados pelo módulo para cada execução incluem: um sinalizador booleano indicando se o caso foi considerado correto pelo juiz, o tempo total de execução do fluxo em segundos e o número de *tokens* consumidos, decomposto em *tokens* de entrada (*prompt*) e de saída (resposta gerada). Esses dados são exportados em planilhas e formam a base da análise quantitativa da Seção 4.

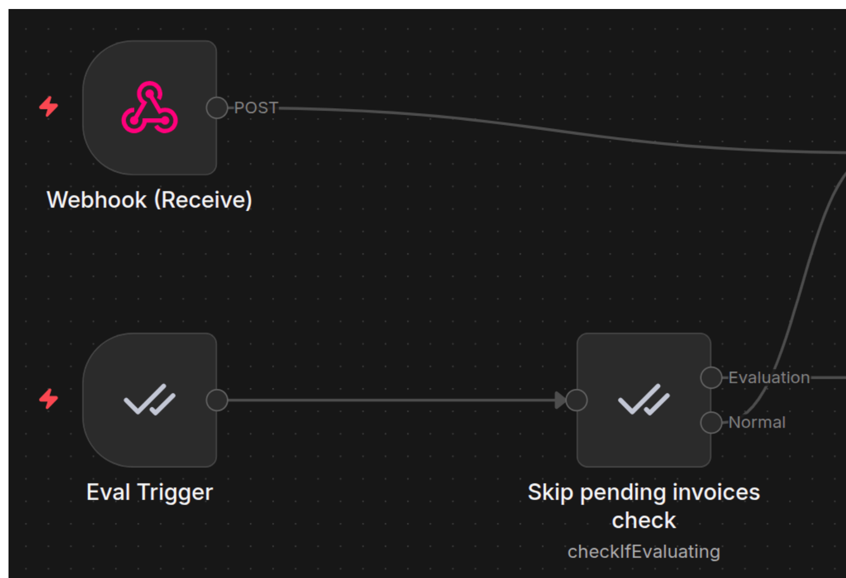

Figura 7. Configuração do gatilho de avaliação no N8N. O nó *Eval Trigger* dispara o *workflow* para cada caso de teste, e o nó condicional subsequente determina quais etapas do fluxo devem ser ignoradas durante a execução em modo de avaliação. Fonte: própria da autora.

3.6. Versão 1 (V1) — Fluxo único

A primeira versão do Finbot foi construída como um único *workflow* no N8N, concentrando em um só fluxo todas as etapas do processamento de uma mensagem: recebimento via *webhook*, identificação do usuário, classificação da intenção, execução da ação e envio da resposta. O resultado foi um fluxo com mais de uma centena de nós interligados, ilustrado de forma parcial na Figura 8.

Nessa arquitetura, o papel da LLM era deliberadamente limitado, atuando somente como um classificador de intenção. A partir da mensagem do usuário, o modelo produzia um rótulo de ação, como `save_expense`, `generate_report` ou `ask_user`, e um nó de roteamento (`switch`) direcionava o fluxo para o ramo correspondente. Toda a lógica de negócio subsequente, incluindo extração de valores e datas, normalização de campos, montagem de consultas ao banco, construção das mensagens de resposta e controle do estado da conversa, era implementada manualmente por meio de nós de código e da manipulação de dados. O estado conversacional entre turnos era persistido nas tabelas `conversation_state`, permitindo que o fluxo retomasse uma interação em andamento, como o preenchimento progressivo dos dados de um cartão.

Mesmo nessa versão centrada em código, alguns pontos já expunham os limites da abordagem. O reconhecimento do cartão mencionado pelo usuário, por exemplo, exigia uma etapa manual de normalização textual seguida de uma busca aproximada, e ainda assim recorria a uma chamada adicional à LLM quando a correspondência era ambígua. De forma semelhante, o controle de sessão dependia de um mecanismo explícito de tempo de expiração (`TTL`): antes de processar cada mensagem, o fluxo verificava se a conversa em andamento havia expirado, com base em um tempo limite configurável, e reiniciava o estado quando necessário. Esses mecanismos, implementados de forma manual, davam indícios de que tarefas de interpretação de linguagem e de gerenciamento de contexto se-

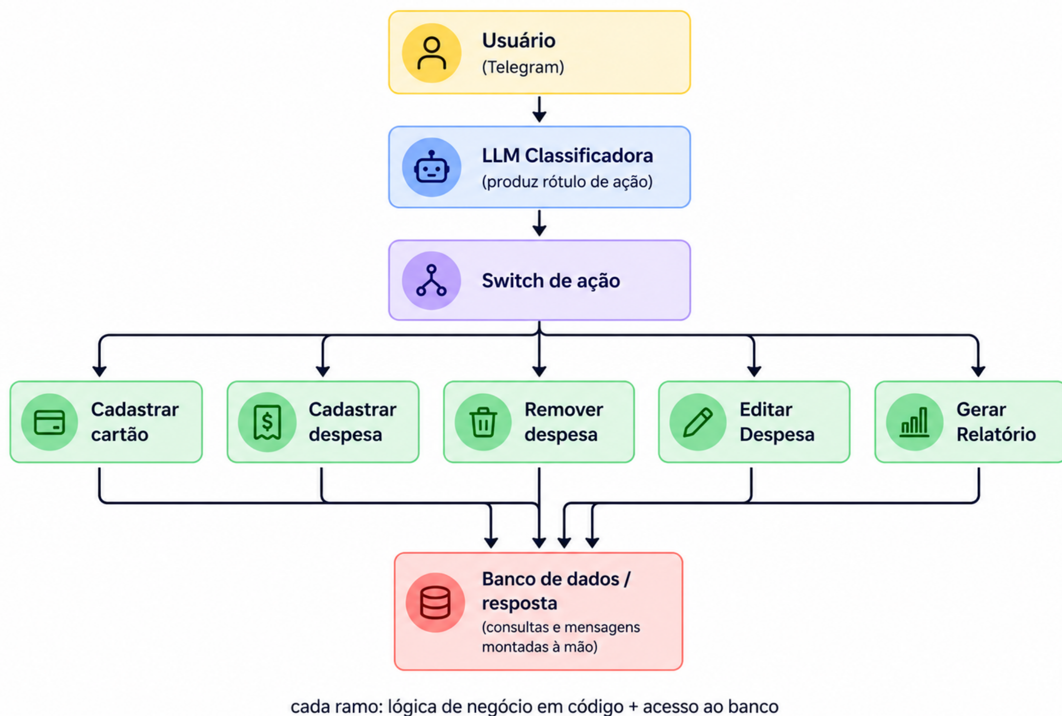

Figura 8. Fluxo lógico da Versão 1 (fluxo único). A mensagem recebida via *webhook* é classificada pela LLM, que produz um rótulo de ação; um nó *switch* roteia para o ramo correspondente, onde a lógica de negócio é tratada manualmente por código antes do acesso ao banco e do envio da resposta. Fonte: própria da autora.

riam melhor tratadas por uma arquitetura que delegasse mais responsabilidade ao modelo.

Essa abordagem funcionou como prova de viabilidade, mas deixou claras algumas limitações. O alto número de nós em um único fluxo resultou em um forte acoplamento: alterar o comportamento de uma funcionalidade exigia navegar por um grafo extenso e compreender as dependências pouco explícitas entre os nós, o que tornava a manutenção e a depuração trabalhosas. Cada nova funcionalidade ampliava a complexidade do mesmo fluxo, em vez de ser adicionada de forma isolada. Além disso, ao restringir a LLM ao papel de classificador, grande parte da capacidade do modelo era subutilizada: tarefas que ele poderia executar de forma flexível, como a extração de múltiplas informações de uma frase, precisavam ser tratadas por código rígido, sensível a variações na forma como o usuário se expressava. Foi isso que levou à reestruturação da segunda versão.

3.7. Versão 2 (V2) — Multi-workflow com Extração por LLM

A segunda versão reestruturou o sistema para resolver o acoplamento da arquitetura monolítica. Em vez de um único fluxo, a lógica foi distribuída em múltiplos *workflows*: um orquestrador central e um sub-*workflow* dedicado a cada funcionalidade (cadastro de despesa, cadastro de cartão, geração de relatório, edição e remoção de despesa). Cada funcionalidade passou a ser desenvolvida, testada e mantida de forma isolada, reduzindo o impacto de alterações sobre o restante do sistema. A Figura 9 apresenta uma visão geral dessa arquitetura.

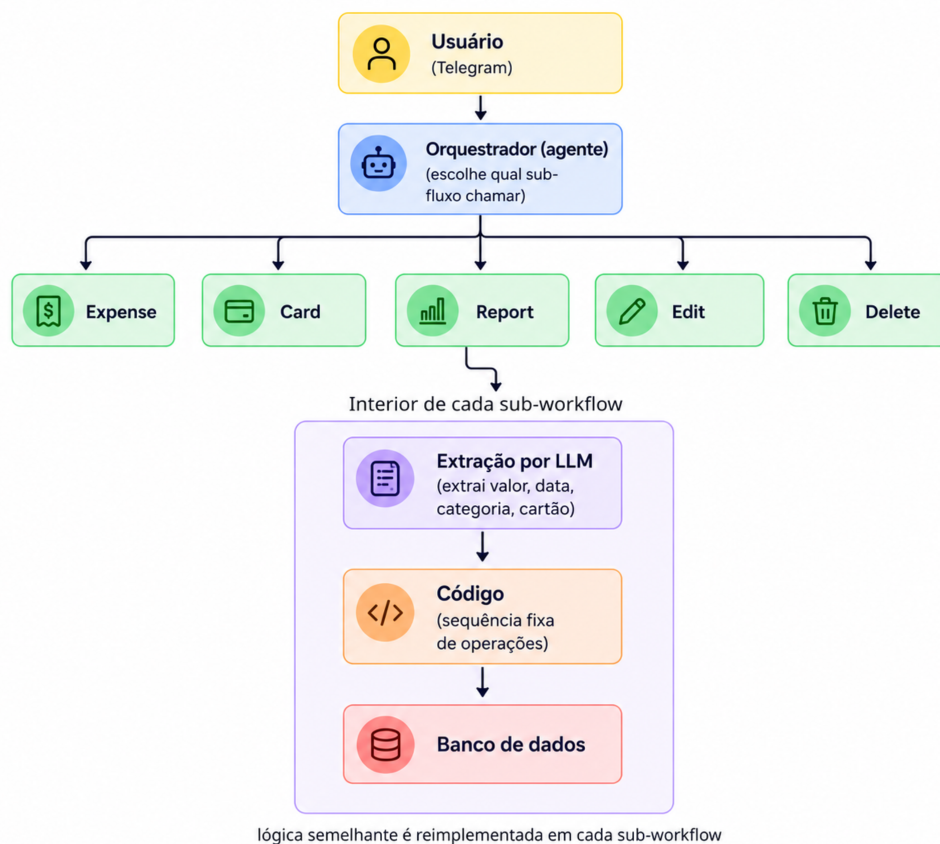

Figura 9. Arquitetura da Versão 2: o orquestrador roteia para sub-*workflows* por funcionalidade. No interior de cada um, uma única extração por LLM é seguida de código que executa as operações no banco. Fonte: própria da autora.

A principal mudança em relação à V1 foi o papel atribuído à LLM. O orquestrador deixou de apenas classificar a intenção e passou a ser construído como um agente de IA: um modelo de linguagem recebia a mensagem do usuário e, com base nas ferramentas disponíveis, decidia autonomamente qual sub-*workflow* acionar. Cada sub-*workflow* era exposto ao agente como uma ferramenta (*tool*), e o roteamento, antes feito por um nó de seleção condicional sobre um rótulo de ação, passou a ser responsabilidade do próprio agente. O estado conversacional e o histórico de mensagens, que na V1 eram mantidos em tabela no banco, passaram a ser gerenciados no Redis, oferecendo acesso de baixa latência ao contexto entre turnos.

O Redis passou a guardar dois tipos de informação: o histórico recente de mensagens da conversa e o estado da sessão (o ponto em que o usuário se encontra em um fluxo, como `expense_flow` ou `card_flow`, junto dos dados parciais já coletados). Vale distinguir esse papel do banco relacional: o PostgreSQL armazena os dados de domínio que precisam ser duráveis e consultáveis a longo prazo, usuários, cartões, faturas e despesas enquanto o Redis guarda o contexto conversacional transitório, que só faz sentido enquanto uma interação está em andamento e pode expirar sem prejuízo. Na V1, esse contexto transitório ficava nas tabelas `conversation_state` do próprio banco; movê-lo

para o Redis reduziu a latência de acesso entre turnos e separou claramente o que é dado de negócio do que é estado de conversa.

Dentro de cada sub-*workflow*, a LLM também assumiu a tarefa de extração de entidades. A partir da mensagem do usuário, um modelo extraía de uma só vez as informações relevantes à operação, como valor, data, categoria e cartão associado a uma despesa, retornando-as em formato estruturado. Isso eliminou boa parte da lógica de normalização manual que a V1 exigia, tornando o sistema mais robusto a variações na forma como o usuário se expressava.

Apesar desses avanços, parte significativa da lógica permaneceu controlada por código. Após a etapa de extração, cada sub-*workflow* processava o resultado por meio de nós de código e de manipulação de dados que decidiam, de forma fixa, como montar e executar as operações no banco: verificar se uma despesa era parcelada, preparar a inserção das parcelas, tratar duplicidade de cartões, montar as consultas de relatório, e assim por diante. Ou seja, embora a LLM já decidisse *o que* fazer e extraísse *quais* dados usar, o *como* executar cada ação continuava rigidamente codificado. Além disso, observou-se uma duplicação considerável de lógica entre os sub-*workflows*: ainda que correspondessem a funcionalidades distintas, muitos compartilhavam sequências de operações semelhantes. O cadastro de uma despesa, por exemplo, precisava verificar se o cartão informado já existia, mesma verificação presente em outros fluxos; e a edição de uma despesa consistia, internamente, em remover o registro existente e inserir uma versão atualizada, reaproveitando a lógica de remoção e de cadastro. Como cada sub-*workflow* implementava essas etapas de forma independente, qualquer ajuste em uma regra comum precisava ser replicado em vários lugares. Esse acoplamento entre a extração e uma sequência fixa de passos, somado à repetição de lógica entre fluxos, limitava a flexibilidade e a manutenibilidade do sistema, o que abriu caminho para a terceira versão.

3.8. Versão 3 (V3) — Multi-agente com Tools Atômicas

No contexto desta versão, adota-se o conceito de ferramentas atômicas: ferramentas com escopo mínimo e bem delimitado, cada uma responsável por uma única ação sobre o sistema, como inserir um registro no banco de dados, consultar o estado da sessão ou formatar uma mensagem de saída. Esse princípio de responsabilidade única no projeto de ferramentas é apontado na literatura como boa prática de engenharia de sistemas agênticos, por reduzir a superfície de erro na seleção de ferramentas pela LLM e por favorecer a composição de operações simples para realizar tarefas complexas [Bandara et al. 2025]. O grão fino de decomposição aumenta a reusabilidade das ferramentas entre diferentes agentes e facilita a depuração, uma vez que cada chamada tem efeito previsível e isolado.

A terceira e a última versão reorganizou o sistema em torno desse conceito, conforme ilustrado na Figura 10. A estrutura de orquestração foi mantida: um agente orquestrador continua recebendo a mensagem do usuário e decidindo qual agente especializado acionar. A diferença fundamental está na camada inferior. Na V2, cada sub-*workflow* fazia uma única extração por LLM e executava o restante por meio de código. Na V3, cada sub-*workflow* passou a ser, ele próprio, um agente de IA dotado de um conjunto de ferramentas atômicas, às quais delega as ações concretas. O sistema é composto por cinco agentes especializados, organizados por domínio de funcionalidade: o agente de despesas (*Expense*), o de cartões (*Card*), o de relatórios (*Report*), o de edição (*Edit*) e o de remoção (*Delete*).

Cada agente dispõe apenas de ferramentas pertinentes ao seu domínio, e decide de forma autônoma quando e com quais argumentos invocá-las, como ilustrado na Figura 11. O agente de cartões, por exemplo, dispõe das ferramentas list_cards e insert_card: ao receber uma mensagem como "cadastre meu cartão Nubank, fechamento dia 5, vencimento dia 12, limite 2000", o agente primeiro aciona a list_cards para verificar os cartões já cadastrados pelo usuário e evitar duplicatas e, não havendo conflito, invoca a insert_card com os dados extraídos da mensagem para efetivar o cadastro. Quando o usuário fornece apenas parte das informações, o próprio agente identifica o que falta e solicita os campos restantes antes de chamar a ferramenta de inserção. Cada ferramenta atômica encapsula uma única operação bem definida e retorna um resultado estruturado: a insert_card, por exemplo, recebe nome, limite, dia de fechamento e dia de vencimento e insere o registro no banco, devolvendo o identificador gerado.

A principal vantagem dessa organização é a reutilização de ferramentas entre agentes, que resolve a duplicação de lógica observada na V2. Ferramentas como list_cards passaram a ser compartilhadas pelos agentes de despesa e de cartão, e as ferramentas de estado temporário redis_set e redis_get são utilizadas tanto pelo agente de edição quanto pelo de remoção, que precisam guardar entre turnos a lista de despesas candidatas a serem editadas ou removidas. A própria edição da despesa, que na V2 exigia código dedicado, foi reformulada como a composição de ferramentas já existentes: o agente de edição lista as despesas, remove a despesa original por meio da delete_expense e cadastra a versão atualizada reutilizando as ferramentas de inserção. Da mesma forma, a geração de relatórios foi dividida entre uma ferramenta que obtém os dados (get_report_data) e outra que formata a mensagem final (build_report), separando a recuperação de dados da apresentação.

Para garantir que a saída do orquestrador pudesse ser processada de forma confiável pelas etapas seguintes, mesmo com a maior delegação de decisões à LLM, adotou-se o padrão de saída estruturada. A resposta do agente é validada contra um esquema predefinido por meio de um analisador de saída estruturada (*structured output parser*); quando a resposta não está em conformidade com o esquema esperado, um segundo modelo, configurado como *parser retry*, é acionado para corrigir e reformatar a saída. Esse mecanismo reduz a fragilidade introduzida pelo não-determinismo do modelo, evitando que respostas mal formatadas interrompam o fluxo.

A construção dos *prompts* seguiu princípios que se consolidaram ao longo das versões e refletem diretamente a arquitetura adotada. O primeiro é a separação de responsabilidades: o *prompt* do orquestrador trata exclusivamente de roteamento, ler o prefixo do state_string, decidir qual agente acionar e repassar a resposta sem reinterpretá-la, enquanto os *prompts* dos agentes especializados concentram a lógica de cada domínio. Essa divisão é a contraparte textual da arquitetura em camadas: cada *prompt* é tão focado quanto o componente que governa, o que reduz o espaço de decisão exposto ao modelo em cada chamada e, como discutido nas lições aprendidas, diminui a chance de alucinação. O segundo princípio é a especificação explícita de invariantes críticas: o formato de saída em *JSON* é repetido e reforçado, o state_string é declarado imutável pelo orquestrador (por ser o que sustenta a continuidade entre turnos), e a obrigação de chamar a *tool* antes de confirmar qualquer operação é afirmada de forma enfática, justamente para conter o risco de o modelo simular um resultado que não ocorreu. O terceiro é

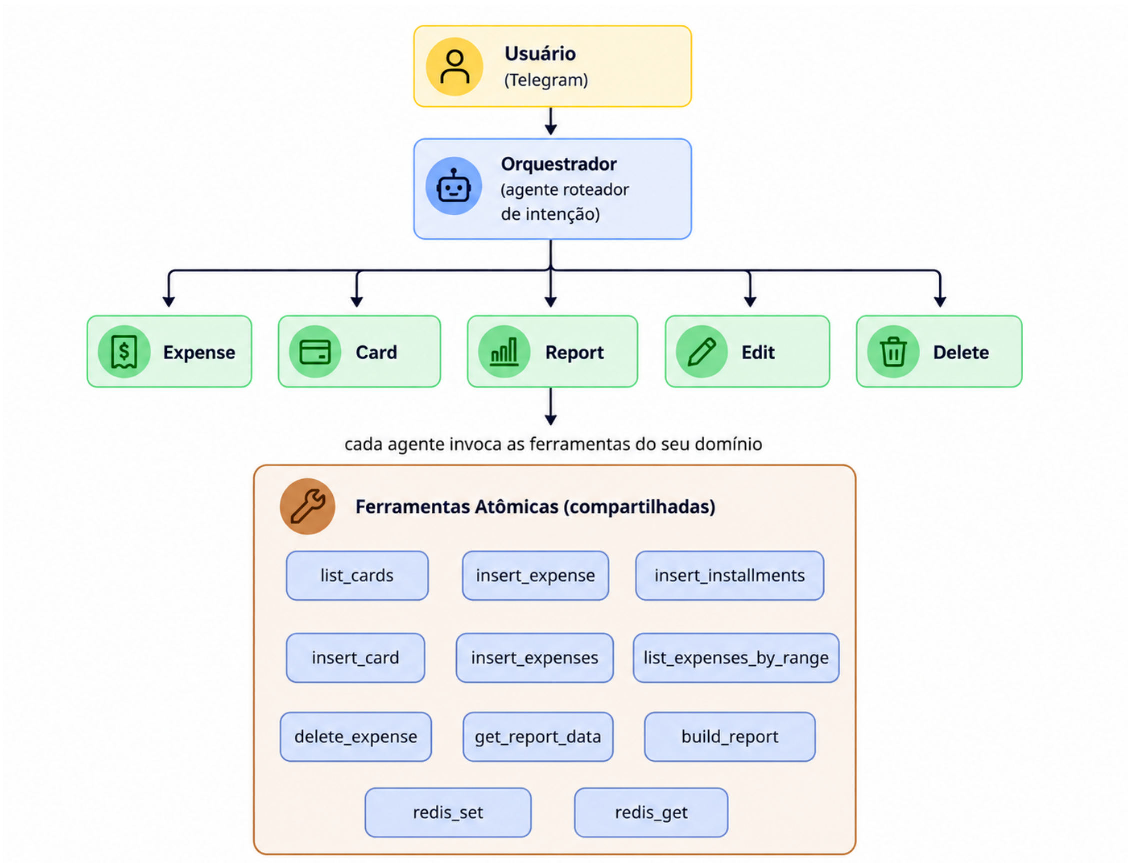

Figura 10. Arquitetura em camadas da Versão 3: o orquestrador roteia a mensagem para um agente especializado, que invoca ferramentas atômicas reutilizáveis. Fonte: própria da autora.

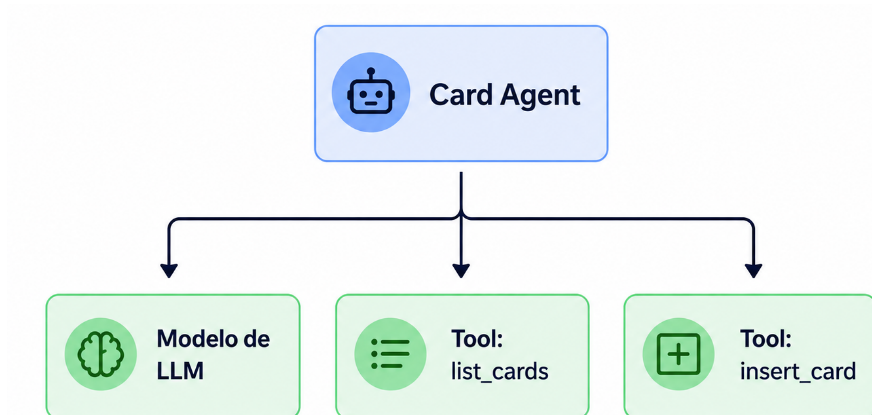

Figura 11. Agente de cartões (*Card Agent*): o modelo de linguagem central decide quando invocar as ferramentas atômicas `list_cards` e `insert_card` para cumprir a solicitação do usuário. Fonte: própria da autora.

a antecipação de casos de borda diretamente no *prompt*, cancelamento no meio do fluxo, cartão inexistente, parcelamento fora do intervalo permitido, desvio para o cadastro de

cartão durante o registro de uma despesa, transferindo para a instrução em linguagem natural um tratamento que, nas versões anteriores, dependia de ramificações codificadas. Essa mudança é o que viabiliza, na prática, a manutenibilidade "via *prompt*" relatada na análise qualitativa: ajustar um comportamento passou a significar, em muitos casos, editar uma instrução textual em vez de reestruturar nós. Os *prompts* completos do orquestrador e do agente de despesas, escolhidos como representativos das duas camadas, estão reproduzidos no Apêndice A.

As operações destrutivas, como a edição e a remoção de despesas, exigiram um cuidado adicional no gerenciamento de estado entre turnos. Em vez de remover um registro imediatamente, o agente correspondente primeiro lista as despesas que correspondem ao pedido do usuário, as armazena temporariamente no Redis como uma lista de candidatas, por meio das ferramentas `redis_set` e `redis_get`, e solicita a confirmação do usuário antes de efetivar a ação. Esse padrão de confirmação em duas etapas evita remoções ou edições acidentais quando há ambiguidade sobre qual despesa o usuário deseja alterar, e ilustra como o estado conversacional de curto prazo, com tempo de expiração definido, é utilizado para coordenar interações que se estendem por mais de uma mensagem.

Essa arquitetura tornou o sistema mais fluido e modular: adicionar uma nova funcionalidade passou a significar, em muitos casos, compor ferramentas atômicas já existentes em um novo agente, em vez de reimplementar lógica do zero. Por outro lado, a maior delegação de decisões à LLM introduziu novos *trade-offs*. Como cada agente realiza múltiplas chamadas ao modelo, uma para decidir qual ferramenta usar e outras a cada passo do raciocínio, o consumo de *tokens* e, consequentemente, o custo e o tempo de resposta tendem a ser maiores do que em uma abordagem controlada por código. Além disso, o sistema passou a depender mais fortemente da capacidade do modelo de escolher e parametrizar corretamente as ferramentas, o que torna a confiabilidade da arquitetura mais sensível à qualidade do modelo empregado. A análise quantitativa desses *trade-offs* é apresentada na Seção 4.

3.9. Visão Geral da Arquitetura

A Figura 12 apresenta uma visão arquitetural simplificada do Finbot, independente das versões implementadas. O fluxo inicia com o envio de mensagens pelo usuário através do Telegram. A mensagem é recebida pelo fluxo de *trigger* para o Telegram, implementado no N8N, que pode ser processado por diferentes versões da arquitetura (V1, V2 ou V3). Durante o processamento, o sistema utiliza mecanismos de memória temporária em Redis, modelos de linguagem (LLM) para interpretação e geração de respostas e um banco de dados PostgreSQL para persistência das informações financeiras. Ao final, a resposta é enviada novamente ao usuário pelo Telegram.

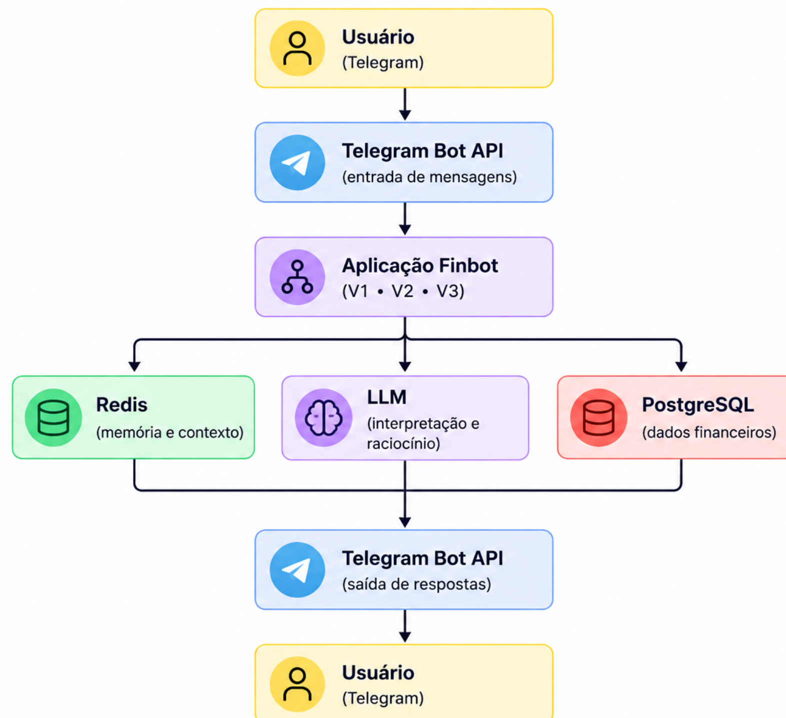

Figura 12. Visão Arquitetural geral do Finbot. Fonte: própria da autora.

4. Resultados e Discussão

Esta seção apresenta os resultados do desenvolvimento e da avaliação das três versões do Finbot, organizados em quatro frentes: a comparação qualitativa entre as versões sob critérios de engenharia de software; os resultados quantitativos do módulo *N8N Evaluations*; as lições aprendidas ao longo do desenvolvimento e demonstrações reais do funcionamento da versão final.

4.1. Comparação Arquitetural Qualitativa

Antes de analisar as métricas coletadas, é útil sintetizar as diferenças arquiteturais entre as três versões do sistema sob critérios de engenharia de software. A Tabela 2 resume essa comparação a partir dos quatro critérios qualitativos definidos na metodologia: manutenibilidade, acoplamento, delegação à LLM e adequação funcional.

Esses critérios, embora baseados em princípios de engenharia de software, são inevitavelmente subjetivos. Por isso, vale ancorar a análise na experiência prática do desenvolvimento das três versões.

Na V1, o alto acoplamento se manifestou de forma evidente: cada nova funcionalidade adicionada exigia revisitar todo o fluxo para evitar quebras retroativas em funcionalidades já implementadas. A manutenibilidade foi igualmente prejudicada, uma vez que identificar a origem de *bugs* ou comportamentos inesperados implicava percorrer toda a cadeia de nós interconectados. A delegação à LLM, apesar de presente, era mínima e mal aproveitada: como todas as funcionalidades coexistiam em um único fluxo, era necessário enviar ao modelo uma quantidade considerável de informações apenas para a classificação da mensagem do usuário. Esses fatores combinados resultaram em uma baixa adequação

Tabela 2. Comparação qualitativa entre as três versões do Finbot. Fonte: própria da autora.

Critério	V1 (Fluxo Único)	V2 (Multi-workflow)	V3 (Multi-agente)
Estrutura	Workflow único com mais de 100 nós	Orquestrador e 5 sub-workflows	Orquestrador, 5 agentes e ferramentas atômicas compartilhadas
Acoplamento	Alto: alterar uma funcionalidade afeta todo o fluxo	Médio: sub-workflows isolados, mas com lógica duplicada entre si	Baixo: ferramentas atômicas reutilizadas entre agentes
Manutenibilidade	Baixa: depuração custosa pela quantidade de nós	Média: depuração por sub-workflow, mas duplicação dificulta evolução	Alta: novas funcionalidades compostas a partir de ferramentas existentes
Delegação à LLM	Baixa: apenas classificação de intenção	Média: extração de entidades nos sub-workflows	Alta: agentes decidem quando e como invocar ferramentas
Adequação funcional	Baixa: respostas engessadas, dependentes do casamento exato de padrões	Intermediária: maior naturalidade na extração, mas execução ainda rígida	Alta: respostas e ações mais alinhadas à intenção expressa pelo usuário

funcional, com o sistema falhando frequentemente em compreender e executar o que era solicitado pelo usuário. Isso motivou a criação da V2.

Na V2, a separação das funcionalidades em sub-*workflows* dedicados trouxe melhorias perceptíveis no acoplamento e na manutenibilidade, ao permitir que cada funcionalidade fosse desenvolvida e depurada de forma isolada. Entretanto, observou-se que muitos nós de código e fluxo de lógica permaneciam duplicados entre os sub-*workflows*, o que dificultava a evolução do sistema: qualquer ajuste em uma regra comum precisava ser replicado em vários lugares. A ampliação do papel da LLM, agora também responsável pela extração de entidades, refletiu positivamente na adequação funcional, mas o modelo ainda estava sendo subutilizado, exercendo essencialmente o papel de classificador de intenção e extrator de informações. Essas limitações motivaram a reformulação proposta na V3.

Na V3, foi dedicado um tempo maior à fase de planejamento, com levantamento das ações que se repetiam entre os fluxos e identificação das que poderiam ser delegadas à LLM. A partir desse mapeamento, as ferramentas atômicas foram construídas a partir de nós já existentes na V2, agora organizadas como peças reutilizáveis. Esse investimento inicial mostrou retorno claro nas etapas seguintes: uma vez disponíveis as ferramentas, a construção dos agentes especializados se tornou consideravelmente mais simples, restando essencialmente especificar no *prompt* de cada agente quais funcionalidades ele deveria atender. A maior delegação à LLM teve impacto direto na manutenibilidade, e ao final do desenvolvimento foi observado que adicionar novas funcionalidades passou a ter uma natureza próxima de *plug-and-play*, com a composição de ferramentas existentes substituindo a reimplementação de lógica. A manutenibilidade passou a residir, em

grande parte, na forma como as instruções eram apresentadas à LLM: ajustes de comportamento se davam principalmente por meio de modificações no *prompt*, e não mais na lógica de código ou na estrutura dos nós.

O padrão entre as versões é claro: cada iteração reduz o acoplamento, amplia o papel da LLM e melhora a adequação funcional. As métricas da próxima seção colocam números nesses ganhos.

4.2. Avaliação Quantitativa via N8N Evaluations

Para avaliar quantitativamente as três versões, foram coletadas métricas de execução por meio do módulo *N8N Evaluations* sobre um conjunto de casos de teste com entradas e respostas esperadas. As métricas analisadas foram: taxa de acerto, calculada pela proporção de casos em que a resposta gerada foi considerada correta por um modelo juiz, conforme o padrão *LLM-as-Judge*; tempo médio de execução por interação em segundos e consumo médio de *tokens* totais utilizados, decomposto também em entrada e saída. Como o número de casos varia entre agentes da V3, foi adotada a média ponderada pelo número de casos para o cálculo do valor agregado de cada versão. Todas as métricas reportadas a seguir foram coletadas com o modelo `gpt-4.1-mini` como modelo de interação dos agentes e com um modelo distinto, o `claude-3-5-haiku`, no papel de juiz, conforme o padrão *LLM-as-Judge*. O uso de um modelo de família diferente para o julgamento é deliberado: reduz o risco de viés de auto-aprimoramento, em que um modelo tende a avaliar de forma mais favorável as respostas geradas por ele mesmo, limitação já apontada na literatura sobre o padrão. A Tabela 3 apresenta os resultados consolidados das três versões.

Tabela 3. Métricas agregadas por versão (média ponderada pelo número de casos). Fonte: própria da autora.

Versão	Acerto (%)	Tempo (s)	Total Tokens	Input Tokens	Output Tokens
V1	9,52	5,86	5.942,57	5.778,48	164,10
V2	42,85	10,04	2.903,33	2.517,41	385,92
V3	75,51	7,27	4.538,47	4.327,63	210,84

O resultado mais expressivo é o crescimento da taxa de acerto entre as versões: a V1 obteve apenas 9,52% de respostas consideradas corretas, a V2 alcançou 42,85% e a V3 chegou a 75,51%. A V3 acerta cerca de oito vezes mais que a V1 nos mesmos tipos de tarefas: quanto mais a LLM assume responsabilidades antes delegadas ao código, mais o sistema se mostra capaz de interpretar entradas variadas em linguagem natural.

O tempo de resposta segue um padrão diferente do esperado. A V1, que é a versão mais dependente de código, leva em média 5,86 segundos por interação. A V2 salta para 10,04 segundos, justamente por introduzir uma extração por LLM em cada sub-*workflow*. Já a V3, mesmo fazendo várias chamadas ao modelo para escolher e parametrizar ferramentas, fica em 7,27 segundos, entre as duas. Esse resultado contraria a expectativa de que mais chamadas à LLM significariam sempre mais tempo: na V2, a combinação de uma extração completa com uma sequência fixa de etapas posteriores acaba acumulando mais latência do que a V3, em que o agente encerra o raciocínio assim que a ferramenta certa é identificada e executada.

Os *tokens* consumidos seguem uma lógica parecida. A V1 gasta em média 5.942 *tokens* por interação, número alto porque seus *prompts* precisam descrever todas as intenções possíveis para que o modelo consiga classificá-las. A V2 baixa esse total para 2.903 *tokens*, já que cada sub-*workflow* trabalha com *prompts* mais enxutos e focados em uma única funcionalidade. A V3 sobe novamente, para 4.538 *tokens*, refletindo o custo das múltiplas chamadas ao modelo por interação, mas ainda fica abaixo da V1. A V2, por sua vez, produz proporcionalmente mais *tokens* de saída do que a V3, o que faz sentido dado que cada sub-*workflow* retornava os dados extraídos em formato estruturado para processamento posterior por código, ao passo que na V3 a composição da resposta final fica a cargo do agente.

Olhando as três métricas em conjunto, o custo adicional da V3 em relação à V2, tanto em tempo quanto em *tokens*, vem acompanhado de um ganho substancial em qualidade: a taxa de acerto quase dobra. Em relação à V1, a comparação é ainda mais favorável: a V3 entrega oito vezes mais respostas corretas com um acréscimo de apenas cerca de um segundo e meio por interação. Ou seja, o custo extra da arquitetura multiagente se justifica quando o objetivo é a qualidade da conversa com o usuário, e não a economia máxima de recursos.

4.3. Lições Aprendidas

O desenvolvimento das três versões do Finbot deixou claro que sistemas baseados em LLMs são um campo ainda em construção. Não existe um caminho único ou definitivo para projetar esse tipo de aplicação: o que se observa, tanto na literatura quanto na prática, é um conjunto de *trade-offs* cuja melhor combinação depende do que se busca em cada sistema. Cada uma das decisões tomadas ao longo do trabalho exigiu equilibrar referências de artigos e relatos práticos com aprendizados extraídos das próprias falhas durante o desenvolvimento.

A V1 pode ser entendida como uma espécie de versão de aprendizado. Naquele momento, o estudo da plataforma N8N e da própria construção de sistemas baseados em LLMs ainda estava em estágio inicial, o que resultou em uma arquitetura desenhada com um olhar muito próximo ao do desenvolvimento tradicional: micro-gerenciamento explícito do estado, controle manual de cada caminho possível e uso da LLM apenas para classificar a intenção. Uma das lições mais importantes dessa fase foi perceber que concentrar todas as intenções e todos os caminhos possíveis em um único *workflow* e em um único *prompt* aumenta consideravelmente a chance de alucinação do modelo, que precisa lidar com um espaço de possibilidades grande demais para uma única decisão.

A V2 corrigiu parte desse problema ao distribuir essas responsabilidades em sub-*workflows* dedicados, o que reduziu a complexidade exposta ao modelo em cada chamada. Por outro lado, ainda persistia um forte micro-gerenciamento do que acontecia após a extração das informações pela LLM, mantendo o modelo subutilizado em relação ao que ele poderia fazer. Ainda assim, essa versão foi nitidamente mais funcional que a anterior, o que reforçou a ideia de que delegar mais responsabilidade ao modelo era um caminho promissor.

A V3, considerada aqui a versão final do trabalho, levou essa delegação a outro patamar: a LLM passou a decidir tanto o quê fazer quanto como fazer, com base nas ferramentas disponíveis e na mensagem do usuário em linguagem natural. Esse ganho

de autonomia, porém, trouxe um novo conjunto de desafios. Aumentar a delegação ao modelo significa abrir mão de parte do controle determinístico sobre o fluxo, o que torna o teste e a validação atividades centrais: é preciso verificar repetidamente se o modelo está escolhendo as ferramentas corretas e usando-as com os argumentos certos. A expressão "versão final" aparece entre aspas neste trabalho de forma deliberada, já que um sistema dessa natureza dificilmente está de fato concluído: sempre há um detalhe a melhorar com base em algo recém-aprendido ou em uma técnica recém-publicada.

Ao longo do desenvolvimento, vários problemas concretos foram enfrentados e resolvidos de forma incremental, e cada um deles contribuiu para a forma final da arquitetura. As decisões tomadas em cada versão emergiram tanto da leitura de trabalhos relacionados quanto da observação direta das falhas que iam aparecendo, o que reforça a natureza experimental desse tipo de desenvolvimento.

Por fim, a escolha de infraestrutura também se mostrou um fator relevante no fluxo de desenvolvimento. A opção pela modalidade *self-hosted*, embora exija um esforço inicial maior de configuração, oferece em troca controle total sobre o ambiente e maior flexibilidade para customizar aspectos específicos do sistema quando necessário, o que se mostrou valioso ao longo das iterações.

4.4. Demonstração do Finbot

Esta subseção apresenta interações reais com a versão final do Finbot, ilustrando como a arquitetura multi-agente com ferramentas atômicas se materializa do ponto de vista do usuário. Os exemplos a seguir foram extraídos de conversas no Telegram e cobrem as principais funcionalidades do sistema, demonstrando ações que internamente envolvem múltiplas chamadas a ferramentas, manutenção de estado entre turnos e validação de dados estruturados, mas que aparecem ao usuário final apenas como uma conversa em linguagem natural.

- **Cadastro de cartão**: A Figura 13 mostra o cadastro de um cartão de crédito por meio de uma conversa. O agente de cartões coleta as informações necessárias (nome, limite, dia de fechamento e dia de vencimento) e confirma o cadastro ao final. Quando o usuário fornece apenas parte dos dados em uma mensagem, o agente identifica o que está faltando e solicita os campos restantes, sem reiniciar o fluxo. Caso o usuário envie um nome que já corresponde a um cartão cadastrado, o agente alerta sobre a duplicidade antes de prosseguir, evitando inserções acidentais.
- **Cadastro de despesa única**: Na Figura 14, o usuário registra uma despesa enviando todas as informações relevantes em uma única mensagem. O agente de despesas extrai simultaneamente o valor, a data, o cartão e a categoria e aciona a ferramenta `insert_expense` para registrar a transação. Esse cenário ilustra um dos principais ganhos da arquitetura multi-agente em relação às versões anteriores: o usuário não precisa preencher cada campo separadamente nem seguir uma ordem rígida, podendo expressar a despesa de forma livre.
- **Cadastro de despesa parcelada**: A Figura 15 ilustra um cenário em que o usuário informa uma compra parcelada. O agente identifica a presença do número de parcelas e, em vez de acionar a ferramenta de inserção simples, invoca a `insert_installments`, que se encarrega de dividir o valor total em parcelas e associá-las às faturas corretas. Cada parcela é tratada como uma despesa

individual no banco de dados, vinculada à compra original por meio do campo `purchase_id`, e alocada na fatura correspondente ao seu ciclo de cobrança. A decisão entre cadastrar uma despesa simples ou parcelada é feita autonomamente pelo agente com base na mensagem do usuário, sem que seja necessário acionar um comando específico para cada caso.

- **Geração de relatório**: A Figura 16 mostra a resposta gerada pelo agente de relatórios quando o usuário solicita um resumo da fatura. A composição entre as ferramentas `get_report_data`, que recupera os dados, e `build_report`, que formata a saída final, resulta em uma mensagem estruturada por categorias. Essa separação de responsabilidades entre uma ferramenta de recuperação e outra de apresentação ilustra na prática o princípio de ferramentas atômicas adotado nesta versão: cada operação tem um escopo bem delimitado e pode ser reutilizada ou recombinada conforme a necessidade. O agente escolhe entre apresentar um relatório do mês corrente ou um relatório completo do cartão com base na mensagem do usuário, e em casos de cartões com nomes ambíguos solicita o esclarecimento antes de prosseguir.
- **Remoção com confirmação**: Por fim, a Figura 17 ilustra o padrão de confirmação em duas etapas adotado para operações destrutivas. O agente de remoção primeiro lista as despesas candidatas, armazena-as temporariamente no Redis e solicita a confirmação do usuário antes de efetivar a ação. Esse cuidado adicional foi pensado para evitar remoções acidentais quando há ambiguidade sobre qual despesa o usuário deseja apagar, especialmente em casos em que a busca retorna múltiplos resultados.

No conjunto, os exemplos mostram que a arquitetura proposta é funcional do ponto de vista do usuário final: ações que envolvem extração de informação estruturada, composição de ferramentas e gerenciamento de estado entre turnos são executadas de forma natural, sem a necessidade de comandos rígidos ou interfaces engessadas. A complexidade interna do sistema, com múltiplos agentes coordenados por um orquestrador e ferramentas atômicas compartilhadas entre eles, fica integralmente abstraída em uma conversa que se aproxima da forma como um usuário descreveria suas finanças a outra pessoa.

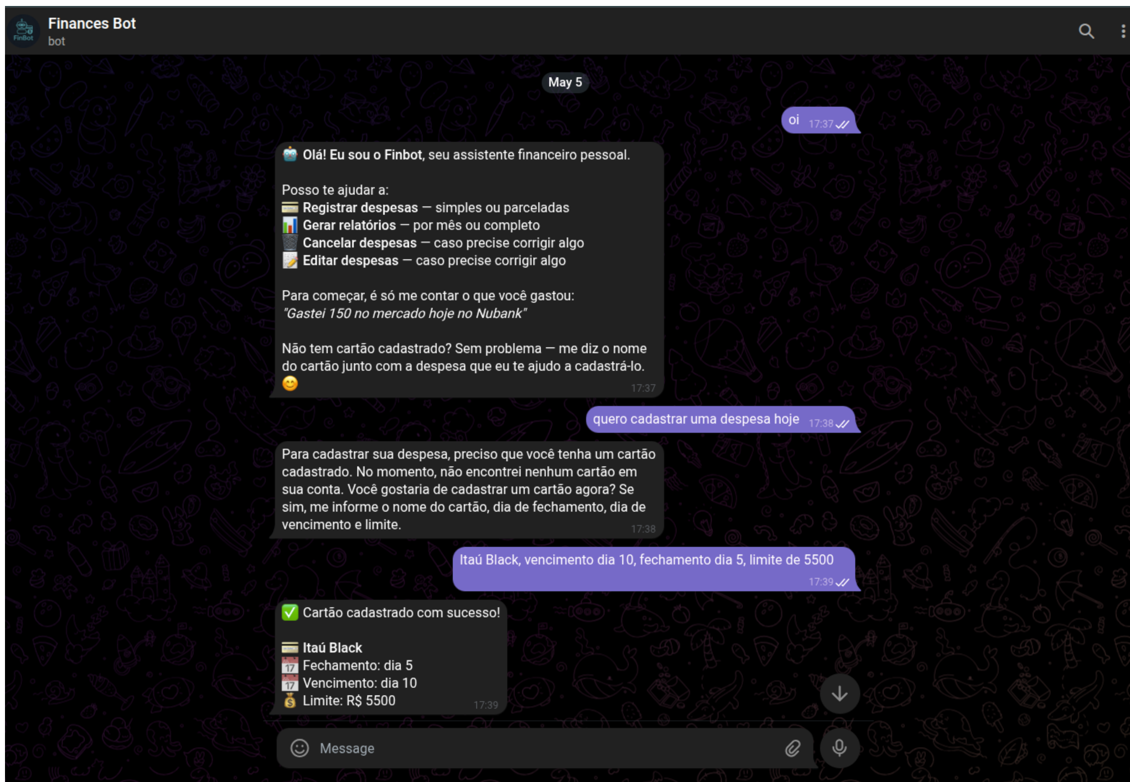

Figura 13. Cadastro de cartão de crédito por meio de conversa em linguagem natural. Fonte: própria da autora.

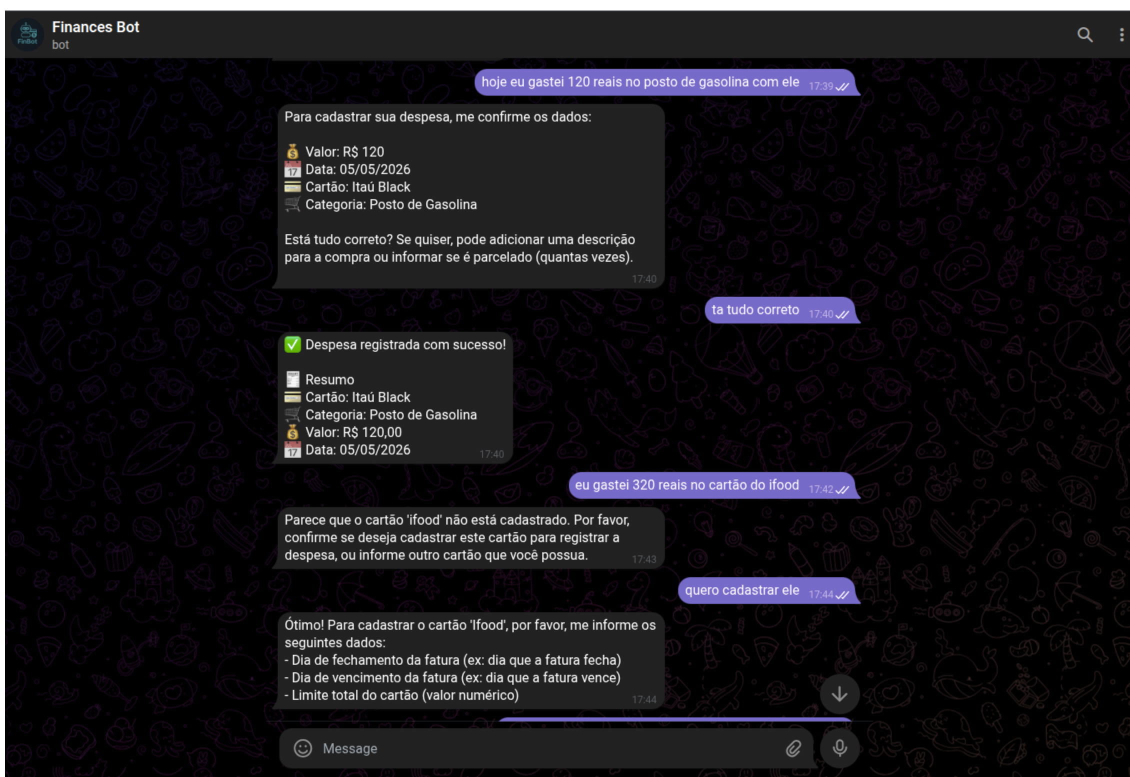

Figura 14. Cadastro de despesa por meio de conversa em linguagem natural. Fonte: própria da autora.

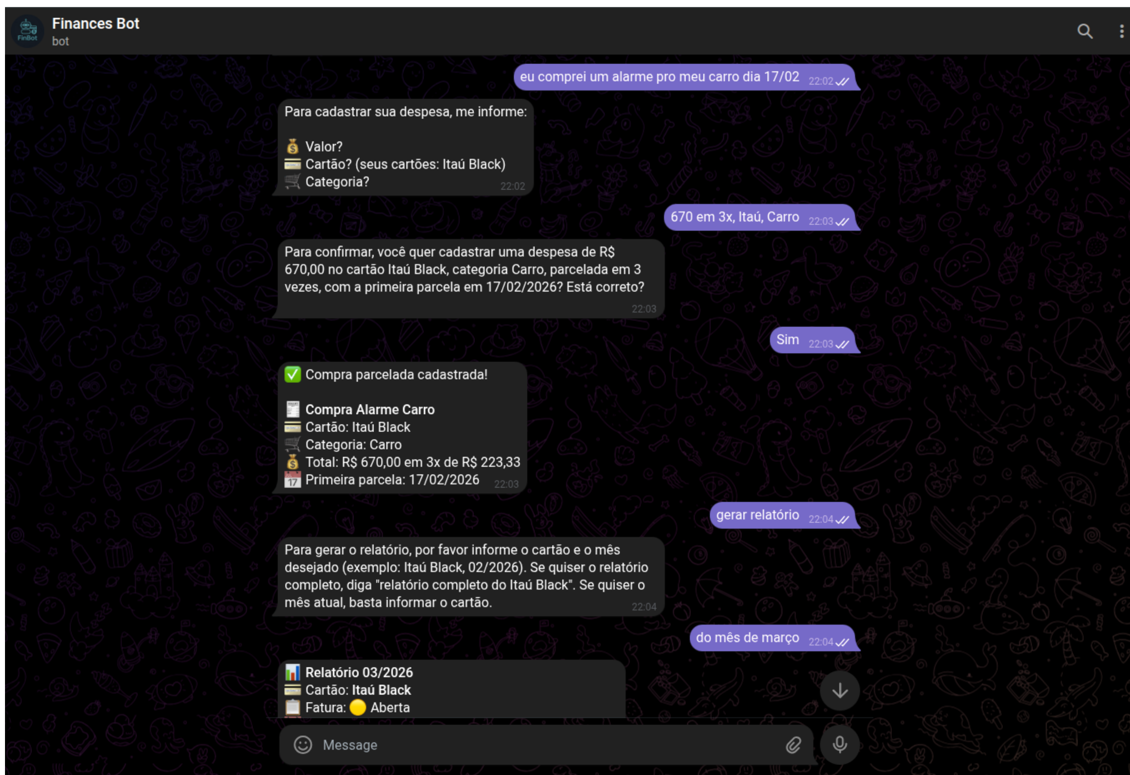

Figura 15. Cadastro de despesa parcelada por meio de conversa em linguagem natural. Fonte: própria da autora.

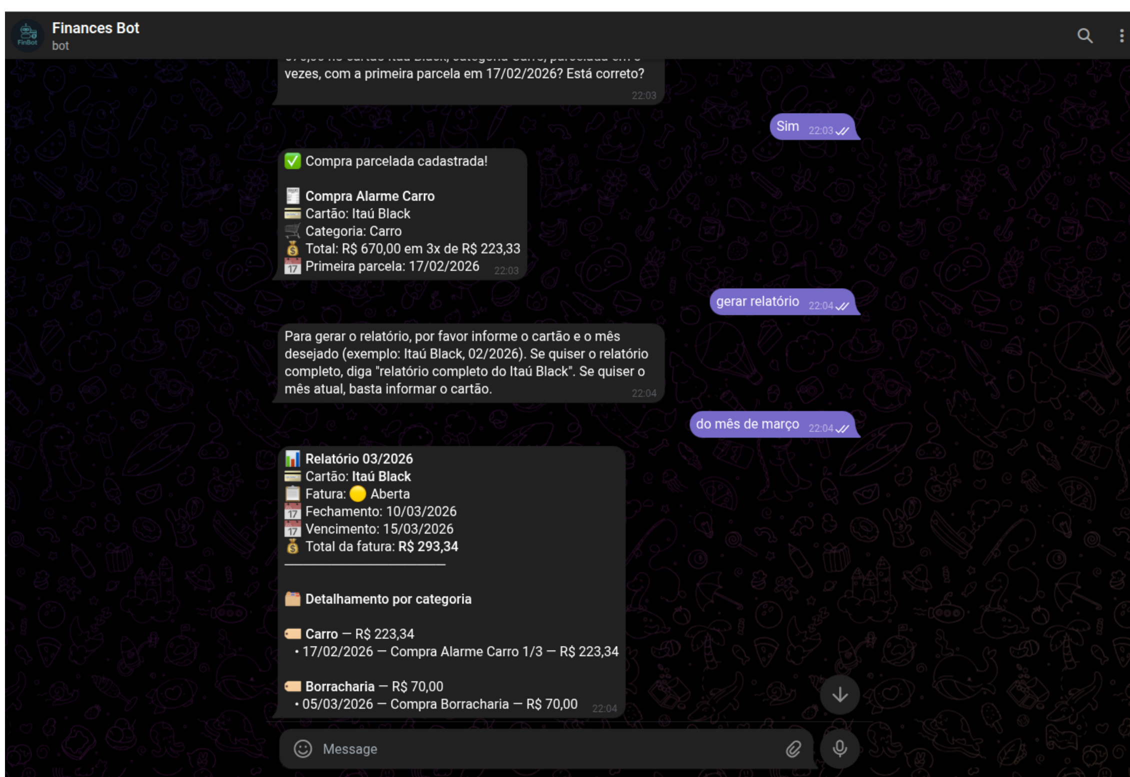

Figura 16. Geração de relatório por meio de conversa em linguagem natural. Fonte: própria da autora.

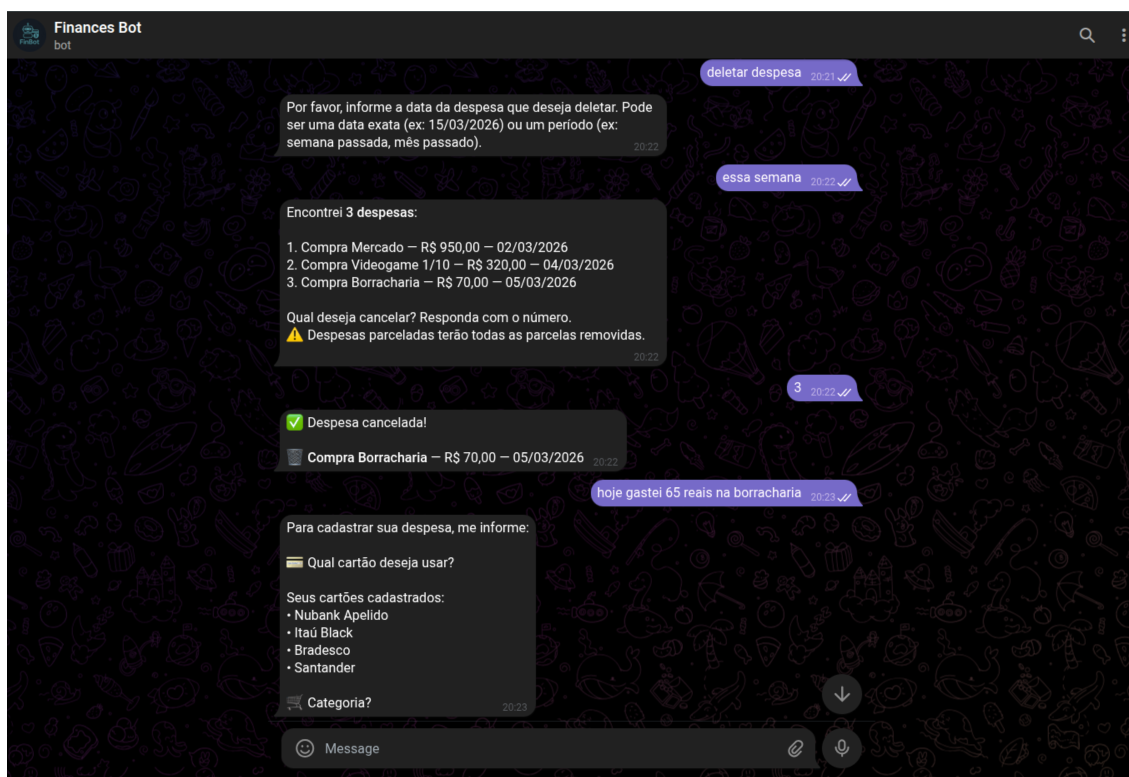

Figura 17. Remoção com confirmação por meio de conversa em linguagem natural. Fonte: própria da autora.

5. Conclusão

Este trabalho investigou os *trade-offs* arquiteturais no desenvolvimento iterativo de um assistente conversacional baseado em LLMs, documentando a construção de um assistente conversacional de finanças ao longo de três versões arquiteturais progressivas: um sistema monolítico, uma arquitetura baseada em múltiplos *workflows* com extração de entidades por LLM, e uma arquitetura multi-agente com ferramentas atômicas.

Cada iteração trouxe ganhos claros em acoplamento, manutenibilidade e adequação funcional. A taxa de acerto cresceu de 9,52% na V1 para 75,51% na V3, um ganho de cerca de oito vezes nos mesmos tipos de tarefa, enquanto o tempo médio de resposta da versão final ficou apenas pouco mais de um segundo acima da V1, mesmo com a maior delegação de decisões à LLM. A V3 também consumiu menos *tokens* em média que a V1, sinal de que aumentar o papel do modelo não implica necessariamente maior custo operacional, desde que a arquitetura distribua as decisões em *prompts* mais focados.

No lado qualitativo, delegar mais responsabilidade à LLM tende a melhorar a adequação funcional do sistema, mas essa delegação tem um custo que vai além do consumo de *tokens* e precisa ser reconhecido. Transferir ao modelo a decisão do “quê” e “como” executar significa abrir mão de garantias determinísticas: o agente pode selecionar a ferramenta errada, parametrizá-la com argumentos incorretos, ignorar uma etapa de confirmação ou produzir uma saída fora do esquema esperado, e essas falhas nem sempre se manifestam de forma evidente. Em operações destrutivas, como a remoção de uma despesa, um erro de delegação pode ter efeito irreversível sobre os dados do usuário. Por isso,

quanto mais se delega, mais indispensável se torna a camada de contenção ao redor do modelo (*harness*): validação de saída estruturada, confirmação em duas etapas para ações sensíveis, restrição do conjunto de ferramentas disponível a cada agente e, sobretudo, um regime contínuo de testes que verifique se o modelo continua escolhendo e usando as ferramentas corretamente. A maior adequação funcional observada na V3 não é, portanto, um efeito automático da delegação, mas o resultado de delegação acompanhada desses mecanismos de controle. Também ficou claro que a escolha da plataforma de orquestração e da modalidade de infraestrutura tem impacto direto no ritmo de iteração: o uso do N8N em modalidade *self-hosted* permitiu prototipar e ajustar rapidamente as três versões sem o custo recorrente de assinatura da modalidade *cloud*, ao preço de uma configuração inicial mais trabalhosa.

O trabalho tem limitações que vale registrar. A avaliação quantitativa usou um único provedor de LLM, o que impede afirmações empíricas sobre o impacto de diferentes modelos ou provedores no desempenho do sistema. A migração do Groq para o OpenRouter partiu de observações práticas durante o desenvolvimento, mas não foram coletadas métricas comparativas entre os dois. Além disso, o sistema não foi testado com usuários reais em condições de uso contínuo, e a operação dos modelos depende de provedores externos cobrados por *tokens*, o que gera um custo recorrente e cria uma dependência que pode afetar a viabilidade do sistema em uso prolongado.

Três frentes se abrem para a continuidade do trabalho. Uma é avaliar o Finbot com usuários reais em produção, medindo aspectos como facilidade de uso, frequência de erros e satisfação que dificilmente são capturados por casos de teste sintéticos. Vale também uma comparação empírica entre diferentes provedores e modelos de LLM aplicados à mesma arquitetura, permitindo quantificar o impacto da escolha do modelo sobre o desempenho do sistema. Por fim, expandir o conjunto de funcionalidades financeiras (metas de gastos, alertas automáticos, novas categorias de transações) ampliaria o escopo do assistente permitindo testar a escalabilidade da arquitetura proposta diante de demandas mais diversas.

Referências

Anthropic (2024). The Claude 3 model family: Opus, sonnet, haiku. Disponível em: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf. Acesso em: 06 jun. 2026.

Bandara, E., Gore, R., Foytik, P., Shetty, S., Mukkamala, R., Rahman, A., Liang, X., Bouk, S. H., Hass, A., Rajapakse, S., Keong, N. W., De Zoysa, K., Withanage, A., and Loganathan, N. (2025). A practical guide for designing, developing, and deploying production-grade agentic AI workflows. *arXiv preprint arXiv:2512.08769*. Disponível em: https://arxiv.org/abs/2512.08769.

CNC (2026). Pesquisa de endividamento e inadimplência do consumidor (PEIC). Disponível em: https://pesquisascnc.com.br/pesquisa-peic/. Acesso em: 17 mar. 2026.

CNDL, SPC Brasil, and Offerwise (2023). Mais da metade dos brasileiros não controlam gastos com cartão de crédito. Disponível em: https:

//www.cdlpalmas.com.br/conteudo/mais-da-metade-dos-brasileiros-nao-controlam-gastos-com-cartao-de-credito-1. Acesso em: 17 mar. 2026.

DeepSeek (2024). DeepSeek: Open-source language models. Disponível em: https://www.deepseek.com/. Acesso em: 06 jun. 2026.

Gartner (2021). Gartner forecasts worldwide low-code development technologies market to grow 23% in 2021. Disponível em: https://www.gartner.com/en/newsroom/press-releases/2021-02-15-gartner-forecasts-worldwide-low-code-development-technologies-market-to-grow-23-percent-in-2021. Acesso em: 06 mai. 2026.

Google and DeepMind (2023). Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*. Disponível em: https://arxiv.org/abs/2312.11805.

Groq (2024). Groqcloud documentation. Disponível em: https://groq.com/groqcloud. Acesso em: 26 mai. 2026.

Hu, K. (2023). ChatGPT sets record for fastest-growing user base — analyst note. Reuters. Disponível em: https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/. Acesso em: 06 jun. 2026.

IBM (2024a). What is generative AI? Disponível em: https://www.ibm.com/think/topics/generative-ai. Acesso em: 26 mai. 2026.

IBM (2024b). What is LangChain? Disponível em: https://www.ibm.com/think/topics/langchain. Acesso em: 14 mai. 2026.

IstoÉ Dinheiro (2026). Brasil passa a ter mais cartões de crédito do que habitantes; veja números. Disponível em: https://istoedinheiro.com.br/brasil-cartoes-de-credito-habitantes. Acesso em: 29 mai. 2026.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*. Disponível em: https://arxiv.org/abs/2001.08361. Acesso em: 04 jun. 2026.

Kapoor, A. et al. (2026). AI agent systems: Architectures, applications, and evaluation. *arXiv preprint arXiv:2601.01743*. Disponível em: https://arxiv.org/abs/2601.01743. Acesso em: 15 mai. 2026.

LangChain (2024). LangGraph documentation. Disponível em: https://langchain-ai.github.io/langgraph/. Acesso em: 06 jun. 2026.

Meta AI (2024). Llama: Open foundation language models. Disponível em: https://www.llama.com/. Acesso em: 06 jun. 2026.

n8n-io (2024). n8n: Fair-code workflow automation platform with native AI capabilities. Disponível em: https://github.com/n8n-io/n8n. Acesso em: 10 mai. 2026.

OpenAI (2023). GPT-4 technical report. *arXiv preprint arXiv:2303.08774*. Disponível em: https://arxiv.org/abs/2303.08774.

OpenRouter (2024). Openrouter documentation. Disponível em: https://openrouter.ai/docs. Acesso em: 26 mai. 2026.

Redis (2024a). How to build AI agents with Redis memory management. Disponível em: https://redis.io/blog/build-smarter-ai-agents-manage-short-term-and-long-term-memory-with-redis/. Acesso em: 14 mai. 2026.

Redis (2024b). Llm session memory. Disponível em: https://redis.io/docs/latest/develop/ai/redisvl/user_guide/session_manager/. Acesso em: 14 mai. 2026.

Stanford HAI (2025). The 2025 AI index report. Technical report, Stanford Institute for Human-Centered AI (HAI). Disponível em: https://hai.stanford.edu/ai-index/2025-ai-index-report. Acesso em: 12 mai. 2026.

Supabase (2024). Supabase: The postgres development platform. Disponível em: https://github.com/supabase/supabase. Acesso em: 14 mai. 2026.

Telegram (2024a). Marvin's marvellous guide to all things webhook. Disponível em: https://core.telegram.org/bots/webhooks. Acesso em: 14 mai. 2026.

Telegram (2024b). Telegram bot API. Disponível em: https://core.telegram.org/bots/api. Acesso em: 14 mai. 2026.

Topsakal, O. and Akinci, T. C. (2024). LangChain: A powerful framework for LLM applications. *ResearchGate preprint*. Disponível em: https://www.researchgate.net/publication/385681151_LangChain. Acesso em: 02 jun. 2026.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Disponível em: https://arxiv.org/abs/1706.03762. Acesso em: 20 mai. 2026.

Yao, S., Zhao, Jeffrey Ext, Y. D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*. Disponível em: https://arxiv.org/abs/2210.03629. Acesso em: 22 mai. 2026.

ZenML (2024). LLM agents in production: Architectures, challenges, and best practices. Disponível em: https://www.zenml.io/blog/llm-agents-in-production-architectures-challenges-and-best-practices. Acesso em: 28 abr. 2026.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, Dacheng Caps, X. E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. (2023). Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS)*. Disponível em: https://arxiv.org/abs/2306.05685. Acesso em: 25 mai. 2026.

A. Prompts dos Agentes

Esta seção reproduz dois dos *prompts* utilizados na Versão 3 do Finbot, escolhidos por representarem as duas camadas da arquitetura multi-agente: o *prompt* do agente orquestrador, responsável apenas pelo roteamento entre os agentes especializados, e o *prompt* do

agente de despesas (*Expense Agent*), responsável pela lógica de um domínio específico. As expressões entre chaves duplas correspondem a variáveis interpoladas pelo N8N em tempo de execução, como a data atual, os identificadores do usuário e o histórico da conversa.

```
Voce e o FINBOT Orchestrator Agent, responsavel por decidir qual tool chamar para responder o usuario de um app de controle financeiro pessoal.

IMPORTANTE: Se o estado indica uma conversa em andamento, PRIORIZE SEMPRE continuar esse fluxo. Se tiver duvida sobre a intencao: retorne o texto de
saudacao como fallback.

=== REPASSE DE RESPOSTAS ===
Quando uma tool (expense_tool, card_tool, report_tool, delete_tool,
edit_tool) retornar um objeto com `response` e `state_string`, repasse
EXATAMENTE esses valores no seu output.
NUNCA reescreva, resuma ou interprete a resposta da tool.
NUNCA informe erro ao usuario se a tool nao retornou um campo `error`
explicito. O seu papel e rotear, nao reinterpretar.

Leia o prefixo do state_string para priorizar o flow em andamento:
- Comeca com "expense_flow |" = chame expense_tool
- Comeca com "card_flow |"    = chame card_tool
- Comeca com "report_flow |"  = chame report_tool
- Comeca com "delete_flow |"  = chame delete_tool
- Comeca com "edit_flow |"    = chame edit_tool
- Comeca com "idle" ou vazio  = classifique pela mensagem normalmente

Classificacao pela mensagem:
- expense_tool = gasto, compra, valor, parcelado, ou resposta no meio de um cadastro de despesa (ex: informando cartao, categoria, data)
- card_tool = cadastrar cartao, fechamento, vencimento, limite, ou resposta no meio de um cadastro de cartao
- report_tool = relatorio, resumo, quanto gastei
- delete_tool = deletar, remover, apagar despesa, ou confirmacao no meio de um cancelamento
- edit_tool = corrigir, retificar, alterar despesa, mudar despesa, ou resposta no meio de uma edicao
- Saudacao ou texto claramente nao financeiro = NAO chame nenhuma tool, mas AINDA ASSIM retorne o JSON de saudacao.

Apos receber o retorno de qualquer tool:
```

```
31 - O retorno vira como JSON com os campos "response" e "
      state_string"
32 - Retorne EXATAMENTE: {"response": "<response>", "state_string":
       "<state_string EXATO>"}
33 - O state_string NUNCA pode ser modificado, copie o valor exato,
       ele e critico para o funcionamento do sistema
34 - O response pode receber emojis adicionais SOMENTE se estiver
      sem emojis, mas o texto nao pode ser alterado
35 - NUNCA retorne apenas o texto do response sem o state_string
36 - NUNCA escreva nada fora do JSON
37
38 IMPORTANTE: Se o state_string comeca com qualquer prefixo de
      flow, voce
39 DEVE chamar a tool correspondente. NUNCA trate a mensagem
      diretamente
40 quando ha um flow em andamento.
41
42 data atual: {{ data atual }}
43 user_id: {{ user_id }}
44 chat_id: {{ chat_id }}
45 estado atual da conversa: {{ state_string }}
46 historico recente: {{ history }}
```

Código 3. Prompt do agente orquestrador da Versão 3.

```
1  Voce e o Expense Agent do Finbot, responsavel por cadastrar
      despesas (simples ou parceladas).
2
3  Voce recebe: user_id, chat_id, state_string (estado atual, pode
      conter dados parciais), message (ultima mensagem) e history (
      historico recente).
4
5  Seu fluxo:
6  1. Use `list_cards` para buscar os cartoes do usuario.
7  2. Extraia da mensagem: valor, descricao, data, cartao,
      categoria, se e parcelado e quantas parcelas.
8  3. Se algum dado obrigatorio (valor, cartao, data, categoria)
      estiver faltando, pergunte ao usuario e atualize o
      state_string com os dados coletados ate agora.
9  4. A categoria pode ser presumida pela mensagem (merchant como
      Comper, Posto Shell, iFood, ou algo generico como mercado,
      farmacia). Sempre em Title Case.
10 5. Se o cartao mencionado nao existir nos resultados de
      list_cards, informe o usuario e sugira os cartoes disponiveis
       ou ofereca cadastrar.
11 6. Quando tiver todos os dados:
12    - OBRIGATORIAMENTE chame a tool antes de retornar qualquer
         resposta
13    - So retorne a mensagem de sucesso apos o retorno real da
         tool
```

```
14     - Parcelado = `insert_installments`; Simples = `
         insert_expense`
15  7. Retorne SEMPRE um JSON: { "response": "...", "state_string":
       "..." }
16
17  === CADASTRO DE CARTAO NO MEIO DO FLUXO ===
18  Se o usuario pedir para cadastrar cartao com uma despesa em
       andamento: nao chame tool de insercao, salve os dados da
       despesa no state_string com prefixo card_flow e oriente o
       cadastro do cartao. Ao retornar com prefixo expense_flow,
       recupere os dados pendentes e retome a despesa.
19
20  === VALIDACAO DE PARCELAS ===
21  - number_installments deve ser entre 2 e 48
22  - 1x = trate como despesa simples (insert_expense)
23  - 0x ou negativo = "O numero de parcelas deve ser entre 2 e
       48..."
24  - mais de 48x = "O maximo de parcelas permitido e 48..."
25
26  === DATAS ===
27  Converta internamente para YYYY-MM-DD antes de chamar tools, mas
        NUNCA exiba datas nesse formato ao usuario, use sempre DD/MM
       /YYYY. Resolva referencias relativas (hoje, ontem, anteontem,
        dias da semana) com base na data atual fornecida no prompt.
28
29  === CONFIANCA NAS TOOLS ===
30  NUNCA simule ou assuma o resultado de uma operacao. Voce DEVE
       chamar insert_expense ou insert_installments e aguardar o
       retorno antes de confirmar qualquer cadastro. Sucesso sem
       chamada previa a tool e falha grave.
31
32  === CANCELAMENTO ===
33  Se o usuario disser "cancelar", "esquece", "para", "deixa", "nao
        quero mais", "desistir" ou similar: encerre sem chamar tool,
        state_string = "idle | Nenhuma conversa em andamento.", e
       responda cordialmente.
34
35  === FORMATO DE SAIDA OBRIGATORIO ===
36  Sua mensagem final DEVE ser SEMPRE e SOMENTE este JSON, sem
       texto antes ou depois: { "response": "...", "state_string":
       "..." }
37  Mesmo que a tool ja tenha retornado uma mensagem pronta, envolva
       -a no JSON.
38
39  data atual: {{ data atual }}
40  user_id: {{ user_id }}
41  chat_id: {{ chat_id }}
42  estado atual: {{ state_string }}
43  historico: {{ history }}
```

Código 4. Prompt do agente de despesas (*Expense Agent*) da Versão 3.

B. Casos de Teste das Avaliações

Esta seção apresenta os casos de teste utilizados nas avaliações automatizadas das três versões do Finbot por meio do módulo *N8N Evaluations*. Para cada caso são informados o tipo de agente avaliado, uma descrição do cenário, a mensagem do usuário, o estado inicial da sessão, o conteúdo esperado na resposta e o estado final esperado após o processamento.

B.1. Versão 1 (V1)

A Tabela 4 lista os 21 casos de teste utilizados na avaliação da Versão 1.

Tabela 4. Casos de teste da Versão 1. Fonte: autoria da autora.

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
card	Cancelamento mid-flow — deve encerrar sem chamar tool	esquece	card_flow	Encerra o cadastro sem criar o cartão e se coloca à disposição	idle
card	Campos parciais — deve perguntar o que falta	XP Visa, limite 8000	card_flow	Solicita o fechamento e vencimento que estão faltando	card_flow
card	Ambiguidade fechamento/vencimento — deve perguntar qual é qual	Bradesco, dia 10, limite 2000	card_flow	Pergunta se o dia 10 é de fechamento ou vencimento pois não ficou claro	card_flow
card	Inserção completa — deve cadastrar Eval Novo Card com sucesso	Eval Novo Card, fechamento dia 6, vencimento dia 10, limite 3400	card_flow	Confirma criação do Eval Novo Card com fechamento dia 6, vencimento dia 10 e limite 3400	`idle`
expense	Todos os campos de uma vez — despesa simples	70 reais, ontem, Itaú Black, Borracharia	expense_flow	Confirma cadastro da despesa com valor 70, cartão Itaú Black e categoria Borracharia	idle

continua na próxima página

Tabela 4 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
expense	Campos faltando — cartão e categoria	150 reais, hoje	expense_flow	Pergunta qual cartão usar e qual a categoria	expense_flow
expense	Cancelamento mid-flow	esquece	expense_flow	Encerra sem cadastrar e se coloca à disposição	idle
expense	Cartão não encontrado	50 reais, hoje, Bradesco, Farmácia	expense_flow	Informa que o cartão Bradesco não foi encontrado e lista os cartões disponíveis	expense_flow
expense	Despesa parcelada válida — 10x	500 reais, hoje, Itaú Black, Eletrônicos, 10 vezes	expense_flow	Confirma cadastro parcelado com 10 parcelas, valor total 500 e cartão Itaú Black	idle
expense	Parcelamento 1x — deve tratar como despesa simples	200 reais, hoje, Itaú Black, Roupas, 1 vez	expense_flow	Trata como despesa simples e confirma o cadastro normalmente	idle
expense	Parcelamento acima de 48x — deve recusar	1000 reais, hoje, Itaú Black, Móveis, 60 vezes	expense_flow	Informa que o máximo é 48 parcelas e pede quantas parcelas deseja	expense_flow
expense	Desvio para cadastro de cartão no meio do fluxo	cadastrar cartão	expense_flow	Salva os dados da despesa pendente e redireciona para o cadastro de cartão	card_flow
edit	Sem data — deve perguntar quando foi	quero editar aquela compra do mercado	idle	Pergunta quando foi a compra ou pede uma referência de data	edit_flow
edit	Cancelamento mid-flow — deve encerrar sem alterar	esquece	edit_flow	Informa que a edição foi cancelada e a despesa não foi alterada	idle

continua na próxima página

Tabela 4 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
edit	Muitos resultados — deve pedir mais detalhes	quero editar uma despesa do mês passado	idle	Pergunta mais detalhes sobre a despesa para filtrar os resultados	edit_flow
edit	Data exata — deve encontrar fixture e mostrar dados atuais	quero editar a despesa de hoje	idle	Mostra os dados atuais da Despesa Eval Edit e pergunta o que deseja alterar	edit_flow
delete	Sem data — deve perguntar quando foi	quero apagar aquela compra do mercado	idle	Pergunta quando foi a compra ou pede uma referência de data	delete_flow
delete	Referência de período — deve buscar e listar ou pedir mais detalhes	semana passada tinha uma despesa de farmácia	idle	Pergunta mais detalhes ou lista as despesas encontradas no período	delete_flow
delete	Negação — não deve deletar e deve encerrar	não	delete_flow	Informa que a despesa não foi removida	idle
delete	Muitos resultados — deve pedir mais detalhes antes de listar	quero apagar uma despesa do mês passado	idle	Pergunta mais detalhes sobre a despesa para filtrar os resultados	delete_flow
delete	Data exata — deve encontrar fixture e pedir confirmação	quero apagar a despesa de hoje	idle	Mostra Despesa Eval Delete com valor 99,90 e pede confirmação para cancelar	delete_flow

B.2. Versão 2 (V2)

A Tabela 5 lista os 49 casos de teste utilizados na avaliação da Versão 2, distribuídos entre o orquestrador e os cinco sub-*workflows* dedicados a cada funcionalidade.

Tabela 5. Casos de teste da Versão 2. Fonte: autoria da autora.

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
orchestrator	Roteamento para expense_tool — mensagem de gasto simples	gastei 80 reais no mercado hoje no Nubank	idle — Nenhuma conversa em andamento.	Retorna a mensagem de boas-vindas do Finbot	Nenhum
orchestrator	Roteamento para card_tool — mensagem de cadastro de cartão	quero cadastrar um cartão	idle — Nenhuma conversa em andamento.	Retorna a mensagem de boas-vindas do Finbot	Nenhum
orchestrator	Roteamento para report_tool — pedido de relatório	quero ver quanto gastei esse mês	idle — Nenhuma conversa em andamento.	Apresenta ou inicia a geração do relatório, ou informa que é necessário cadastrar um cartão antes de gerar relatórios	Nenhum
orchestrator	Roteamento para delete_tool — pedido de remoção	quero apagar uma despesa	idle — Nenhuma conversa em andamento.	Pergunta qual despesa remover ou inicia o fluxo de remoção	delete_flow
orchestrator	Roteamento para edit_tool — pedido de edição	preciso corrigir uma despesa	idle — Nenhuma conversa em andamento.	Pergunta qual despesa editar ou inicia o fluxo de edição	edit_flow
orchestrator	State expense_flow — mensagem ambígua deve continuar no flow	sim	expense_flow — valor=150 aguardando=cartão	Continua o fluxo de despesa pedindo ou confirmando o cartão	expense_flow

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
orchestrator	State card_flow — mensagem que parece ser de despesa deve continuar no card flow	Nubank, fechamento 2, vencimento 8	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Continua o fluxo de cartão processando os dados informados	card_flow
orchestrator	State delete_flow — confirmação deve continuar no delete flow	pode apagar	delete_flow — Aguardando confirmação para deletar despesa id=abc123	Confirma a remoção da despesa	idle
orchestrator	State edit_flow — resposta do usuário deve continuar no edit flow	muda para 200 reais	edit_flow — Aguardando novo valor para despesa id=xyz456	Continua o fluxo de edição processando o novo valor informado	edit_flow
orchestrator	Saudação — não deve chamar nenhuma tool	oi tudo bem	idle — Nenhuma conversa em andamento.	Retorna a mensagem de boas-vindas do Finbot	idle
orchestrator	Mensagem fora de escopo — não deve chamar nenhuma tool	qual a previsão do tempo hoje	idle — Nenhuma conversa em andamento.	Retorna a mensagem de boas-vindas sem tentar responder sobre previsão do tempo	idle

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
orchestrator	State expense_flow — mensagem parece cancelamento mas o estado não é idle	esquece	expense_flow — valor=300 aguardando=cartão	Encerra o fluxo de despesa e se coloca à disposição	idle
expense	Todos os campos de uma vez — despesa simples	70 reais, ontem, Itaú Black, Borracharia	expense_flow — Aguardando campos.	Confirma cadastro da despesa com valor 70, cartão Itaú Black e categoria Borracharia	idle
expense	Campos faltando — cartão e categoria	150 reais, hoje	expense_flow — Aguardando campos.	Pergunta qual cartão usar e qual a categoria	expense_flow
expense	Cancelamento mid-flow	esquece	expense_flow — valor=80 cartão=nubank aguardando=categoria	Encerra sem cadastrar e se coloca à disposição	idle
expense	Cartão não encontrado	50 reais, hoje, Bradesco, Farmácia	expense_flow — Aguardando campos.	Informa que o cartão Bradesco não foi encontrado e lista os cartões disponíveis	expense_flow
expense	Despesa parcelada válida — 10x	500 reais, hoje, Itaú Black, Eletrônicos, 10 vezes	expense_flow — Aguardando campos.	Confirma cadastro parcelado com 10 parcelas, valor total 500 e cartão Itaú Black	idle
expense	Parcelamento 1x — deve tratar como despesa simples	200 reais, hoje, Itaú Black, Roupas, 1 vez	expense_flow — Aguardando campos.	Trata como despesa simples e confirma o cadastro normalmente	idle
expense	Parcelamento acima de 48x — deve recusar	1000 reais, hoje, Itaú Black, Móveis, 60 vezes	expense_flow — Aguardando campos.	Informa que o máximo é 48 parcelas e pede quantas parcelas deseja	expense_flow

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
expense	Desvio para cadastro de cartão no meio do fluxo	cadastrar cartão	expense_flow — valor=300 aguardando=cartão	Salva os dados da despesa pendente e redireciona para o cadastro de cartão	card_flow
card	Cancelamento mid-flow — deve encerrar sem chamar tool	esquece	card_flow — Cadastro em andamento: Nubank. Falta: fechamento, vencimento, limite.	Encerra o cadastro sem criar o cartão e se coloca à disposição	idle
card	Campos parciais — deve perguntar o que falta	XP Visa, limite 8000	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Solicita o fechamento e vencimento que estão faltando	card_flow
card	Ambiguidade fechamento/vencimento — deve perguntar qual é qual	Bradesco, dia 10, limite 2000	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Pergunta se o dia 10 é de fechamento ou vencimento pois não ficou claro	card_flow

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
card	Inserção completa — deve cadastrar Eval Novo Card com sucesso	Eval Novo Card, fechamento dia 6, vencimento dia 10, limite 3400	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Confirma criação do Eval Novo Card com fechamento dia 6, vencimento dia 10 e limite 3400	idle
card	Duplicata — deve informar que Eval Fixture Card já existe	Eval Fixture Card, fechamento 5, vencimento 12, limite 1000	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Informa que já existe um cartão com esse nome e pede outro nome ou confirmação de cancelamento	card_flow
card	Inserção com despesa pendente — deve voltar ao expense_flow	Eval Novo Card, fechamento 2, vencimento 8, limite 6000	card_flow — Despesa em andamento: valor=150 categoria=Alimentação — Cadastro: . Falta: nome, fechamento, vencimento, limite.	Confirma criação do cartão e avisa que está voltando para o cadastro da despesa pendente	expense_flow
report	Sem cartão — deve sugerir cadastrar	quero ver meu relatório	idle — Nenhuma conversa em andamento.	Informa que não há cartões cadastrados e sugere cadastrar um	idle

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
report	Cartão não encontrado — deve listar disponíveis	relatório do Bradesco	idle — Nenhuma conversa em andamento.	Informa que Bradesco não foi encontrado e lista os cartões disponíveis incluindo Itaú Black Eval	report_flow
report	Cartão ambíguo — deve perguntar qual	relatório do itaú	idle — Nenhuma conversa em andamento.	Pergunta qual cartão Itaú deseja usar listando as opções	report_flow
report	Relatório mensal — deve gerar relatório do mês atual	relatório do Itaú Black Eval desse mês	idle — Nenhuma conversa em andamento.	Apresenta relatório do Itaú Black Eval com a despesa Despesa Eval Report de R$ 250	idle
report	Relatório completo — deve listar todas as despesas do cartão	relatório completo do Itaú Black Eval	idle — Nenhuma conversa em andamento.	Apresenta relatório completo do Itaú Black Eval incluindo Despesa Eval Report	idle
report	Mês sem despesas — deve informar que não há despesas no período	relatório do Itaú Black Eval de 01/2000	idle — Nenhuma conversa em andamento.	Informa que não há despesas para o período informado	idle
report	Mid-flow — cartão já identificado aguardando período	esse mês	report_flow — Cartão: Itaú Black Eval (id: {{fixture_card_id}}). Aguardando: período.	Apresenta o relatório do mês atual do Itaú Black Eval	idle
edit	Sem data — deve perguntar quando foi	quero editar aquela compra do mercado	idle — Nenhuma conversa em andamento.	Pergunta quando foi a compra ou pede uma referência de data	edit_flow

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
edit	Cancelamento mid-flow — deve encerrar sem alterar	esquece	edit_flow — Aguardando: data da despesa.	Informa que a edição foi cancelada e a despesa não foi alterada	idle
edit	Muitos resultados — deve pedir mais detalhes	quero editar uma despesa do mês passado	idle — Nenhuma conversa em andamento.	Pergunta mais detalhes sobre a despesa para filtrar os resultados	edit_flow
edit	Data exata — deve encontrar fixture e mostrar dados atuais	quero editar a despesa de hoje	idle — Nenhuma conversa em andamento.	Mostra os dados atuais da Despesa Eval Edit e pergunta o que deseja alterar	edit_flow
edit	Escolha de número — deve recuperar do Redis e mostrar dados	1	edit_flow — Múltiplos candidatos. Aguardando: escolha.	Mostra os dados atuais da despesa escolhida e pergunta o que deseja alterar	edit_flow
edit	Edição completa — deve deletar e recriar com novos dados	muda o valor para 200 e a categoria para Transporte	edit_flow — Coletando alterações. Original: {{fixture_id}}.	Confirma que a despesa foi corrigida com valor 200 e categoria Transporte	idle
edit	Nenhuma despesa encontrada — deve informar e encerrar	quero editar a despesa do dia 01/01/2000	idle — Nenhuma conversa em andamento.	Informa que não encontrou nenhuma despesa nessa data e se coloca à disposição	idle
delete	Sem data — deve perguntar quando foi	quero apagar aquela compra do mercado	idle — Nenhuma conversa em andamento.	Pergunta quando foi a compra ou pede uma referência de data	delete_flow

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
delete	Referência de período — deve buscar e listar ou pedir mais detalhes	semana passada tinha uma despesa de farmácia	idle — Nenhuma conversa em andamento.	Pergunta mais detalhes ou lista as despesas encontradas no período	delete_flow
delete	Cancelamento mid-flow — deve encerrar imediatamente sem chamar tool	esquece	delete_flow — Aguardando: data da despesa.	Informa que a despesa não foi removida	idle
delete	Negação — não deve deletar e deve encerrar	não	delete_flow — Candidatos salvos. Aguardando: confirmação.	Informa que a despesa não foi removida	idle
delete	Muitos resultados — deve pedir mais detalhes antes de listar	quero apagar uma despesa do mês passado	idle — Nenhuma conversa em andamento.	Pergunta mais detalhes sobre a despesa para filtrar os resultados	delete_flow
delete	Data exata — deve encontrar fixture e pedir confirmação	quero apagar a despesa de hoje	idle — Nenhuma conversa em andamento.	Mostra Despesa Eval Delete com valor 99,90 e pede confirmação para cancelar	delete_flow
delete	Confirmação — deve chamar delete_expense e encerrar	sim	delete_flow — Candidatos salvos. Aguardando: confirmação.	Confirma que a despesa Despesa Eval Delete foi cancelada com valor 99,90	idle
delete	Escolha de número na lista — deve recuperar do Redis e deletar	1	delete_flow — Múltiplos candidatos. Aguardando: escolha.	Confirma o cancelamento de Despesa Eval Delete	idle

continua na próxima página

Tabela 5 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
delete	Nenhuma despesa encontrada — deve informar e encerrar	quero apagar a despesa do dia 01/01/2000	idle — Nenhuma conversa em andamento.	Informa que não encontrou nenhuma despesa nessa data e se coloca à disposição	idle

B.3. Versão 3 (V3)

A Tabela 6 lista os 49 casos de teste utilizados na avaliação da Versão 3, distribuídos entre o orquestrador e os cinco agentes especializados que compõem a arquitetura multi-agente.

Tabela 6. Casos de teste da Versão 3. Fonte: autoria da autora.

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
orchestrator	Roteamento para expense_tool — mensagem de gasto simples	gastei 80 reais no mercado hoje no Nubank	idle — Nenhuma conversa em andamento.	Confirma o cadastro da despesa, pede mais informações sobre ela, ou informa que é necessário cadastrar um cartão antes de registrar despesas	Nenhum
orchestrator	Roteamento para card_tool — mensagem de cadastro de cartão	quero cadastrar um cartão	idle — Nenhuma conversa em andamento.	Solicita os dados do cartão ou inicia o fluxo de cadastro	card_flow
orchestrator	Roteamento para report_tool — pedido de relatório	quero ver quanto gastei esse mês	idle — Nenhuma conversa em andamento.	Apresenta ou inicia a geração do relatório, ou informa que é necessário cadastrar um cartão antes de gerar relatórios	idle ou Nenhum

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
orchestrator	Roteamento para delete_tool — pedido de remoção	quero apagar uma despesa	idle — Nenhuma conversa em andamento.	Pergunta qual despesa remover ou inicia o fluxo de remoção	delete_flow
orchestrator	Roteamento para edit_tool — pedido de edição	preciso corrigir uma despesa	idle — Nenhuma conversa em andamento.	Pergunta qual despesa editar ou inicia o fluxo de edição	edit_flow
orchestrator	State expense_flow — mensagem ambígua deve continuar no flow	sim	expense_flow — valor=150 aguardando=cartão	Continua o fluxo de despesa pedindo ou confirmando o cartão	expense_flow
orchestrator	State card_flow — mensagem que parece ser de despesa deve continuar no card flow	Nubank, fechamento 2, vencimento 8	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Continua o fluxo de cartão processando os dados informados	card_flow
orchestrator	State delete_flow — confirmação deve continuar no delete flow	pode apagar	delete_flow — Aguardando confirmação de remoção de despesa.	Confirma a remoção da despesa	idle
orchestrator	State edit_flow — resposta do usuário deve continuar no edit flow	muda para 200 reais	edit_flow — Aguardando novo valor para despesa id=xyz456	Continua o fluxo de edição processando o novo valor informado	edit_flow

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
orchestrator	Saudação — não deve chamar nenhuma tool	oi tudo bem	idle — Nenhuma conversa em andamento.	Retorna a mensagem de boas-vindas do Finbot	idle
orchestrator	Mensagem fora de escopo — não deve chamar nenhuma tool	qual a previsão do tempo hoje	idle — Nenhuma conversa em andamento.	Retorna a mensagem de boas-vindas sem tentar responder sobre previsão do tempo	idle
orchestrator	State expense_flow — mensagem parece cancelamento mas o estado não é idle	esquece	expense_flow — valor=300 aguardando=cartão	Encerra o fluxo de despesa e se coloca à disposição	idle
expense	Todos os campos de uma vez — despesa simples	70 reais, ontem, Itaú Black, Borracharia	expense_flow — Aguardando campos.	Confirma cadastro da despesa com valor 70, cartão Itaú Black e categoria Borracharia	idle
expense	Campos faltando — cartão e categoria	150 reais, hoje	expense_flow — Aguardando campos.	Pergunta qual cartão usar e qual a categoria	expense_flow
expense	Cancelamento mid-flow	esquece	expense_flow — valor=80 cartão=nubank aguardando=categoria	Encerra sem cadastrar e se coloca à disposição	idle
expense	Cartão não encontrado	50 reais, hoje, Bradesco, Farmácia	expense_flow — Aguardando campos.	Informa que o cartão Bradesco não foi encontrado e lista os cartões disponíveis	expense_flow

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
expense	Despesa parcelada válida — 10x	500 reais, hoje, Itaú Black, Eletrônicos, 10 vezes	expense_flow — Aguardando campos.	Confirma cadastro parcelado com 10 parcelas, valor total 500 e cartão Itaú Black	idle
expense	Parcelamento 1x — deve tratar como despesa simples	200 reais, hoje, Itaú Black, Roupas, 1 vez	expense_flow — Aguardando campos.	Trata como despesa simples e confirma o cadastro normalmente	idle
expense	Parcelamento acima de 48x — deve recusar	1000 reais, hoje, Itaú Black, Móveis, 60 vezes	expense_flow — Aguardando campos.	Informa que o máximo é 48 parcelas e pede quantas parcelas deseja	expense_flow
expense	Desvio para cadastro de cartão no meio do fluxo	cadastrar cartão	expense_flow — valor=300 aguardando=cartão	Salva os dados da despesa pendente e redireciona para o cadastro de cartão	card_flow
card	Cancelamento mid-flow — deve encerrar sem chamar tool	esquece	card_flow — Cadastro em andamento: Nubank. Falta: fechamento, vencimento, limite.	Encerra o cadastro sem criar o cartão e se coloca à disposição	idle
card	Campos parciais — deve perguntar o que falta	XP Visa, limite 8000	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Solicita o fechamento e vencimento que estão faltando	card_flow

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
card	Ambiguidade fechamento/vencimento — deve perguntar qual é qual	Bradesco, dia 10, limite 2000	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Pergunta se o dia 10 é de fechamento ou vencimento pois não ficou claro	card_flow
card	Inserção completa — deve cadastrar Eval Novo Card com sucesso	Eval Novo Card, fechamento dia 6, vencimento dia 10, limite 3400	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Confirma criação do Eval Novo Card com fechamento dia 6, vencimento dia 10 e limite 3400	idle
card	Duplicata — deve informar que Eval Fixture Card já existe	Eval Fixture Card, fechamento 5, vencimento 12, limite 1000	card_flow — Cadastro em andamento: . Falta: nome, fechamento, vencimento, limite.	Informa que já existe um cartão com esse nome e pede outro nome ou confirmação de cancelamento	card_flow

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
card	Inserção com despesa pendente — deve voltar ao expense_flow	Eval Novo Card, fechamento 2, vencimento 8, limite 6000	card_flow — Despesa em andamento: valor=150 categoria=Alimentação — Cadastro: . Falta: nome, fechamento, vencimento, limite.	Confirma criação do cartão e avisa que está voltando para o cadastro da despesa pendente	expense_flow
report	Sem cartão — deve sugerir cadastrar	quero ver meu relatório	idle — Nenhuma conversa em andamento.	Solicita que o usuário informe ou selecione um cartão para gerar o relatório	report_flow
report	Cartão não encontrado — deve listar disponíveis	relatório do Bradesco	idle — Nenhuma conversa em andamento.	Informa que Bradesco não foi encontrado e lista os cartões disponíveis incluindo Itaú Black Eval	report_flow
report	Cartão ambíguo — deve perguntar qual	relatório do itaú	idle — Nenhuma conversa em andamento.	Pergunta se quer o relatório completo ou o do mês atual para o cartão cadastrado	report_flow
report	Relatório mensal — deve gerar relatório do mês atual	relatório do Itaú Black Eval desse mês	idle — Nenhuma conversa em andamento.	Apresenta um resumo ou relatório de despesas do cartão solicitado, mesmo que a fatura exata do mês não seja encontrada	idle

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	**Descrição**	**Mensagem**	**Estado Inicial**	**Esperado na Resposta**	**Estado Final**
report	Relatório completo — deve listar todas as despesas do cartão	relatório completo do Itaú Black Eval	idle — Nenhuma conversa em andamento.	Apresenta o relatório completo do cartão OU informa claramente que não existem despesas encontradas	idle
report	Mês sem despesas — deve informar que não há despesas no período	relatório do Itaú Black Eval de 01/2000	idle — Nenhuma conversa em andamento.	Informa que não encontrou despesas ou fatura para o período solicitado e evita inventar dados inexistentes	idle
report	Mid-flow — cartão já identificado aguardando período	esse mês	report_flow — Cartão: Itaú Black Eval (id: {{fixture_card_id}}) Aguardando: período.	Gera ou apresenta o relatório do cartão previamente selecionado sem pedir novamente o cartão	idle
edit	Sem data — deve perguntar quando foi	quero editar aquela compra do mercado	idle — Nenhuma conversa em andamento.	Pergunta quando foi a compra ou pede uma referência de data	edit_flow
edit	Cancelamento mid-flow — deve encerrar sem alterar	esquece	edit_flow — Aguardando: data da despesa.	Informa que a edição foi cancelada e a despesa não foi alterada	`idle`
edit	Muitos resultados — deve pedir mais detalhes	quero editar uma despesa do mês passado	idle — Nenhuma conversa em andamento.	Pergunta mais detalhes sobre a despesa para filtrar os resultados	edit_flow
edit	Data exata — deve encontrar fixture e mostrar dados atuais	quero editar a despesa de hoje	idle — Nenhuma conversa em andamento.	Mostra os dados atuais da Despesa Eval Edit e pergunta o que deseja alterar	edit_flow

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
edit	Escolha de número — deve recuperar do Redis e mostrar dados	1	edit_flow — Múltiplos candidatos. Aguardando: escolha.	Mostra os dados atuais da despesa escolhida e pergunta o que deseja alterar	edit_flow
edit	Edição completa — deve deletar e recriar com novos dados	muda o valor para 200 e a categoria para Transporte	edit_flow — Coletando alterações. Original: {{fixture_id}}.	Confirma que a despesa foi corrigida com valor 200 e categoria Transporte	idle
edit	Nenhuma despesa encontrada — deve informar e encerrar	quero editar a despesa do dia 01/01/2000	idle — Nenhuma conversa em andamento.	Informa que não encontrou nenhuma despesa nessa data e se coloca à disposição	idle
delete	Sem data — deve perguntar quando foi	quero apagar aquela compra do mercado	idle — Nenhuma conversa em andamento.	Pergunta quando foi a compra ou pede uma referência de data	delete_flow
delete	Referência de período — deve buscar e listar ou pedir mais detalhes	semana passada tinha uma despesa de farmácia	idle — Nenhuma conversa em andamento.	Pergunta mais detalhes ou lista as despesas encontradas no período	delete_flow
delete	Cancelamento mid-flow — deve encerrar imediatamente sem chamar tool	esquece	delete_flow — Aguardando: data da despesa.	Informa que a despesa não foi removida	idle
delete	Negação — não deve deletar e deve encerrar	não	delete_flow — Candidatos salvos. Aguardando: confirmação.	Informa que a despesa não foi removida	idle

continua na próxima página

Tabela 6 – continuação da página anterior

Agente	Descrição	Mensagem	Estado Inicial	Esperado na Resposta	Estado Final
delete	Muitos resultados — deve pedir mais detalhes antes de listar	quero apagar uma despesa do mês passado	idle — Nenhuma conversa em andamento.	Pergunta mais detalhes sobre a despesa para filtrar os resultados	delete_flow
delete	Data exata — deve encontrar fixture e pedir confirmação	quero apagar a despesa de hoje	idle — Nenhuma conversa em andamento.	Mostra Despesa Eval Delete com valor 99,90 e pede confirmação para cancelar	delete_flow
delete	Confirmação — deve chamar delete_expense e encerrar	sim	delete_flow — Candidatos salvos. Aguardando: confirmação.	Confirma que a despesa Despesa Eval Delete foi cancelada com valor 99,90	idle
delete	Escolha de número na lista — deve recuperar do Redis e deletar	1	delete_flow — Múltiplos candidatos. Aguardando: escolha.	Confirma o cancelamento de Despesa Eval Delete	idle
delete	Nenhuma despesa encontrada — deve informar e encerrar	quero apagar a despesa do dia 01/01/2000	idle — Nenhuma conversa em andamento.	Informa que não encontrou nenhuma despesa nessa data e se coloca à disposição	idle