Clara - Desenvolvimento de uma assistente financeira pessoal no WhatsApp

Willian Robson de Gois[1], Dr. Rafael Vieira Coelho[1]

[1]Instituto Federal do Rio Grande do Sul - Campus Farroupilha
95174-274 – Farroupilha – RS – Brasil

williangoix@gmail.com, rafael.coelho@farroupilha.ifrs.edu.br

Abstract. *This article presents the development and validation of Clara, a chatbot-based personal financial assistant integrated with WhatsApp. The system architecture, built with Node.js/TypeScript, utilizes the OpenAI API for natural language, BullMQ/Redis for asynchronous tasks, Trigger.dev for scheduling, and Prisma for data management in MySQL. The project explores the potential of WhatsApp as a low-friction platform to democratize access to financial management tools in the Brazilian context, aiming to assist users in controlling expenses, income, and reminders. Real-world testing with 20 users over 45 days revealed Clara's technical feasibility, usability, and potential for improvement. The results indicate interest in the convenience of managing finances via WhatsApp, while also highlighting challenges related to using an unofficial WhatsApp library, the need for greater functional depth, and the refinement of the user experience on a social platform.*

Resumo. *Este artigo apresenta o desenvolvimento e validação de Clara, uma assistente financeira pessoal baseada em chatbot integrada ao WhatsApp. A arquitetura do sistema, construída com Node.js/TypeScript, utiliza a API da OpenAI para linguagem natural, BullMQ/Redis para tarefas assíncronas, Trigger.dev para agendamento e Prisma para gestão de dados em MySQL. O projeto explora o potencial do WhatsApp como plataforma de baixo atrito para democratizar o acesso a ferramentas de gestão financeira no contexto brasileiro, visando auxiliar usuários no controle de despesas, receitas e lembretes. Testes em ambiente real com 20 usuários durante 45 dias revelaram a viabilidade técnica, a usabilidade e o potencial de aprimoramento da Clara. Os resultados indicam interesse na conveniência de gerenciar finanças via WhatsApp. Ao mesmo tempo, apontam desafios como o uso de uma biblioteca não oficial, a necessidade de maior profundidade funcional e o refinamento da experiência do usuário em uma plataforma social.*

1. Introdução

O Brasil apresenta um cenário paradoxal: enquanto a digitalização avança rapidamente, com 86,2% da população conectada à internet e o WhatsApp presente em 98% dos smartphones [Kemp 2025], persistem desafios socioeconômicos profundos. Um dos mais críticos é a saúde financeira da população, marcada por altos índices de endividamento, atingindo 76,6 milhões de brasileiros em abril de 2025, segundo dados do levantamento Mapa da Inadimplência e Negociação de Dívidas no Brasil realizado mensalmente pelo Serasa. Pesquisas indicam que 35% dos inadimplentes não gerenciam suas finanças, muitos por confiarem apenas na memória (21%); por crença de que não ajudaria (18%); por falta de disciplina (15%) e por desconhecimento de como começar (13%) [CNDL 2024].

Essa lacuna entre a alta familiaridade com ferramentas de comunicação digital e a dificuldade com a gestão financeira sugere que a barreira não é apenas a falta de conhecimento, mas também o atrito e a complexidade das soluções existentes, como planilhas ou aplicativos dedicados. O WhatsApp, por sua onipresença e inserção no cotidiano, surge como uma plataforma estratégica de baixo atrito, operando em um ambiente onde o usuário já reside digitalmente e eliminando barreiras como *download* e aprendizado de novas interfaces. Aliado ao crescimento exponencial de chatbots e da inteligência artificial (IA), cujo mercado global deve alcançar USD 27,29 bilhões até 2030 [Grand View Research 2024], cria-se um espaço para inovação.

É neste contexto que se insere o projeto Clara, uma assistente financeira pessoal em formato de *chatbot* acessível por meio do WhatsApp, projetada para auxiliar usuários brasileiros na gestão de suas finanças. A proposta visa superar a barreira de usabilidade ao oferecer funcionalidades como registro e categorização de despesas e receitas, visualização de dados e lembretes personalizados dentro de uma interface conversacional, acessível e familiar, potencializado pelo uso de inteligência artificial. A viabilidade deste modelo é validada pelo surgimento de soluções como a *fintech* brasileira Magie [Campos 2024], também integrada de maneira conversacional através do WhatsApp, embora com um foco distinto na centralização de contas bancárias. A Clara, por sua vez, posiciona-se como uma ferramenta de registro e acompanhamento financeiro.

Dessa forma, o objetivo principal deste trabalho é desenvolver e validar a Clara, avaliando sua viabilidade técnica, aceitação pelo usuário e experiência de uso em um ambiente de produção simulado. Ao fazer isso, o projeto conecta a pesquisa teórica sobre arquiteturas de software modernas à sua aplicação prática, ao mesmo tempo que investiga a viabilidade da solução como um potencial produto comercial a partir de métricas técnicas e feedback qualitativo. Em última análise, o trabalho explora o potencial do WhatsApp como um canal de baixo atrito para democratizar o acesso a ferramentas de controle financeiro, servindo como uma prova de conceito para mitigar barreiras de usabilidade e conhecimento por meio de uma interface conversacional acessível.

2. Fundamentação Teórica

A arquitetura e as escolhas tecnológicas do projeto Clara foram baseadas em conceitos e ferramentas estabelecidas no desenvolvimento de software moderno, cada qual selecionada para atender a requisitos específicos do sistema. Esta seção aprofunda a justificativa teórica por trás das principais decisões.

2.1. Desenvolvimento de Bots no WhatsApp: O cenário de APIs e bibliotecas

A escolha do WhatsApp como plataforma central do projeto exige a análise de duas abordagens distintas para a integração programática: o uso da API (*Application Programming Interface* — Interface de Programação de Aplicações) oficial da Meta ou a adoção de bibliotecas não oficiais desenvolvidas pela comunidade. A decisão entre essas alternativas representa um balanço fundamental entre estabilidade, custo, legitimidade e flexibilidade.

A primeira abordagem, e mais robusta, é a utilização da Plataforma WhatsApp Business, com foco na sua implementação em nuvem, a Cloud API. Esta é a solução oferecida oficialmente pela Meta, hospedada em sua própria infraestrutura e acessível por meio da Graph API [META 2025]. Seus principais atributos são a alta escalabilidade e confiabilidade, com metas de disponibilidade de 99.9%, além de uma segurança robusta, aderente a padrões internacionais de proteção de dados como GDPR e LGPD. A API oficial oferece amplas funcionalidades, incluindo mensagens interativas e templates pré-aprovados, ao mesmo tempo que simplifica a operação ao eliminar a

necessidade de gerenciamento de servidores pelo desenvolvedor. O modelo de custo é baseado em conversas (divididas em categorias como marketing, utilidade, serviço, autenticação) [WHATSAPP 2025].

Em contrapartida, a segunda abordagem envolve o uso de bibliotecas não oficiais, como a Baileys. Tais bibliotecas são tipicamente projetos de código aberto que operam por meio de engenharia reversa do cliente WhatsApp Web, utilizando tecnologias como WebSockets ou a simulação de um navegador para replicar o comportamento de um usuário real. Apesar de oferecerem maior flexibilidade inicial e ausência de custos diretos, seu uso acarreta riscos significativos e intransponíveis para uma operação comercial. Elas operam em uma zona de não conformidade, violando os Termos de Serviço do WhatsApp, o que resulta em um risco significativo e permanente de suspensão ou banimento da conta [WHATSAPP 2025]. Adicionalmente, sua estabilidade pode ser precária, pois qualquer atualização na plataforma WhatsApp Web pode inutilizá-las abruptamente, dependendo da agilidade da comunidade para prover correções.

2.2. Estratégia de Monorepo

A adoção de uma arquitetura monorepo, gerenciada pelo Turborepo, foi uma decisão arquitetural chave para o projeto Clara. Um monorepo centraliza o código de múltiplos pacotes ou projetos (no caso, *apps/bot*, *apps/bullboard*, *packages/database*, etc.) em um único repositório de controle de versão. Isso oferece vantagens como gerenciamento simplificado de dependências, capacidade de realizar commits atômicos que afetam múltiplos pacotes, facilitação de refatorações e compartilhamento de código e *pipelines* de CI/CD (*Continuous Integration* / *Continuous Delivery*) mais coesos [MONOREPO.TOOLS 2025].

O Turborepo se destaca por seu foco em performance de build. Ele utiliza técnicas como *builds* incrementais (recompilando apenas o que mudou), *caching* inteligente (local e remoto, compartilhável entre a equipe) e execução paralela de tarefas para acelerar significativamente os tempos de build e teste [Ihnatovich 2025]. Essa escolha prioriza a velocidade de desenvolvimento e da DX (*Developer Experience* — Experiência do Desenvolvedor*)*. A simplicidade de configuração e os ganhos de

performance em builds e testes foram considerados, oferecendo um equilíbrio pragmático entre funcionalidade e facilidade de uso para as necessidades do projeto.

2.3. Filas e processamento de tarefas assíncronas: BullMQ

Aplicações web modernas, especialmente aquelas que interagem com serviços externos como APIs de terceiros (OpenAI) ou realizam operações demoradas, necessitam de mecanismos eficientes para processar tarefas em *background*. Isso melhora a responsividade da aplicação principal (evitando bloqueios) e aumenta a confiabilidade (permitindo retentativas em caso de falha). Filas de mensagens são a solução padrão para esse desacoplamento.

O BullMQ foi escolhido como a biblioteca de enfileiramento para a Clara. Trata-se de uma solução robusta e otimizada para o ambiente Node.js, utilizando o Redis como *backend* para persistência e gerenciamento das filas. O BullMQ oferece um conjunto rico de funcionalidades cruciais para o projeto: definição de prioridades para *jobs*, agendamento de *jobs* para execução futura (*jobs* atrasados), limitação de taxa (*rate limiting*) para controlar o consumo de APIs externas, retentativas automáticas configuráveis para *jobs* que falham, acompanhamento do progresso dos *jobs* e capacidade de escalar horizontalmente adicionando mais *workers* [BULLMQ.IO 2025].

Message brokers são sistemas responsáveis por intermediar a comunicação assíncrona entre diferentes partes de uma aplicação, frequentemente através de filas de mensagens ou tópicos de publicação/assinatura [IBM 2025]. Comparado a *message brokers* mais genéricos e complexos como Kafka ou RabbitMQ, o BullMQ apresenta vantagens significativas no contexto de uma aplicação Node.js. Sua configuração é geralmente mais simples, ele se integra nativamente com o Redis (que já era previsto para uso de cache em outros pontos do projeto) e seu conjunto de funcionalidades é especificamente desenhado para o caso de uso de filas de tarefas/*jobs*, em vez de *streaming* de eventos em larga escala [DRAGONFLY 2025].

2.4. Orquestração de jobs em *background*: O papel do Trigger.dev

O Trigger.dev foi incorporado à pilha de tecnologia da Clara para lidar com a necessidade de orquestração de fluxos de tarefas agendadas e duráveis. Trigger.dev é um *framework* de código aberto projetado para criar, agendar e monitorar *jobs* e

workflows em *background* de forma durável, diretamente no código da aplicação. Suas características relevantes incluem a definição de tarefas como código TypeScript assíncrono, agendamento robusto (similar a *cron*, mas com maior durabilidade e sem tempo de expiração inerente), suporte a tarefas de longa duração através de um sistema de *Checkpoint-Resume* (que permite pausar e retomar a execução de forma eficiente), observabilidade integrada em seu dashboard, retentativas automáticas com suporte a idempotência e integrações com outras ferramentas do ecossistema, como o Prisma, também utilizado no projeto [TRIGGER.DEV 2025].

2.5. Garantindo integridade e acesso aos dados: Zod e Prisma ORM

A integridade dos dados e a segurança no acesso a eles são fundamentais em qualquer aplicação, especialmente em sistemas com interações financeiras, como o projeto Clara. Para garantir esses aspectos dentro do ecossistema TypeScript, o projeto utiliza duas ferramentas centrais: Zod e Prisma.

Zod é uma biblioteca voltada para a declaração e validação de *schemas* (estrutura que define a forma, os tipos e as regras de validação de dados) (Figura 1) com forte integração ao TypeScript. Seu grande diferencial está na capacidade de inferir tipos estáticos diretamente a partir dos *schemas* definidos, o que permite que um único *schema* sirva tanto para validação em tempo de execução quanto para garantir segurança de tipos em tempo de compilação. Na Clara, essa abordagem é aplicada para validar variáveis de ambiente, *payloads* de comandos e mensagens, além de outros dados que circulam no sistema, prevenindo falhas decorrentes de dados malformados ou inesperados [ZOD 2025].

```
import { z } from 'zod'

const userSchema = z.object({
  name: z.string(),
  age: z.number().int().positive(),
})
```

Figura 1: Exemplo de um *schema* Zod.

Complementarmente, o Prisma é adotado como ORM (*Object-Relational Mapping*) pela sua ênfase em *type-safety* e produtividade do desenvolvedor. O modelo de dados é descrito de forma declarativa via arquivo schema.prisma, a partir do qual é gerado automaticamente o Prisma Client. Esse *client* oferece uma API de acesso ao banco de dados que respeita o schema definido, permitindo que todas as queries sejam verificadas em tempo de compilação. Com isso, elimina-se uma série de erros típicos de tempo de execução, como acessos a colunas inexistentes ou incompatibilidades de tipo [PRISMA.IO 2025].

2.6. Estratégia de conteinerização: Docker e Compose

A conteinerização com Docker é uma prática padrão no desenvolvimento de software moderno, e foi adotada no projeto Clara para obter diversos benefícios [DOCKER 2025]:

- **Consistência de ambiente:** Docker garante que o ambiente de execução da aplicação seja idêntico em desenvolvimento e produção, eliminando problemas de inconsistência da aplicação em diferentes ambientes.
- **Gerenciamento de dependências simplificado:** Todas as dependências (*runtime* Node.js, bibliotecas do sistema, etc.) são empacotadas dentro da imagem Docker, simplificando a configuração em diferentes máquinas.
- **Isolamento:** Cada serviço (bot, banco de dados, Redis) roda em seu próprio contêiner isolado, prevenindo conflitos.

- **Portabilidade e implantação:** Imagens Docker são portáveis e facilitam o processo de implantação em qualquer ambiente que suporte Docker, como a máquina virtual da Oracle Cloud planejada para produção.

O Docker Compose é utilizado para orquestrar múltiplos contêineres que compõem a aplicação. Os arquivos docker-compose.base.yml, docker-compose.dev.yml e docker-compose.prod.yml definem os serviços necessários (bot, bullboard, mysql, redis), suas configurações, volumes de dados, redes e dependências, permitindo iniciar e parar todo o ambiente da aplicação com comandos simples. O uso de arquivos separados para desenvolvimento e produção permite customizar configurações específicas para cada ambiente (ex: arquivo de variável de ambiente, volumes para código em dev, configurações otimizadas em prod) [DOCKER COMPOSE 2025].

3. Metodologia

A realização do projeto Clara seguiu uma metodologia estruturada, combinando um conjunto específico de tecnologias com um processo iterativo de desenvolvimento e uma validação multifacetada em ambiente real. O objetivo foi garantir não apenas a funcionalidade técnica, mas também a robustez, a usabilidade e a viabilidade da solução.

3.1. Materiais: A escolha tecnológica

A construção da Clara e as escolhas de ferramentas e tecnologias foram feitas visando criar um ecossistema coeso, moderno e eficiente, predominantemente centrado na linguagem de programação **TypeScript** e no ambiente de execução **Node.js**. A escolha foi também determinada pela familiaridade prévia do autor com esse conjunto de tecnologias, o que favoreceu maior produtividade e domínio na construção da solução. Desse modo, a base da aplicação foi desenvolvida em **Node.js** com **TypeScript**, uma escolha que combina o ecossistema robusto do **Node.js** com a segurança de tipos e a clareza de código proporcionadas pelo **TypeScript**, reduzindo erros em tempo de execução e facilitando a manutenção do projeto [NODE.JS 2025; TYPESCRIPT 2025]. O desenvolvimento foi conduzido no **Visual Studio Code** [VSCODE 2025], e a consistência entre os ambientes de desenvolvimento e produção foi assegurada pela

adoção de **Docker** e **Docker Compose**, containerizando a aplicação e suas dependências [DOCKER 2025; DOCKER COMPOSE 2025].

Para a persistência de dados, optou-se pela simplicidade e confiabilidade do **MySQL** como sistema de banco de dados relacional [MySQL 2025]. A interação com o banco foi abstraída pelo **Prisma ORM**, uma ferramenta moderna escolhida por sua forte segurança de tipos (*type safety*), seu cliente de banco de dados autogerado e seu sistema de migração declarativo, o que garantiu a integridade e a consistência do acesso aos dados [PRISMA.IO 2025]. Complementarmente, a biblioteca **Zod** foi utilizada para a validação de *schemas* em diversas camadas da aplicação, desde variáveis de ambiente até *payloads* de API, assegurando que apenas dados corretos transitassem pelo sistema [ZOD 2025].

A arquitetura foi projetada para ser responsiva e resiliente, tratando operações demoradas de forma assíncrona. Para isso, o **Redis** foi empregado como um armazenamento rápido em memória [REDIS 2025], servindo não apenas como cache para sessões do WhatsApp, mas também como a base para o **BullMQ**, uma robusta biblioteca de filas [BULLMQ.IO 2025]. Essa combinação permitiu o desacoplamento de tarefas, como o processamento e envio de mensagens, e a implementação de um sistema de *rate limiting* com a estratégia de *token bucket*, protegendo a aplicação contra abusos. A estrutura do código foi organizada em um monorepo gerenciado pelo **Turborepo** e **pnpm**, otimizando os processos de *build*, facilitando o compartilhamento de código e agilizando o ciclo de desenvolvimento [TURBOREPO 2025; PNPM 2025].

Houve grande integração com serviços externos. A inteligência conversacional foi viabilizada pela API da **OpenAI** [OpenAI 2025], com os custos e o desempenho das chamadas ao LLM (*Large Language Models* — Grande Modelo de Linguagem) monitorados pela plataforma de observabilidade **Helicone** [HELICONE 2025]. Para a funcionalidade de lembretes, o Trigger.dev foi incorporado para agendar e orquestrar tarefas em *background* de forma durável e confiável [TRIGGER.DEV 2025]. A decisão mais crítica, contudo, foi a integração com o WhatsApp. Embora a API Oficial da Meta represente a solução ideal em termos de estabilidade e segurança, sua utilização apresentou barreiras burocráticas e de custo, como a necessidade de um CNPJ ativo e verificação empresarial, inviáveis para o escopo acadêmico do projeto. Diante disso,

optou-se pragmaticamente pela biblioteca não oficial **Baileys**, que permitiu a prototipagem rápida e a validação do conceito com usuários reais, mesmo ciente dos riscos de instabilidade e da necessidade de uma futura migração para a solução oficial. Dentre as opções de bibliotecas não oficiais, esta foi escolhida por usar abordagem baseada em *websockets* e engenharia reversa do protocolo de comunicação do WhatsApp Web, que é menos custosa em termos de memória RAM (*Random Access Memory*) em comparação com outras bibliotecas que seguem uma abordagem de emular um navegador sem interface gráfica (conhecida como *headless browser*), geralmente com Chromium, e manipular as informações do WhatsApp Web através do HTML/CSS das páginas [BAILEYS 2025].

Finalmente, para validar a solução em um cenário realista, um ambiente de produção foi provisionado em uma Máquina Virtual na **Oracle Cloud**, com um número de telefone brasileiro dedicado adquirido via **BR.did** [BR.DID 2025]. Essa infraestrutura permitiu a implantação de todos os serviços conteinerizados e a realização de testes em condições próximas às de um produto real. O conjunto completo dessas tecnologias está detalhado no Quadro 1.

Quadro 1: *Stack* tecnológica principal e justificativa.

Categoria	Ferramenta/ tecnologia	Papel no projeto	Justificativa/Benefíci o chave
Linguagem/*Runti me*	TypeScript / Node.js	Linguagem principal e ambiente de execução	Ecossistema maduro (Node.js), segurança de tipos, DX moderno, manutenibilidade (TypeScript)
Monorepo	Turborepo / pnpm	Gerenciamento da base de código multi-pacote	Builds rápidos, *caching* eficiente, compartilhamento de código, DX simplificado

Categoria	Ferramenta/ tecnologia	Papel no projeto	Justificativa/Benefício chave
Banco de dados / ORM	MySQL / Prisma	Persistência de dados relacionais e acesso ao banco	ORM moderno, type safety ponta-a-ponta, migrações robustas, DX superior
Filas / *Cache*	Redis / BullMQ	Processamento e envio das mensagens de forma assíncronas, cache em memória e limitador de taxa (*rate limiter*)	Filas robustas para Node.js, funcionalidades (prioridades, *delays*, *retries*), simplicidade vs Kafka/MQ
Agendamento	Trigger.dev	Agendamento e execução de tarefas em *background* (ex: lembretes)	Tarefas duráveis, observabilidade, DX para *jobs* complexos/agendados
Validação de dados	Zod	Validação de schemas e garantia de integridade de dados	Integração com TypeScript, inferência de tipos, validação *runtime*, DX
Conteinerização	Docker / Docker Compose	Empacotamento e orquestração de serviços	Consistência de ambiente (dev/prod), isolamento, portabilidade, implantação simplificada
Observabilidade	Sentry / Helicone / Telegram	Monitoramento de erros, performance, chamadas LLM, alertas	Detecção proativa de problemas, debugging facilitado, visibilidade operacional
Integração com WhatsApp	Baileys	Biblioteca não oficial para assumir o controle de um número do WhatsApp	Sem burocracias para iniciar como a API oficial; Uso de websockets em vez de *headless browsers*

Categoria	Ferramenta/ tecnologia	Papel no projeto	Justificativa/Benefício chave
			como outras bibliotecas
Modelo de IA	OpenAI	Processamento de linguagem natural, geração de respostas	Interpretação de mensagens e conversação
Hospedagem em produção	Oracle Cloud	Infraestrutura para implantação em produção	Ambiente de nuvem para execução da VM/contêineres

A combinação dessas ferramentas foi elaborada prezando por um ecossistema moderno e integrado, priorizando a segurança de tipos, a DX e o desempenho no desenvolvimento de aplicações Node.js, principalmente executando uma instância de um número de WhatsApp.

3.2. Métodos: Processo de desenvolvimento e validação

A construção e a avaliação da Clara seguiram etapas bem definidas, desde a concepção até a análise de *feedback* dos usuários. A metodologia adotada buscou uma orientação clara para a construção de um sistema robusto e pronto para ambiente produtivo, incorporando práticas de observabilidade, segurança, implantação containerizada e testes com usuários finais.

3.2.1. Desenvolvimento e implantação em produção

O desenvolvimento da clara seguiu um ciclo incremental. Inicialmente, realizou-se o levantamento de requisitos, com foco na definição das funcionalidades essenciais do chatbot, suas limitações e os requisitos não funcionais, como observabilidade, desempenho e segurança, alinhados aos objetivos do projeto. Em seguida, passou-se ao desenho da arquitetura, que envolveu o planejamento da estrutura geral do sistema, incluindo a organização do monorepo (separação entre *apps* e *packages*) e o fluxo de dados entre os componentes principais, como *bot*, filas, banco de dados, IA e agendador.

Com a arquitetura definida, a construção da aplicação procedeu de forma iterativa. As funcionalidades foram desenvolvidas e integradas progressivamente, partindo da conexão inicial com o WhatsApp, passando pela incorporação da API da OpenAI e pelo desenvolvimento dos *workers* responsáveis pelo processamento assíncrono das filas.

Para validar a robustez e o desempenho da solução em um cenário de uso real, a etapa seguinte consistiu na implantação em um ambiente de produção simulado. Este ambiente foi provisionado em uma Máquina Virtual (VM) na Oracle Cloud, uma escolha que permitiu testar a aplicação em condições realistas. A configuração da infraestrutura envolveu a instalação do Docker e do Docker Compose, o provisionamento de volumes de dados persistentes para o MySQL e o Redis — garantindo a integridade dos dados entre reinicializações dos contêineres — e a configuração de rede para expor o serviço do *bot* à internet.

A implantação foi automatizada por meio de um arquivo docker-compose.prod.yml, que orquestrou a construção das imagens e a inicialização de todos os serviços.

Pensando no realismo dos testes, foi adquirido um número de telefone dedicado (via BR.did), possibilitando que usuários reais interagissem com o bot em seu ambiente de produção. Essa abordagem possibilitou uma avaliação precisa do desempenho da aplicação sob carga, do consumo de recursos na VM e da identificação de problemas de configuração ou interação entre serviços que não seriam evidentes em um ambiente de desenvolvimento local.

3.2.2. Testes de usabilidade e coleta de feedback

A validação funcional e da experiência do usuário (UX) foi conduzida com um grupo de 20 usuários reais, composto por colegas, amigos e conhecidos do autor. Durante um período de 45 dias, os participantes foram convidados a interagir livremente com o *chatbot*, utilizando suas funcionalidades no dia a dia.

Ao final do período, foi aplicado um formulário de pesquisa via WhatsApp para os usuários que interagiram com a Clara, contendo dez perguntas qualitativas e quantitativas. As perguntas foram:

- Quão fácil foi usar a Clara pelo WhatsApp? (de 1 a 5)

- O quão útil foi a Clara para te ajudar com suas dúvidas ou tarefas financeiras? (de 1 a 5)
- A Clara respondeu suas perguntas de forma clara e compreensível? (de 1 a 5)
- Você sentiu que podia confiar nas informações fornecidas? (de 1 a 5)
- Qual a probabilidade de você usar um assistente como a Clara no seu dia a dia? (de 1 a 5)
- Você usa ou já usou alguma forma de controle financeiro? (assistentes, aplicativos, planilha, etc). Se sim, quais? (descritiva)
- Houve alguma funcionalidade que você esperava e não encontrou? Qual? (descritiva)
- Você pagaria por um assistente financeiro como a Clara? (sim, não ou talvez)
- Se sim, até quanto? (opções de valores)
- Alguma sugestão de melhoria, novo recurso ou feedback geral? (descritiva)

O objetivo foi coletar *feedbacks* sobre a facilidade de uso, utilidade e confiança na solução, além de identificar bugs, limitações e sugestões de melhorias, bem como avaliar o interesse na proposta como um produto comercial. Foram analisados diferentes aspectos, como a satisfação geral dos usuários com a experiência proporcionada, os relatos de erros ou comportamentos inesperados encontrados durante o uso, e as sugestões de novas funcionalidades, melhorias na interface conversacional ou ajustes no fluxo de interação. Também foram observadas percepções sobre a clareza das informações financeiras apresentadas e o nível de confiança no bot para executar tarefas financeiras. Por fim, considerou-se o grau de interesse dos participantes em continuar utilizando a solução como um produto viável no mercado.

3.2.3. Monitoramento técnico contínuo

Paralelamente aos testes com usuários, foi implementada uma estratégia de monitoramento contínuo para garantir a observabilidade do sistema. Ferramentas como Sentry foram utilizadas para rastreamento de erros e análise de performance (Figura 4); Helicone para monitorar custos, latência e sucesso das chamadas à API da OpenAI (Figura 3); e um *bot* de alertas no Telegram para notificar imediatamente sobre eventos críticos, como a queda e reinício do serviço (Figura 2). Os dados coletados por essas

ferramentas foram analisados continuamente durante a fase testes, e inclusive alimentando o processo de desenvolvimento com informações para correção de *bugs*, otimização de performance e ajuste fino das integrações.

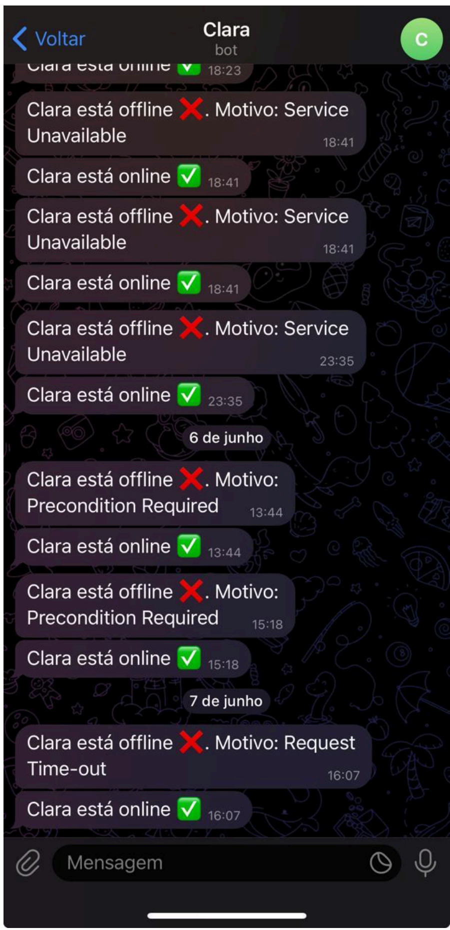


Figura 2: Alertas enviados pelo *bot* do Telegram.

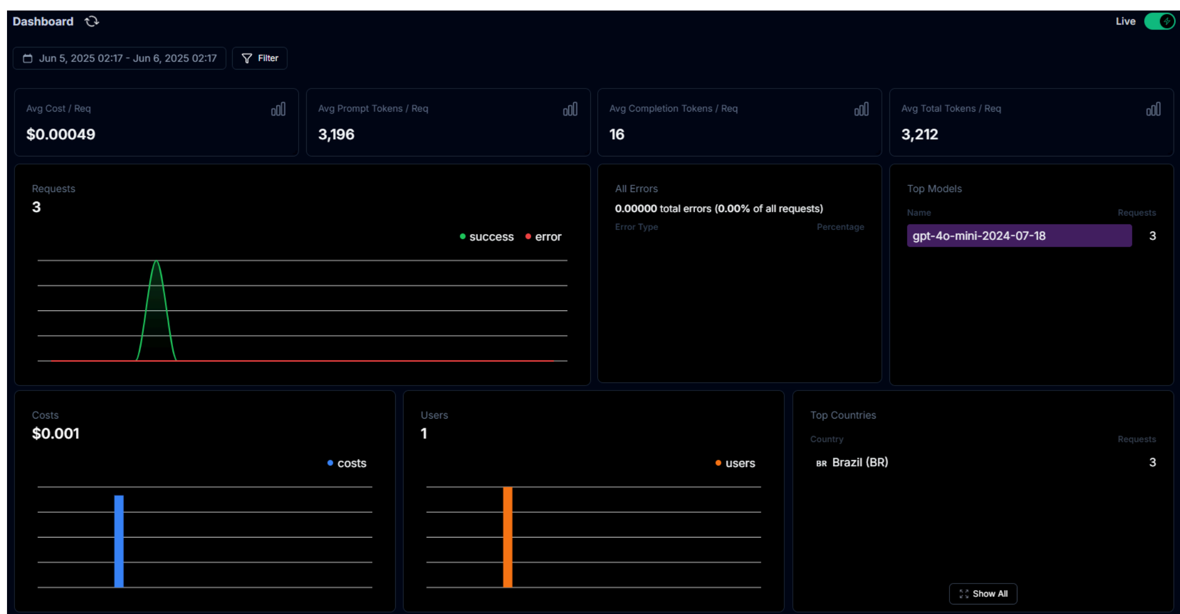


Figura 3: *Dashboard* de métricas do Helicone.

Figura 4: Registro de uma requisição para OpenAI mapeada no Helicone.

3.3. Modelagem de banco, *migrations* e LGPD

A persistência dos dados da aplicação é gerenciada por um banco de dados MySQL, com o *schema* definido declarativamente no arquivo **schema.prisma**. Este arquivo descreve as tabelas (*models*), colunas e relacionamentos na sintaxe própria do Prisma. As entidades incluem:

- **User:** Para armazenar informações sobre os usuários que em algum momento interagiram com a Clara. Contém índice na coluna *phone* por ser o campo de referência para buscas em troca de mensagens (Figura 5). Para atender aos princípios de privacidade e conformidade com a LGPD (Lei Geral de Proteção de Dados Pessoais), foram adotadas medidas técnicas específicas, como a criptografia simétrica (AES-256-CBC) no campo *phone*, garantindo a confidencialidade de um dado pessoal sensível e diretamente identificável. Para possibilitar buscas rápidas e seguras sem expor o dado original, utiliza-se também o campo *phoneHash*, que armazena o *hash* SHA-256 do número,

marcado como *@unique* para garantir unicidade e usado nos filtros de consulta. Essa configuração equilibra segurança e performance:

- ○ O campo phone pode ser descriptografado quando necessário (ex: exibição ou envio de mensagens);
- ○ O campo *phoneHash* permite comparações eficientes e seguras sem necessidade de descriptografia.

```
model User {
  id        String   @id @default(cuid())
  phone     String // AES encrypted
  phoneHash String   @unique
  name      String?  @db.Text
  balance   Float    @default(0)
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")

  transactions Transaction[]
  goals        Goal[]
  reminders    Reminder[]

  @@index([phone])
  @@map("users")
}
```

Figura 5: Definição do modelo *User* no *schema.prisma*.

- **Transaction:** Para registar as transações de receitas e despesas relatadas pelos usuários (Figura 6).

```
enum TransactionType {
  INCOME
  EXPENSE
}

model Transaction {
  id          String          @id @default(cuid())
  userId      String          @map("user_id")
  type        TransactionType
  amount      Float
  category    String
  description String?
  date        DateTime        @default(now())
  createdAt   DateTime        @default(now()) @map("created_at")

  user User @relation(fields: [userId], references: [id])

  @@map("transactions")
}
```

Figura 6: Definição do modelo *Transaction* no *schema.prisma*.

- **Reminder:** Para armazenar informações sobre tarefas agendadas via Trigger.dev (Figura 7).

```
model Reminder {
  id                String    @id @default(cuid())
  userId            String    @map("user_id")
  lastTriggerRunId String?   @map("last_trigger_run_id")
  message           String
  nextReminderAt    DateTime @map("next_reminder_at")
  maxOcurrences     Int?      @map("max_ocurrences")
  intervalDays      Int?      @map("interval_days")
  sentCount         Int       @default(0) @map("sent_count")
  createdAt         DateTime @default(now()) @map("created_at")

  user User @relation(fields: [userId], references: [id])

  @@map("reminders")
}
```

Figura 7: Definição do modelo *Reminder* no *schema.prisma*.

Com base neste *schema*, o comando *prisma generate* cria o PrismaClient type-safe. Este cliente, importado e utilizado principalmente em apps/bot, fornece métodos para todas as operações CRUD (*Create*, *Read*, *Update*, *Delete*) com garantia de tipos em tempo de compilação.

A evolução do *schema* é gerenciada pelo *Prisma Migrate*, que gera e aplica migrações SQL (*Structured Query Language*) de forma versionada e consistente, garantindo que a estrutura do banco de dados seja a mesma em todos os ambientes.

4. Implementação

Esta seção detalha a concretização da arquitetura e dos conceitos teóricos na implementação específica do *chatbot* Clara, descrevendo a estrutura do código, os fluxos de dados e a integração dos componentes tecnológicos.

4.1. Visão geral da estrutura

Conforme delineado na metodologia e fundamentação, a arquitetura de software da Clara foi concebida sob o paradigma de um monorepo, gerenciado pela ferramenta Turborepo, visando otimizar o ciclo de desenvolvimento, simplificar o gerenciamento de dependências e promover o reuso de código.

A estrutura organiza-se logicamente em dois diretórios centrais: ***apps***, que contém as aplicações executáveis, e ***packages***, que abriga os módulos de lógica e configuração compartilhados. Essa separação de responsabilidades é fundamental para a manutenção e escalabilidade do projeto.

O diretório ***apps*** concentra os componentes principais e executáveis do sistema. Nele reside o *apps/bot*, o serviço Node.js que constitui o núcleo operacional da Clara. Este componente é responsável por toda a lógica de negócio, desde a interação com o WhatsApp e o processamento de comandos envolvendo as filas do BullMQ até a orquestração com serviços da OpenAI e a comunicação com o banco de dados. Complementando-o, o *apps/bullboard* fornece uma interface visual, a *Bull Dashboard*, que se mostrou essencial para o monitoramento e a depuração das filas de tarefas assíncronas, garantindo a observabilidade operacional do sistema ao permitir a visualização de *jobs* em diferentes estados.

Em contrapartida, o diretório ***packages*** foi projetado para centralizar a lógica e as configurações, evitando a duplicação de código e garantindo consistência em todo o projeto. O *packages/database*, por exemplo, encapsula toda a configuração do Prisma, incluindo o *schema* e as migrações, fornecendo um ponto único e seguro de acesso ao banco de dados. De forma semelhante, o *packages/env* utiliza a biblioteca Zod para gerenciar a validação e o carregamento de variáveis de ambiente, assegurando que as configurações sejam consistentes e seguras. Adicionalmente, pacotes como *eslint-config* e *typescript-config* impõem um padrão de codificação uniforme, reforçando a qualidade e a coesão do código-fonte.

Finalmente, a conteinerização com Docker e Docker Compose unifica o ambiente de desenvolvimento e produção, garantindo consistência e simplificando o processo de implantação. Os arquivos de configuração, como docker-compose.dev.yml e docker-compose.prod.yml, definem como os serviços bot, Bullboard, Redis e MySQL são construídos em imagens Docker e orquestrados para execução conjunta, gerenciando redes, volumes e variáveis de ambiente específicas para cada contexto. Essa abordagem encapsula a complexidade da infraestrutura e facilita a portabilidade da aplicação.

4.2. Operações assíncronas: Filas, *workers* e padrões de *design*

O sistema lida com operações assíncronas por meio do uso de filas, *workers* e padrões de *design* apropriados. Tarefas que não podem ou não devem ser executadas de forma síncrona, como chamadas a APIs externas ou processamento intensivo, são tratadas com o auxílio do BullMQ e do Redis.

A configuração das filas, localizada em *apps/bot/src/queues/*, define duas filas principais: ***incomingMessagesQueue*** e ***outgoingMessagesQueue***. A conexão com o Redis, por sua vez, é centralizada no módulo *libs/redis/*.

A implementação dos *workers*, encontrada em *apps/bot/src/workers/*, define os workers ***incomingMessagesWorker*** e ***outgoingMessagesWorker***, responsáveis por processar as mensagens presentes nas respectivas filas. Esses jobs são organizados em etapas utilizando o *Pipeline Pattern*, com sua lógica localizada em *apps/bot/src/pipelines/*.

Já o *Strategy Pattern*, implementado em *apps/bot/src/strategies/*, é empregado para tratar os diferentes tipos de mensagens recebidas — como texto, imagem e áudio —, permitindo aplicar o processamento mais adequado conforme o conteúdo.

4.3. Fluxo de mensagens

O fluxo principal de interação do *bot* começa no ponto de entrada *apps/bot/src/index.ts* (Figura 8). Este arquivo inicia a conexão com a API do WhatsApp (através da implementação da biblioteca Baileys em *libs/whatsapp/*) e configura os *listeners* para receber mensagens ou eventos da instância.

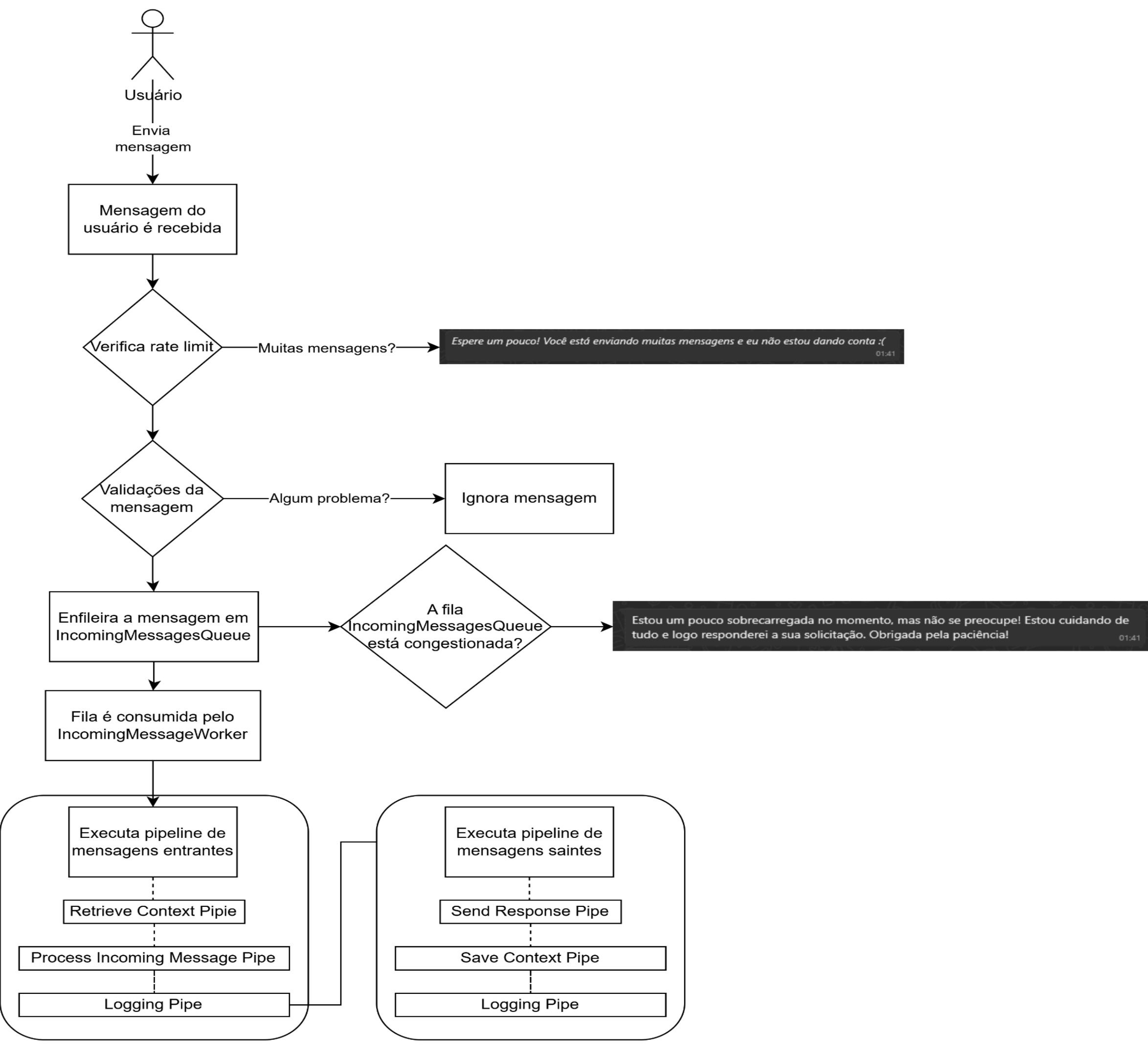


Figura 8: Fluxograma do processamento de mensagens.

4.3.1. Etapas do fluxo de mensagens

O processamento das mensagens foi implementado de forma assíncrona através de filas do BullMQ visando suportar picos de volume e controlar o volume de envio via instância do WhatsApp, mitigando assim o principal gargalo de desempenho da aplicação (Baileys) e garantido a ordem de envio das mensagens, contornando uma limitação do Baileys, que pode causar inversão na entrega quando múltiplas mensagens são enviadas em sequência rápida.

Quando uma nova mensagem do usuário chega ao sistema, ela é imediatamente submetida a um processo de validação em múltiplas etapas, que atua como uma primeira linha de defesa para a integridade e estabilidade da aplicação. A **primeira verificação** é um mecanismo de *rate limiting*, implementado com o algoritmo *Token Bucket* sobre o Redis (Figura 9). Essa estratégia associa a cada usuário um "balde" de tokens que é reabastecido a uma taxa constante, prevenindo sobrecarga por excesso de requisições. Se o usuário possuir *tokens*, um é consumido e a mensagem prossegue; caso contrário, a mensagem é descartada e o usuário é notificado da restrição. Superada essa primeira barreira, a mensagem passa por uma validação de conteúdo e estrutura através de *schemas* definidos com a biblioteca Zod, garantindo que apenas dados bem-formados e com os atributos necessários avancem para a próxima fase.

```
const { allowed, blockHits } = await rateLimiter(
  this.redisService,
  {
    key: `user:${userId}`,
    capacity: 5,
    refillRate: 1 / 5,
  }
);


if (!allowed) {
  if (blockHits === 1) {
    this.logger.warn(`Rate limit exceeded for user ${userId}`);
    await this.enqueueErrorMessage(
      userId,
      "_Espere um pouco! Você está enviando muitas mensagens e eu não estou dando conta :(_"
    );
  }
  return;
}
```

Figura 9: Trecho de código que verifica o *rate limit*.

Uma vez validada, a mensagem é enfileirada na fila *IncomingMessagesQueue*, aguardando para ser consumida pelo *IncomingMessageWorker*, que dará início à *pipeline* de processamento de fato, detalhada na seção seguinte. Paralelamente a esse fluxo, o sistema implementa um mecanismo de gestão de carga para manter a transparência com o usuário. Ele monitora continuamente o estado da *IncomingMessagesQueue* e, caso detecte um congestionamento — definido como mais de dez tarefas pendentes (Figura 10) —, uma mensagem de cortesia é proativamente enviada ao usuário, informando sobre um possível atraso na resposta. Para evitar notificações repetitivas, essa mensagem de aviso é enviada apenas uma vez por minuto, um controle também gerenciado por uma chave temporária no Redis. Esse conjunto de validações e controles assegura que apenas mensagens legítimas entrem na *pipeline* principal de processamento, ao mesmo tempo em que o sistema gerencia sua própria carga de forma transparente ao usuário, visando uma boa experiência do usuário.

```
export class IncomingMessagesQueue {

  ...

  public isCongested() {
    return this.queue.getJobCounts().then((counts) => {
      const threshold = 10;
      return (counts.waiting || 0) + (counts.delayed || 0) > threshold;
    });
  }
}
```

Figura 10: Função que verifica uma fila congestionada.

4.3.2. *Pipelines* de processamento

Para estruturar o fluxo de dados de maneira organizada e escalável, foi adotado o padrão de *design Pipeline*. Essa abordagem se mostrou vantajosa por promover uma clara separação de responsabilidades entre os fluxos de entrada e saída, facilitar a extensão com novas etapas de processamento com baixo acoplamento e garantir que cada componente (*pipe*) tenha uma responsabilidade única. Com base nesse padrão, a arquitetura foi organizada em duas pipelines distintas e sequenciais: uma para o tratamento de mensagens recebidas (entrantes) e outra para o envio de respostas (saintes).

A **pipeline de mensagens entrantes** é responsável por todo o ciclo de interpretação da solicitação do usuário. O processo inicia-se com o estágio *Retrieve Context Pipe*, que recupera do Redis o histórico recente da conversa, composto pelas dez últimas interações. Em seguida, a mensagem avança para a etapa central, o *Process Incoming Pipe*, onde a lógica de negócio é aplicada. Neste ponto, o padrão de *design Strategy* é utilizado para determinar dinamicamente o tratamento mais adequado conforme o tipo de conteúdo recebido:

- Para interações textuais, a ***TextStrategy*** (Figura 11) utiliza a mensagem do usuário para formular a requisição à API da OpenAI com o modelo **gpt-4o-mini**.

```
export class TextMessageStrategy implements MessageProcessingStrategy {

  name: string = "text";

  constructor(private openAIService: OpenAIService) {}

  async process(context: JobContext, user: User): Promise<JobResponse> {
    const response = await this.openAIService.generateResponse(context.message, user);
    return {
      content: response.text,
      messages: response.messages
    };
  }
}
```

Figura 11: Código da estratégia de resposta para mensagens do tipo texto.

- No caso de mensagens de áudio, a ***AudioStrategy*** (Figura 12) primeiro transcreve o áudio em texto por meio do modelo whisper-1 da OpenAI, subsequentemente, utiliza o resultado para gerar uma resposta em texto através de uma requisição para a API da OpenAI usando o modelo **gpt-4o-mini** [OPENAI 2025].

```
export class AudioMessageStrategy implements MessageProcessingStrategy {
  name: string = "audio";

  constructor(
    private openAIService: OpenAIService
  ) { }

  async process(context: JobContext, user: User): Promise<JobResponse> {
    const data = context.message.messageData;
    const mediaPath = (data.content as AudioContent).filePath;

    if (typeof mediaPath !== 'string' || mediaPath.trim() === '') {
      throw new Error("Media path is required for audio messages");
    }

    const transcription = await this.openAIService.generateTranscription(mediaPath);

    const response = await this.openAIService.generateResponse({
      ...context.message,
      messageData: {
        ...data,
        content: transcription,
        type: "text"
      }
    }, user);

    return {
      content: response.text,
      messages: response.messages
    };
  }
}
```

Figura 12: Código da estratégia de resposta para mensagens do tipo áudio.

- De forma análoga, a ***ImageStrategy*** (Figura 13) processa o conteúdo visual enviado pelo usuário. Primeiramente busca a imagem codificada usando o método *base64* no Redis e envia no conteúdo da requisição para a geração de texto da OpenAI (com modelo **gpt-4o-mini**).

```
export class ImageMessageStrategy implements MessageProcessingStrategy {
  name: string = "image";

  constructor(private openAIService: OpenAIService) {}

  async process(
    context: JobContext,
    user: User
  ): Promise<JobResponse> {
    const data = context.message.messageData;
    const { filePath } = data.content as ImageContent;

    if (typeof filePath !== 'string' || filePath.trim() === '') {
      throw new Error("Caminho da imagem é obrigatório para mensagens de imagem");
    }

    const { text, messages } = await this.openAIService.generateResponse(context.message,
user);

    return {
      content: text,
      messages,
    };
  }
}
```

Figura 13: Código da estratégia de resposta para mensagens do tipo imagem.

Uma vez que a resposta é gerada pela estratégia apropriada, a pipeline conclui seu fluxo com o estágio *Enqueue Message Pipe*, que enfileira a resposta na *OutgoingMessagesQueue*, pronta para ser enviada.

Por sua vez, a **pipeline de mensagens saintes** é ativada para finalizar a interação. Sua primeira etapa, o *Send Response Pipe*, é encarregada de enviar a resposta, já formatada, para o usuário por meio da integração com o WhatsApp. Imediatamente após o envio bem-sucedido, a etapa final, *Save Context Pipe*, atualiza o histórico da conversa no Redis, armazenando tanto a nova mensagem do usuário quanto a resposta do *bot*. Essa ação é fundamental para preservar o estado e garantir a continuidade contextual em futuras interações.

Essa estrutura de pipelines não só organiza o código de forma lógica e manutenível, mas também garante a robustez do processamento assíncrono, desacoplando cada etapa do processamento.

4.4. Camada de inteligência: Integração com OpenAI

A capacidade de compreensão de linguagem natural e a geração de respostas contextuais são providas pela integração com a API da OpenAI, orquestrada pelo OpenAIService, componente central localizado no módulo *libs/ai/services/*. A implementação utiliza a biblioteca Vercel AI SDK (@ai-sdk/openai), que facilita a comunicação com os modelos de linguagem e o gerenciamento de ferramentas.

4.4.1. Funcionalidades e capacidades

O serviço foi projetado para interpretar mensagens de usuários que não seguem um formato de comando estrito, permitindo uma interação conversacional fluida. Em vez de depender de comandos estritos, o sistema utiliza a funcionalidade de *Function Calling* da API da OpenAI, que permite ao modelo de linguagem invocar um conjunto de "ferramentas" (*tools*) predefinidas. Essas ferramentas atuam como uma ponte segura e controlada entre a IA e a lógica de negócio da aplicação, permitindo a manipulação e consulta de dados no banco de dados. Para organizar essas capacidades, as ferramentas foram estruturadas em três categorias funcionais: **gestão de transações**, **controle de saldo** e **agendamento de lembretes**.

O núcleo da funcionalidade da Clara reside na gestão de transações e análise financeira. A ferramenta **trackTransaction** permite que os usuários registrem despesas e receitas de maneira natural, como "gastei 50 reais no almoço", associando o valor ao perfil do usuário (Figura 14).

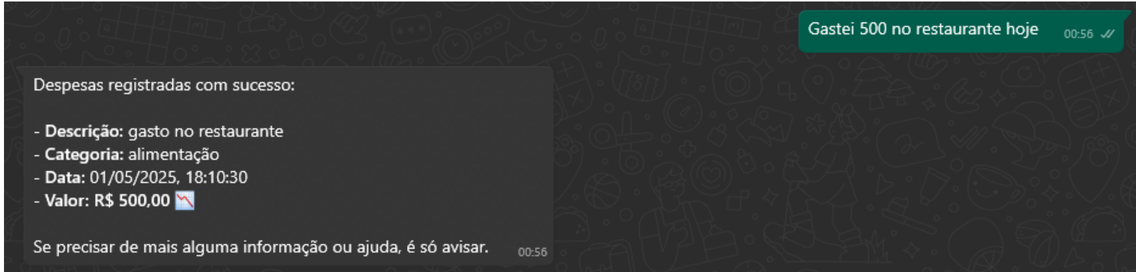


Figura 14: Exemplo de interação em que o *bot* utiliza a ferramenta *trackTransaction*.

Para consultas, a ferramenta **getTransactions** recupera o histórico de movimentações, possibilitando que o bot responda a perguntas como "quais foram meus últimos gastos?" (Figura 15).

Figura 15: Exemplo de interação em que o *bot* utiliza a ferramenta *getTransactions*.

Indo além da simples listagem, a ferramenta **createChart** oferece uma capacidade analítica visual, gerando gráficos de gastos que são enviados diretamente no chat e facilitam a identificação de padrões pelo usuário (Figura 16).

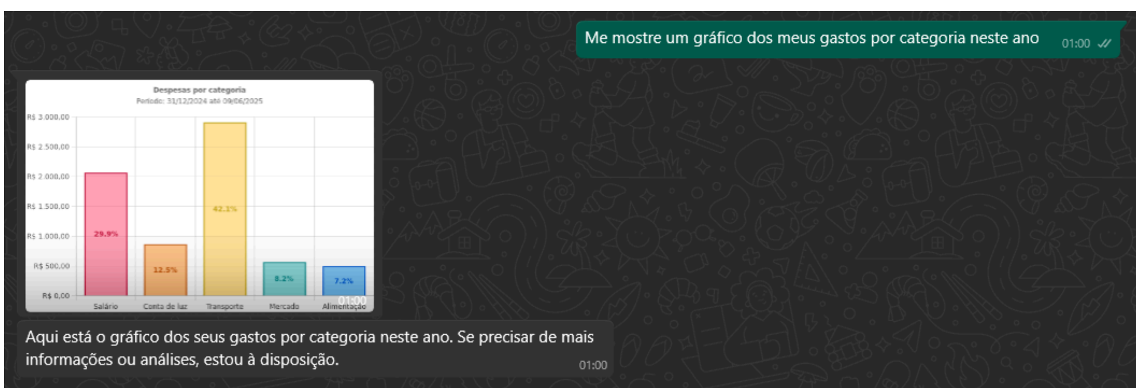


Figura 16: Exemplo de interação em que o *bot* utiliza a ferramenta *createChart*.

Paralelamente, o controle de saldo é gerenciado por um par de ferramentas complementares. A **getBalance** permite consultas rápidas sobre o saldo atual (Figura 17), enquanto a **setBalance** oferece ao usuário a flexibilidade de definir ou ajustar manualmente seu saldo inicial ou corrente (Figura 18).

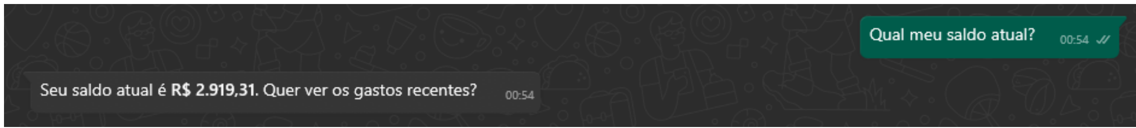


Figura 17: Exemplo de interação em que o *bot* utiliza a ferramenta *getBalance*.

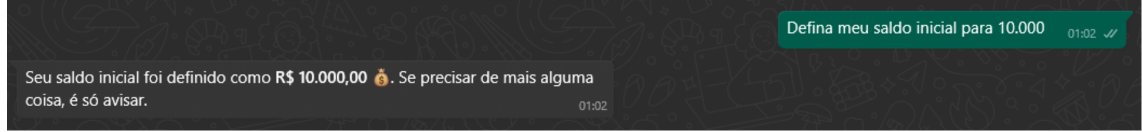


Figura 18: Exemplo de interação em que o *bot* utiliza a ferramenta *setBalance*.

Juntas, essas funcionalidades são cruciais para manter os registros do assistente alinhados com a realidade financeira do usuário, garantindo a precisão das informações.

Finalmente, a capacidade de gerenciamento proativo de tarefas é implementada através do sistema de lembretes. A ferramenta **createReminder** permite agendar notificações para datas futuras, como o pagamento de contas (Figuras 19 e 20).

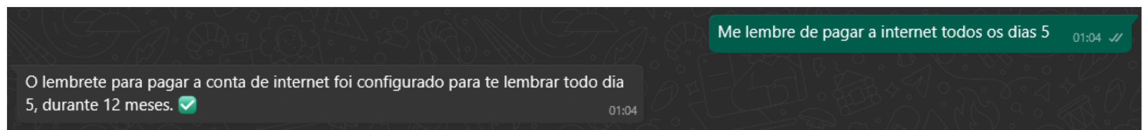


Figura 19: Exemplo de interação em que o *bot* utiliza a ferramenta *createReminder*.

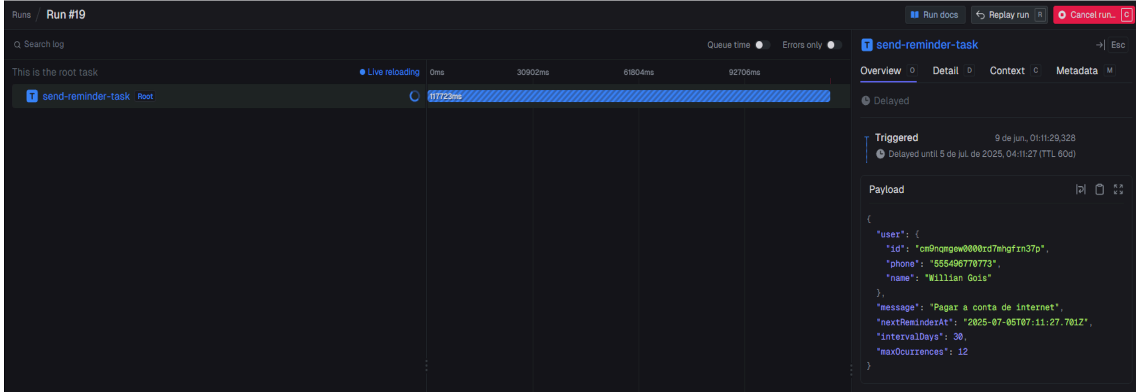


Figura 20: Tela de informações da tarefa (*task*) agendada no Trigger.dev.

O gerenciamento desses agendamentos é completado pelas ferramentas **getReminders**, para listar todos os lembretes ativos (Figura 21), e **cancelReminders**, que possibilita a remoção de lembretes específicos (Figura 22). Esse conjunto de ferramentas transforma a Clara de um simples registrador de transações em um assistente proativo, ajudando o usuário a se organizar financeiramente.

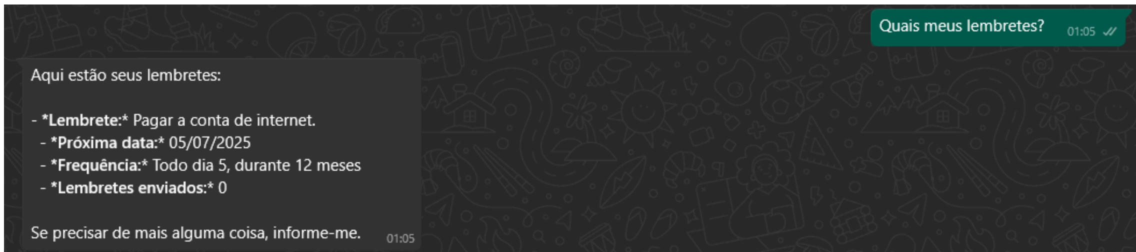


Figura 21: Exemplo de interação em que o *bot* utiliza a ferramenta *getReminders*.

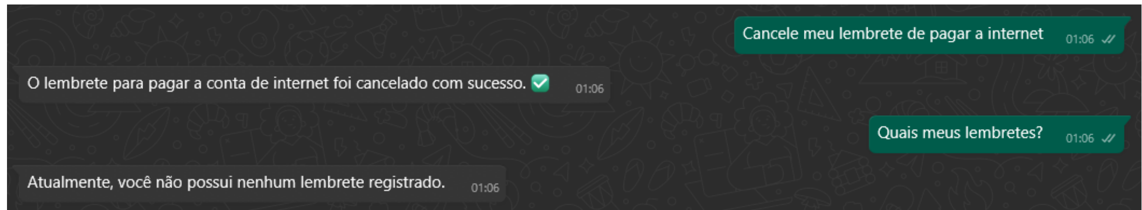


Figura 22: Exemplo de interação em que o *bot* utiliza a ferramenta *cancelReminders*.

4.4.2. Fluxo de integração

Conforme detalhado na seção 4.3.1 deste trabalho, o processamento de uma mensagem de usuário pela camada de inteligência artificial é realizado de forma assíncrona, orquestrado pelo *OpenAIService* a partir de um *job* consumido de uma *fila*. Este job encapsula não apenas a mensagem atual do usuário, mas também seu histórico de conversas e outros metadados relevantes, fornecendo o contexto necessário para uma interação coerente.

O primeiro e mais crucial passo deste fluxo é a construção de um *prompt* dinâmico, uma tarefa realizada pelo método *getPrompt()* (Figura 23). Este prompt é elaborado combinando um *prompt* base, que define a persona da assistente Clara, com dados específicos do usuário. O *prompt* base foi desenvolvido iterativamente seguindo princípios de engenharia de *prompt*, estabelecendo um tom cordial e profissional,

definindo restrições para manter o foco em finanças e especificando o formato de saída, incluindo o uso de emojis e a formatação de texto para o WhatsApp (*negrito*, _itálico_). Além disso, ele contém exemplos de interações, instruções para lidar com ambiguidades e diretrizes para oferecer sugestões proativas, visando uma experiência rica e intuitiva.

```
getPrompt(user: User): string {
  const buildContextualPrompt = () => {
    return `## Contexto
    ${user.name ? `- O nome do usuário é ${user.name}` : ''}
    - O primeiro acesso do usuário foi em ${createDateFormatted(user.createdAt)}
    - Hoje é dia ${createDateFormatted()}
    `;
  };


  return `${OpenAIService.basePrompt}\n\n${buildContextualPrompt()}`;
}
```

Figura 23: Função *getPrompt()* responsável por construir o *prompt*.

Com o *prompt* e a mensagem do usuário em mãos, o serviço seleciona dinamicamente o modelo da OpenAI mais adequado para a tarefa. Para interações textuais, o modelo **gpt-4o-mini** é utilizado por seu equilíbrio entre custo e capacidade, enquanto para a análise de imagens, o modelo **gpt-4.1-nano** foi escolhido por ser uma alternativa mais econômica e igualmente eficaz para essa finalidade.

A etapa de execução é iniciada pela invocação da função *generateText* da biblioteca Vercel AI SDK, que envia a requisição completa para a API da OpenAI. Neste ponto, o modelo de linguagem pode utilizar a funcionalidade de *Function Calling* para invocar uma ou mais das "ferramentas" (*tools*) disponíveis, permitindo-lhe acessar ou modificar dados na base da aplicação, como registrar uma transação ou consultar o saldo. Para aumentar a resiliência do sistema, foi implementado um mecanismo de autocorreção (*experimental_repairToolCall*) da Vercel AI SDK, que capacita a IA a tentar corrigir e reenviar chamadas de ferramentas que falharam inicialmente. Após a conclusão de todas as ferramentas necessárias, o modelo gera a resposta final em texto,

que é então encaminhada para a *pipeline* de mensagens de saída para ser entregue ao usuário, conforme detalhado na seção 4.3.2.

4.4.3. Monitoramento e observabilidade

Todas as interações com a API da OpenAI são roteadas através do Helicone, por meio de uma *baseUrl* e cabeçalhos de autenticação específicos (Helicone-Auth), ou seja, o SDK (*Software Development Kit*) envia a requisição para o Helicone, que funciona como um *middleware* que fará o redirecionamento para a API da OpenAI. Essa integração permite o monitoramento detalhado de custos, latência, taxas de sucesso e erro. Além disso, metadados customizados, como o telefone do usuário (*Helicone-Property-Phone*) e o tipo de mensagem (*Helicone-Tag-MessageType*), são enviados para enriquecer a análise e facilitar a depuração de interações (Figura 24).

```
...

this.openai = createOpenAI({
    apiKey: env.OPENAI_API_KEY,
    baseURL: "https://oai.helicone.ai/v1",
    headers: {
      "Helicone-Auth": `Bearer ${env.HELICONE_API_KEY}`,
    },
  });

...

const { response, text } = await generateText({
  headers: {
    "Helicone-Property-Phone": user.phone,
    "Helicone-Tag-MessageType": type,
  },
  ...
}

...
```

Figura 24: Trechos da instanciação do cliente da API OpenAI com Helicone.

4.5. Agendamento e Lembretes via Trigger.dev

Para a implementação da funcionalidade de agendamento de lembretes, optou-se pela utilização do Trigger.dev, um serviço de código aberto especializado na execução de tarefas agendadas. A solução foi implementada sob o plano gratuito da plataforma, cujo modelo de precificação é baseado no tempo de consumo de *CPU* por tarefa executada.

As tarefas de agendamento desenvolvidas neste projeto mostraram-se eficazes, com um tempo de execução inferior a três segundos, sendo o consumo de recursos computacionais mínimo. Dessa forma, a utilização da ferramenta manteve-se significativamente abaixo dos limites estabelecidos pelo plano, validando sua escolha como uma solução de custo operacional insignificante para o caso de uso apresentado.

4.5.1. Integração e fluxo de execução

A integração com o Trigger.dev se distingue por manter a lógica das tarefas como parte do código-fonte da aplicação, em vez de ser definida em uma interface externa. O processo de implantação e execução das tarefas segue um fluxo bem definido.

Primeiramente, as tarefas (*jobs*) são definidas como funções assíncronas em arquivos TypeScript, localizados no diretório *apps/bot/src/jobs* do projeto. Durante o processo de implantação, executa-se o script **pnpm run deploy:trigger**, que por sua vez roda **pnpm env:load** seguido de **pnpm dlx trigger.dev@3.3.17 deploy**, realizando a implantação para o Trigger.dev conforme as configurações especificadas no arquivo *trigger.config.ts*, conforme demonstrado na Figura 25. Segundo a documentação oficial: “[...] nós compilamos o código da sua tarefa usando nosso sistema de compilação, que é alimentado por esbuild. Durante a implantação, o código é empacotado em uma imagem Docker e implantado na sua instância Trigger.dev” [TRIGGER.DEV 2025].

```ts
import { env } from "@repo/env";
import { syncEnvVars } from "@trigger.dev/build/extensions/core";
import { prismaExtension } from "@trigger.dev/build/extensions/prisma";
import { defineConfig } from "@trigger.dev/sdk/v3";

export default defineConfig({
  project: env.TRIGGER_PROJECT_ID,
  runtime: "node",
  machine: "micro",
  logLevel: "log",
  maxDuration: 10, // 10 seconds
  retries: {
    enabledInDev: false,
    default: {
      maxAttempts: 1,
      minTimeoutInMs: 1000,
      maxTimeoutInMs: 10000,
      factor: 2,
      randomize: true,
    },
  },
  build: {
    extensions: [
      syncEnvVars(async () => {
        return Object.fromEntries(
          Object.entries(env).map(([key, value]) => [key, String(value)])
        );
      }),
      prismaExtension({
        version: "6.4.1",
        schema: "../../packages/database/prisma/schema.prisma",
        directUrlEnvVarName: "DATABASE_URL",
      }),
    ],
  },
  dirs: ["./src/libs/trigger"],
});
```

Figura 25: Arquivo de configuração do Trigger.dev: *trigger.config.ts*.

Quando um usuário solicita um lembrete — por exemplo, “me lembre de pagar a internet dia 5” —, a tool **createReminder** é executada (como descrito na seção 4.4.1 deste trabalho) e uma *task* é programada no Trigger.dev utilizando seu SDK. O *payload* da *task* inclui os dados necessários, como o usuário, a mensagem e a data/hora de execução.

A plataforma Trigger.dev armazena essa solicitação e se encarrega de orquestrar sua execução. No horário programado, o serviço aciona a *task* na nuvem. Para efetuar o envio do lembrete, é inserida uma mensagem na fila do BullMQ, a qual será processada pela aplicação em execução na máquina virtual, iniciando diretamente na pipeline de

mensagens saintes, conforme detalhado na seção 4.3.2. A Figura 26 ilustra os detalhes de uma *task* de envio de lembretes recorrentes que foi concluída com sucesso. Após o envio da mensagem, o sistema verifica, com base nos valores **nextReminderAt**, **intervalDays** e **maxConcurrences**, se uma nova *task* deve ser agendada. Caso positivo, a *task* pai agenda automaticamente a próxima *task* filha.

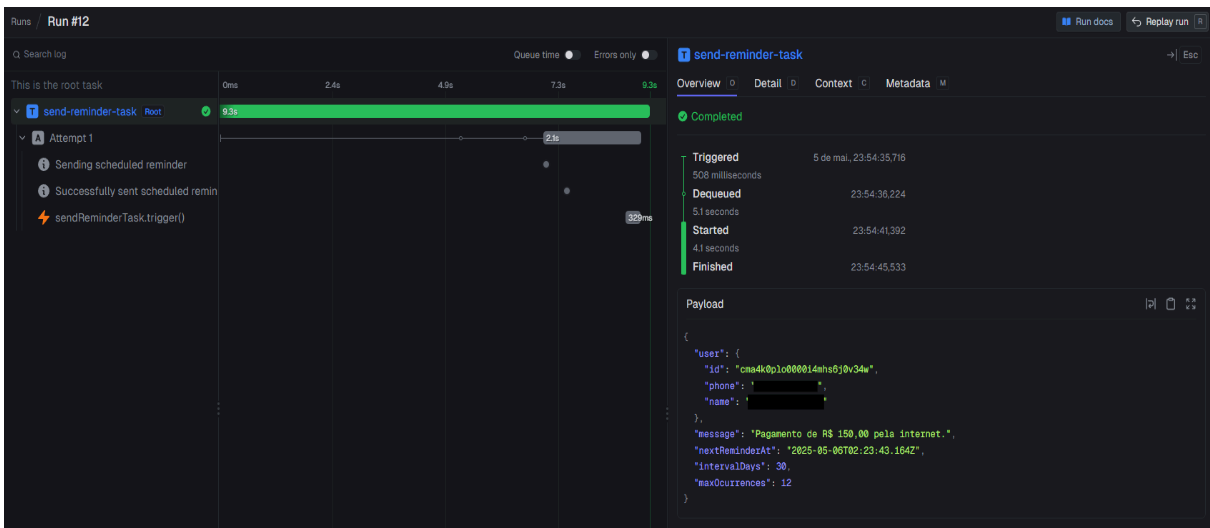


Figura 26: Detalhes de uma *task* concluída de envio de lembrete no Trigger.dev.

4.6. Monitoramento das filas: Bullboard

Para oferecer visibilidade operacional das tarefas assíncronas, foi incluída a interface Bull Board (*apps/bullboard*), ilustrada na Figura 27.

Essa ferramenta visual tem como propósito monitorar a saúde e o status das filas gerenciadas pelo BullMQ. Ela permite acompanhar em tempo real os jobs em diferentes estados — como ativos, aguardando, concluídos, falhados e atrasados — além de possibilitar a inspeção de dados e erros associados a cada job, bem como a realização de ações administrativas, como limpar filas ou executar novamente *jobs* que falharam. A tecnologia por trás da Bull Board é uma aplicação React independente, que se conecta diretamente à mesma instância Redis utilizada pelo BullMQ para acessar os dados das filas [BULLMQ 2025].

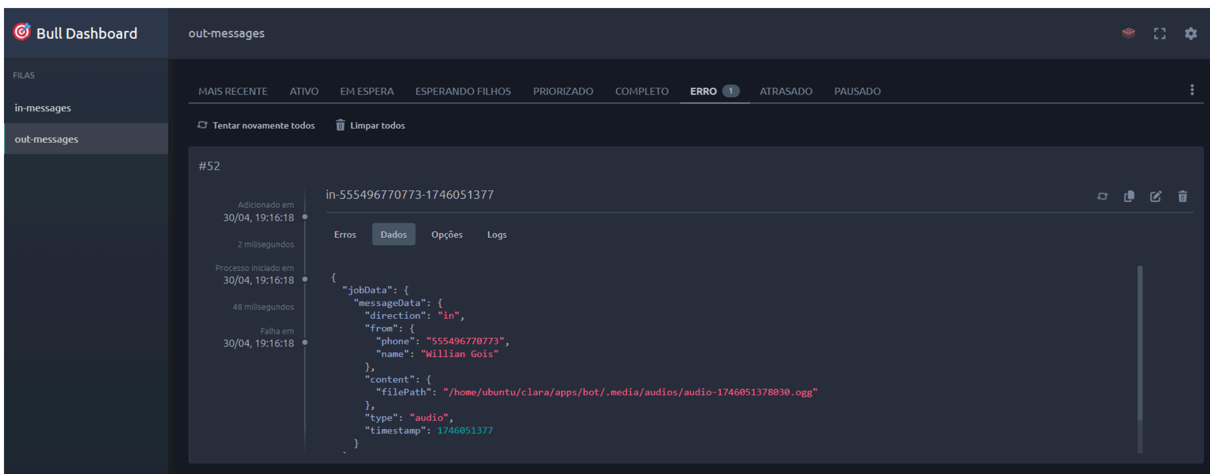


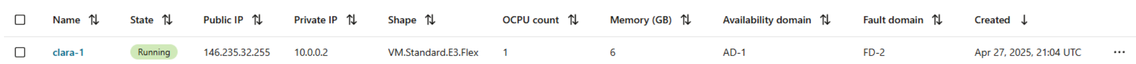
Figura 27: Dashboard do Bullboard, mostrando as informações de *job* que falhou.

4.7. Ambiente de produção

Para validar a performance, a estabilidade e a viabilidade financeira da Clara em um cenário realista, foi provisionado um ambiente de produção na OCI (*Oracle Cloud Infrastructure*). A escolha desta plataforma se deu pela sua robustez e pela oferta de uma cota gratuita generosa, que permitiu uma avaliação de custos operacionais com recursos que excedem a camada gratuita.

4.7.1. Provisionamento e Configuração da Infraestrutura

Foi provisionada uma Máquina Virtual dedicada, denominada clara-1, do tipo VM.Standard.E3.Flex, baseada em processadores AMD EPYC (Figura 28).



Name	State	Public IP	Private IP	Shape	OCPU count	Memory (GB)	Availability domain	Fault domain	Created
clara-1	Running	146.235.32.255	10.0.0.2	VM.Standard.E3.Flex	1	6	AD-1	FD-2	Apr 27, 2025, 21:04 UTC

Figura 28: Detalhes e configurações da VM na Oracle Cloud.

Essa instância recebeu a alocação de 1 OCPU (Oracle CPU) e 6 GB de RAM, uma configuração que se mostrou mais do que suficiente para hospedar toda a *stack* tecnológica da aplicação. O sistema operacional escolhido foi o Ubuntu 22.04 LTS, amplamente adotado no mercado devido à sua estabilidade e ao suporte contínuo da comunidade (Figura 29).

Image details

Operating system	Canonical Ubuntu
Version	22.04
Image	Canonical-Ubuntu-22.04-2025.03.28-0

Figura 29: Imagem Ubuntu utilizada na VM da Oracle Cloud.

A rede foi configurada com um endereço de IP público estático (Figura 30), recurso essencial para garantir a comunicação ininterrupta entre a biblioteca Baileys e os servidores do WhatsApp, além de viabilizar a futura migração para os *webhooks* da API oficial.

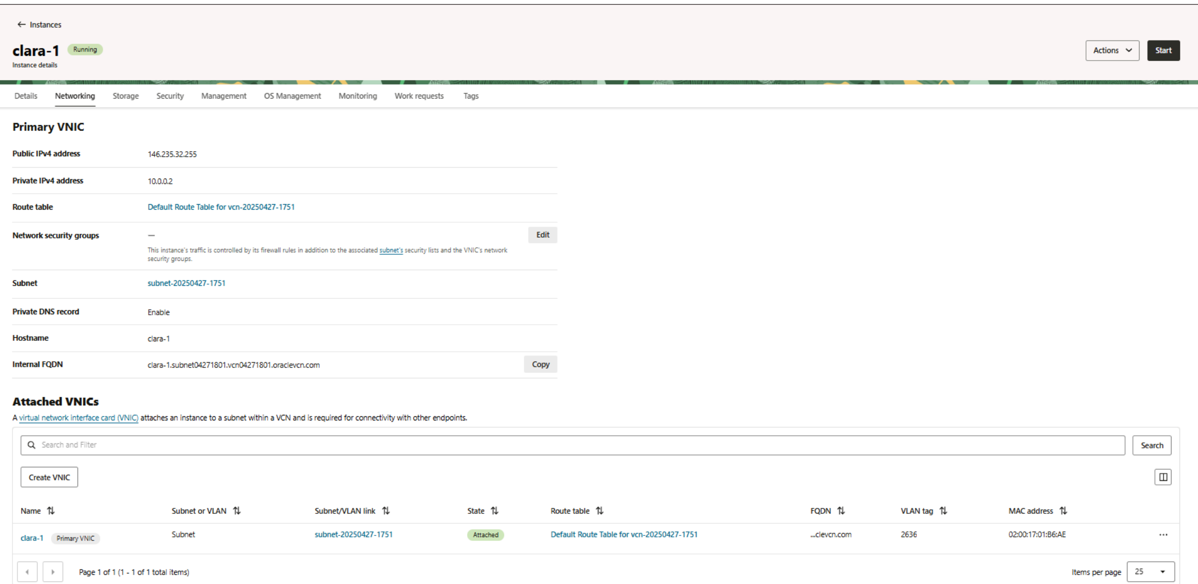


Figura 30: Configurações de rede da VM.

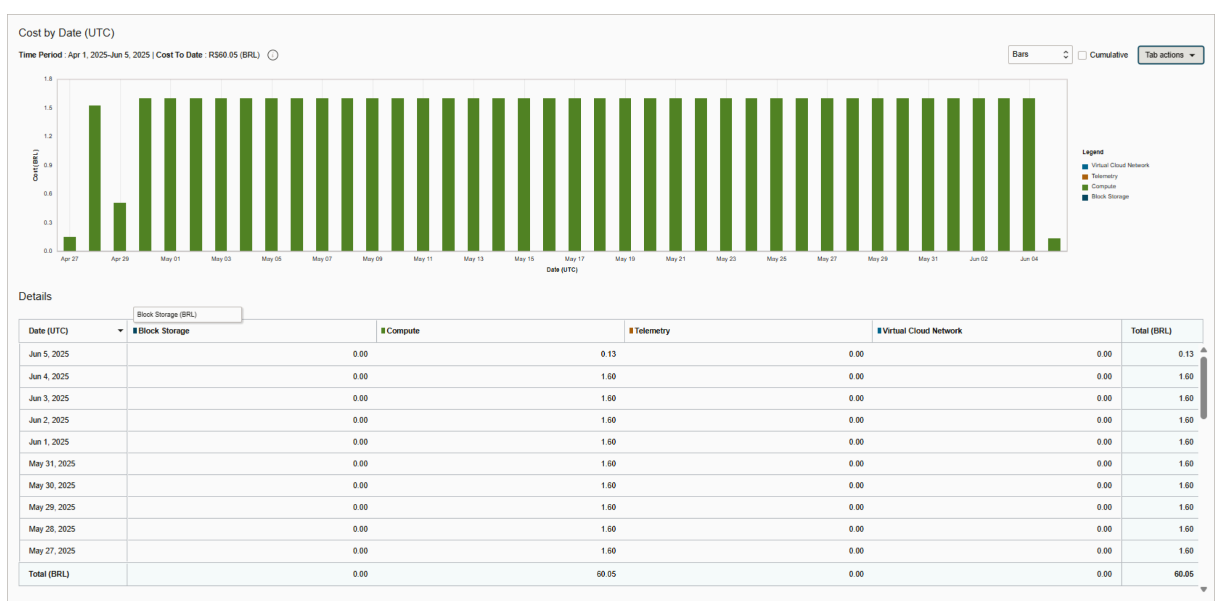
Neste ambiente, o Docker e o Docker Compose foram utilizados para orquestrar a execução de todos os serviços da aplicação de forma conteinerizada, incluindo o bot

principal, o banco de dados MySQL, o *cache* com Redis e a interface de monitoramento de filas Bullboard.

4.7.2. Análise de Custos Operacionais

O monitoramento de custos no período de 27 de abril a 5 de jun. de 2025 (Figura 31) revelou uma alta eficiência financeira. O custo total acumulado no período foi de R$ 60,05. A análise demonstrou que após estabilização inicial, o custo diário permaneceu constante em aproximadamente R$ 1,60.

Este valor, projetado para um mês completo, resulta em um custo operacional mensal estimado de R$ 48,00. É notável que a quase totalidade do custo (99%+) foi atribuída ao serviço de *Compute* (a VM em si), com o custos de *Block Storage* e Virtual Cloud Network sendo desprezível para a escala do projeto. Este baixo custo operacional valida a viabilidade financeira da solução, tornando-a acessível mesmo em um cenário de produto inicial.



Date (UTC)	Block Storage	Compute	Telemetry	Virtual Cloud Network	Total (BRL)
Jun 5, 2025	0.00	0.13	0.00	0.00	0.13
Jun 4, 2025	0.00	1.60	0.00	0.00	1.60
Jun 3, 2025	0.00	1.60	0.00	0.00	1.60
Jun 2, 2025	0.00	1.60	0.00	0.00	1.60
Jun 1, 2025	0.00	1.60	0.00	0.00	1.60
May 31, 2025	0.00	1.60	0.00	0.00	1.60
May 30, 2025	0.00	1.60	0.00	0.00	1.60
May 29, 2025	0.00	1.60	0.00	0.00	1.60
May 28, 2025	0.00	1.60	0.00	0.00	1.60
May 27, 2025	0.00	1.60	0.00	0.00	1.60
Total (BRL)	0.00	60.05	0.00	0.00	60.05

Figura 31: Gráfico e tabela com custos diários da Oracle Cloud em R$.

4.7.3. Desempenho e Estabilidade

A configuração de 1 OCPU e 6 GB de RAM provou ser amplamente adequada para a carga de testes realizada. O monitoramento contínuo não indicou gargalos de CPU ou

memória, mesmo durante operações mais intensas como o processamento de áudios ou a geração de gráficos. A infraestrutura da VM em si apresentou 100% de estabilidade durante todo o período de avaliação.

Contudo, é importante ressaltar a distinção entre a estabilidade da infraestrutura e a estabilidade da aplicação. Conforme discutido nos desafios do projeto, a principal fonte de instabilidade foi a conexão da biblioteca não oficial Baileys. As quedas e reinicializações do serviço, notificadas pelos alertas configurados via Telegram, foram decorrentes de eventos na camada da aplicação, e não de falhas na máquina virtual. Este ambiente de produção foi, portanto, crucial para isolar e confirmar que o principal gargalo de confiabilidade da Clara reside na sua dependência da biblioteca de integração.

5. Resultados e Discussão

A fase de implementação e testes do projeto Clara gerou um conjunto de resultados quantitativos e qualitativos que permitem avaliar o sucesso em relação aos objetivos propostos, identificar os desafios enfrentados e extrair lições valiosas para um possível futuro do projeto.

5.1. Métricas de performance e escalabilidade

A análise do desempenho do sistema, baseada nos dados coletados pelas ferramentas de observabilidade (Sentry, Helicone) e monitoramento da VM Oracle, forneceu indicativos quantitativos sobre a eficiência, o custo operacional e a capacidade de escalonamento da arquitetura implementada. Os resultados validam a viabilidade técnica e financeira da solução no contexto de um protótipo.

As principais métricas observadas durante o período de testes e 45 dias de produção estão resumidas no quadro abaixo:

Quadro 2: Métricas de performance e escalabilidade e respectivos valores.

Categoria	Métrica	Valor Observado	Ferramenta/Fonte

Infraestrutura	Custo operacional (estimado)	~ R$ 60,05	Oracle Cloud (Figura 31)
	Disponibilidade da VM	100%	Oracle Cloud Monitoring
Desempenho do LLM	Custo total (IA)	$0.46 (dólar)	*Dashboard* OpenAI (Figura 32)

Em relação ao tempo de resposta da IA, não foi possível obter a média da API da OpenAI, pois o plano gratuito utilizado no Helicone limitou o intervalo de tempo disponível para a geração de relatórios de uso. Em relação ao custo total da IA, considerou-se o valor acumulado com o uso da API da OpenAI durante o período de desenvolvimento e os testes realizados em ambiente de produção (Figura 32).

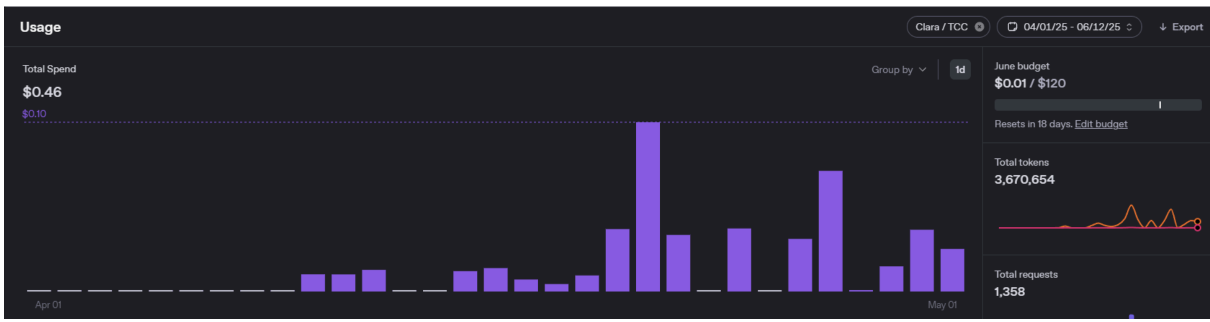


Figura 32: Custo total com a API da OpenAI no período de testes e produção (01/abril até 12/junho de 2025)

A Máquina Virtual utilizada na Oracle Cloud, configurada com 1 OCPU e 6 GB de RAM, demonstrou ser plenamente suficiente para a carga de testes, operando sem gargalos de CPU ou memória. A infraestrutura manteve 100% de estabilidade durante todo o período de uso, com um custo operacional total de R$ 60,05, o que comprova a eficiência financeira da solução adotada.

A observabilidade do processamento assíncrono foi viabilizada principalmente por meio da análise via Bull Board (Figura 27), que revelou um sistema de filas robusto, onde os erros em jobs foram prontamente identificados e atribuídos a fatores externos,

como instabilidades na conexão da biblioteca Baileys ou falhas no processamento de arquivos de áudio, sem relação com o mecanismo de enfileiramento em si.

Em relação à escalabilidade, a arquitetura mostrou forte potencial para crescimento vertical, mediante o aumento dos recursos da própria VM. Contudo, a limitação imposta pela biblioteca não oficial Baileys, que permite apenas uma conexão ativa por número de telefone, foi identificada como o principal gargalo, este fator limita a capacidade de distribuir a carga de entrada entre múltiplas instâncias do *bot*, reforçando que a migração para a API oficial da Meta é um passo indispensável para um produto em escala de produção.

5.2. Retorno dos usuários de teste (satisfação, bugs, sugestões)

Os testes com usuários, embora tenham envolvido 20 participantes, resultaram em apenas 9 respostas ao formulário de *feedback* do Google Forms (Figura 33), configurando uma amostra qualitativa de escopo limitado, porém valiosa. Essas respostas forneceram insights cruciais sobre a usabilidade, utilidade e aceitação da Clara, abrangendo desde pessoas sem experiência prévia em controle financeiro até usuários de aplicativos consolidados como o Mobills.

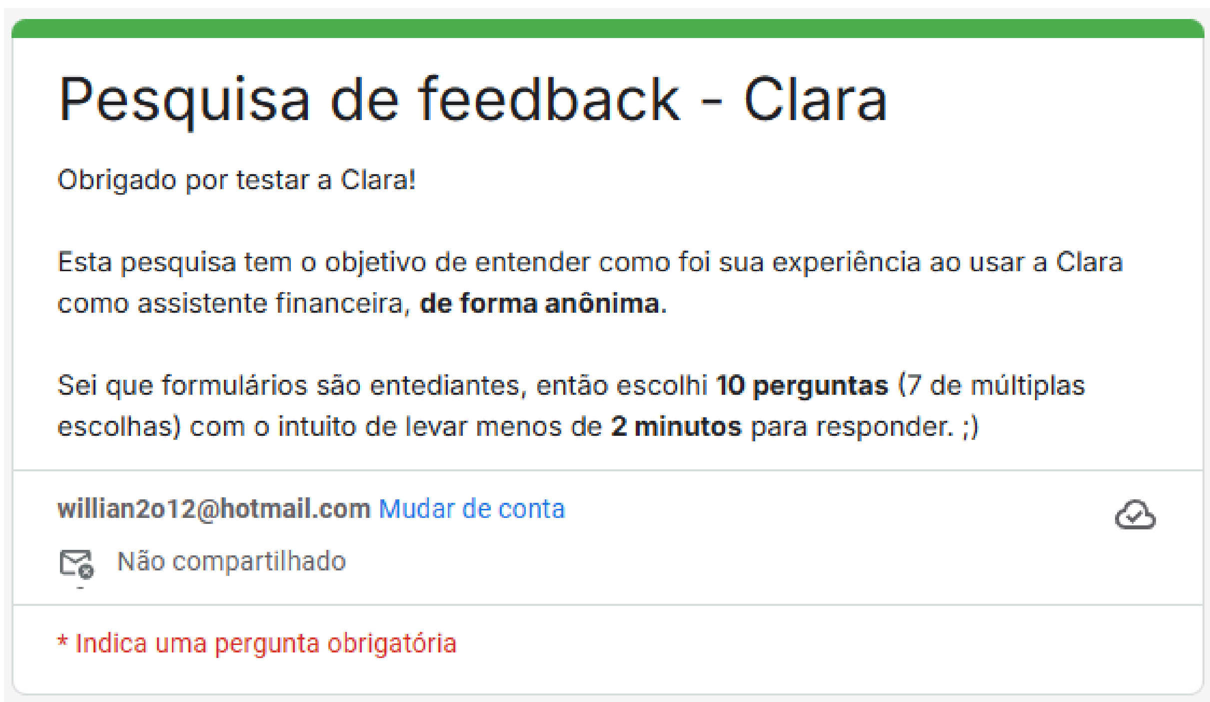


Figura 33: Cabeçalho do formulário de pesquisa no Google Forms.

De modo geral, o sentimento dos usuários foi positivo, validando a premissa central do projeto: a conveniência de gerenciar finanças dentro do WhatsApp é um diferencial significativo. A facilidade de uso e a praticidade para registrar transações rápidas foram os pontos mais elogiados. Contudo, essa satisfação é ponderada por três temas recorrentes: a necessidade de funcionalidades mais avançadas, a ocorrência de instabilidade que afeta a confiança e a preferência de alguns usuários por uma interface mais guiada (com botões) em detrimento de uma interação puramente conversacional.

A satisfação geral com a experiência foi alta. Em uma escala de 1 a 5, a facilidade de uso recebeu uma nota média de 4,2 (Figura 34), indicando que a maioria dos usuários conseguiu interagir com a Clara de forma intuitiva. A utilidade da ferramenta para tarefas financeiras também foi bem avaliada, com média de 4,1 (Figura 35). Um usuário destacou que "a praticidade da Clara resolve esse problema" de esquecer de preencher planilhas. Outro ponto de grande satisfação foi a funcionalidade de lembretes, descrita como "a melhor coisa" por um dos participantes. A clareza e a compreensibilidade das respostas também foram avaliadas, atingindo uma nota média

de 4,1 (Figura 36). A confiança nas informações fornecidas foi igualmente alta, com média de 4,2 (Figura 37).

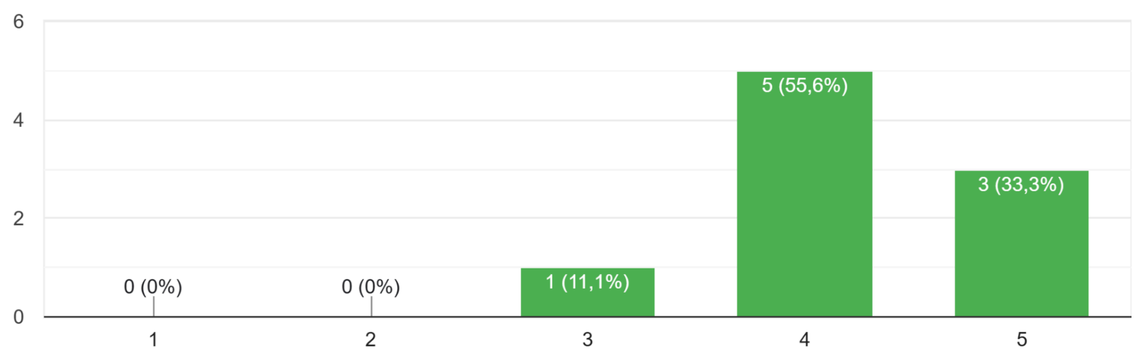


Figura 34: Avaliação da facilidade de uso da Clara (escala de 1 a 5).

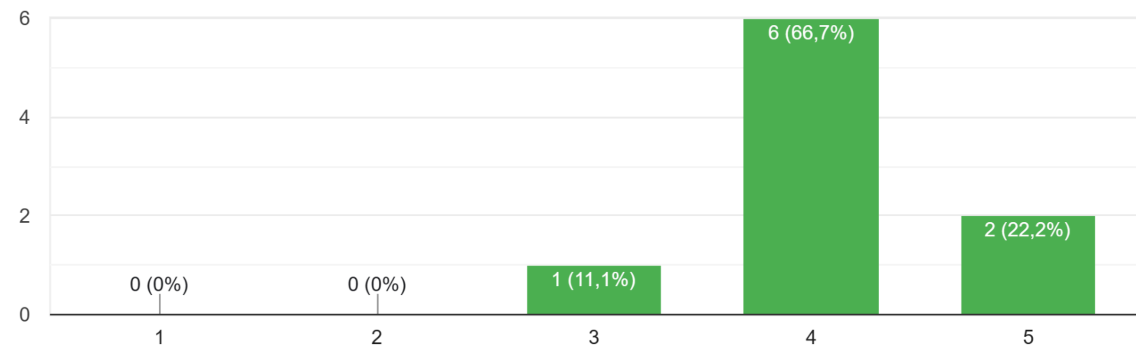


Figura 35: Avaliação de utilidade da Clara para dúvidas ou tarefas financeiras (escala de 1 a 5).

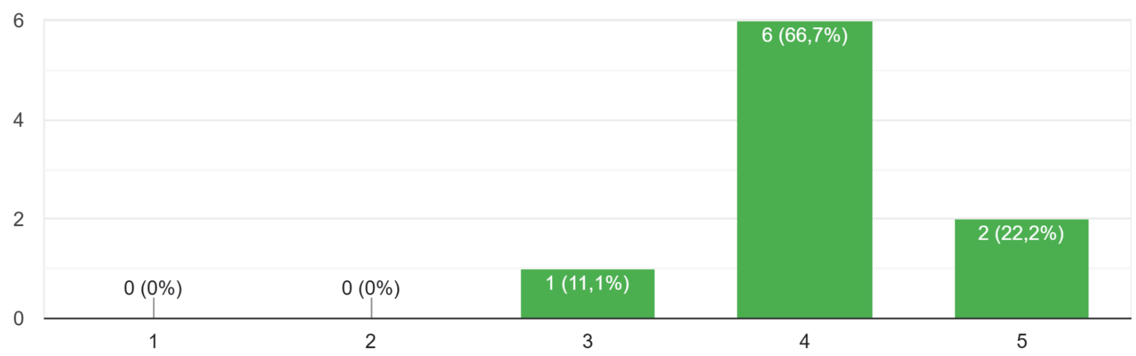


Figura 36: Avaliação da capacidade de resposta da Clara (escala de 1 a 5).

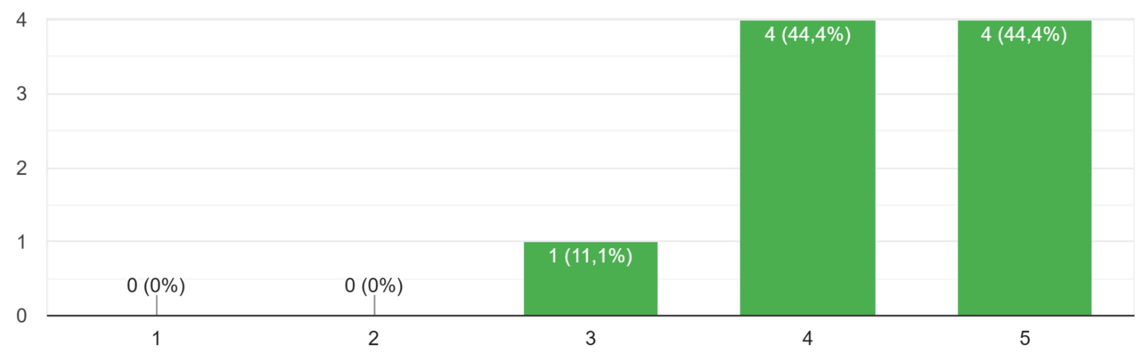


Figura 37: Avaliação das informações fornecidas pela Clara (escala de 1 a 5).

Apesar da boa avaliação geral, foram apontados problemas importantes que impactaram a experiência do usuário. Um dos principais foi a limitação no fluxo de recuperação de erros. Identificou-se uma falha significativa de usabilidade: a impossibilidade de editar uma transação registrada incorretamente. Um usuário relatou que, ao digitar um valor errado, não encontrou uma forma de corrigir a informação e precisou registrar novamente, o que resultou em um saldo incorreto — "Tive que

registrar de novo e o saldo ficou bagunçado", afirmou. Essa ausência de um fluxo de correção representa um grande ponto de atrito.

Outro problema observado foi a ambiguidade na interface conversacional. Um usuário sugeriu a inclusão de botões para ações comuns, como "Digite 1 para ver seu saldo", e comentou que "conversar livremente às vezes é mais difícil do que parece". Esse feedback indica que, para parte do público, uma interface puramente conversacional pode ser menos eficiente do que uma abordagem híbrida ou guiada.

Além disso, os usuários mencionaram funcionalidades desejadas para a evolução da Clara:

- **Definição de orçamentos e metas (2 respostas):** A capacidade de estabelecer limites de gastos por categoria (ex: "não gastar mais de R$ 500 em restaurante").
- **Gestão avançada de transações (2 respostas)**: Incluindo a edição de registros existentes e o planejamento de despesas futuras.
- **Controle de cartão de crédito (1 resposta):** Funcionalidade para gerenciar faturas e compras parceladas.
- **Personalização e exportação (1 resposta):** Permitir a criação de categorias de despesas personalizadas e a exportação de relatórios em formatos como PDF.
- **Integração bancária (1 resposta):** A sugestão de integração com contas bancárias, possivelmente via Open Finance, foi mencionada como um recurso de grande valor agregado

Por fim, o feedback dos usuários validou parcialmente a premissa central de que existe interesse em gerenciar finanças via WhatsApp devido ao baixo atrito. A alta adesão de usuários que antes não utilizavam método algum ou se sentiam sobrecarregados por planilhas comprova o potencial do canal. A maioria dos respondentes (quatro de nove) afirmou que pagaria pelo serviço (Figura 38) — alguns, condicionando o pagamento à adição de novas funcionalidades. Dentre os que pagariam, o valor de R$ 19,90 mensais foi o mais selecionado (quatro respostas), superando as opções de R$ 29,90 e R$ 39,90 mensais (Figura 39).

Você pagaria por um assistente financeiro como a Clara?
9 respostas

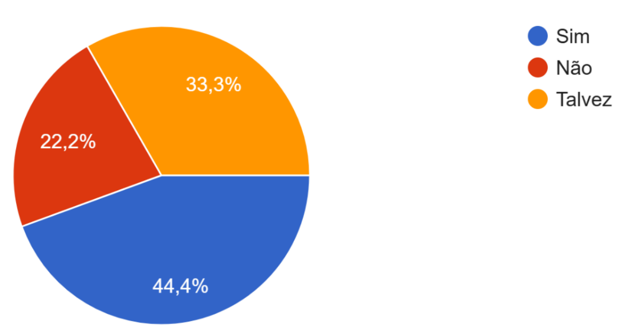


Figura 38: Avaliação de possível uso pago da Clara.

Se sim, até quanto?
5 respostas

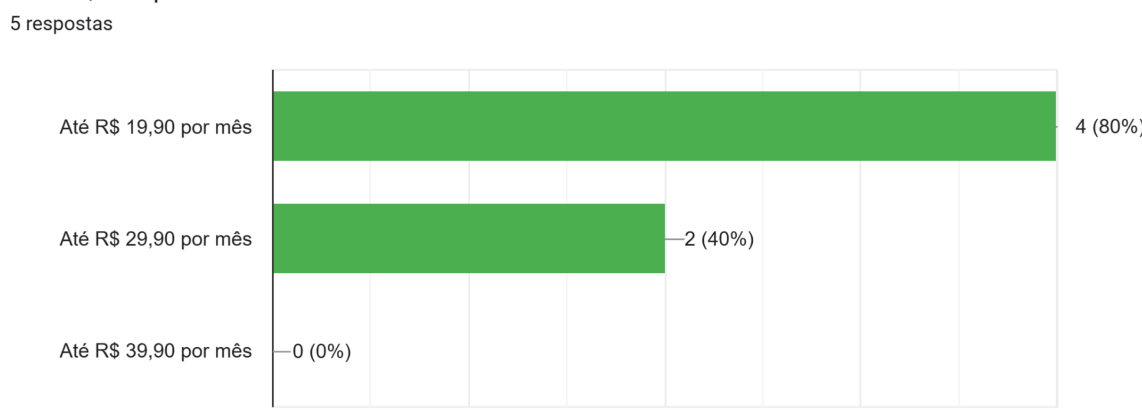


Figura 39: Avaliação de uma possível precificação para a Clara como produto.

Contudo, a análise também destacou a importância de uma interface conversacional bem projetada, respostas precisas e confiáveis da IA, e a necessidade de construir confiança para que os usuários se sintam confortáveis realizando tarefas financeiras neste ambiente. A falta de funcionalidades básicas como a edição de transações e o *design* da conversação são barreiras a serem superadas.

Um ponto de atenção crucial é a nota média de 3.5 (em 5) para a pergunta sobre a probabilidade de uso diário do assistente, um valor relativamente inferior às demais métricas de satisfação, que superaram 4.0 (Figura 40). Essa pontuação pode ser um indicativo de que as barreiras mencionadas impactam diretamente a intenção de adoção contínua. Alternativamente, e de forma mais provável, pode sinalizar um desalinhamento com o público-alvo testado. Uma vez que o formulário foi respondido por uma amostra de conveniência (colegas, amigos e conhecidos sem um perfil de usuário pré-definido), a percepção de necessidade e o interesse genuíno na solução podem ter sido naturalmente diluídos, resultando em uma menor propensão ao uso diário.

Apesar disso, o protótipo cumpriu seu papel de validar o conceito e, mais importante, de revelar os requisitos fundamentais e os desafios de adequação ao mercado para o prosseguimento do projeto.

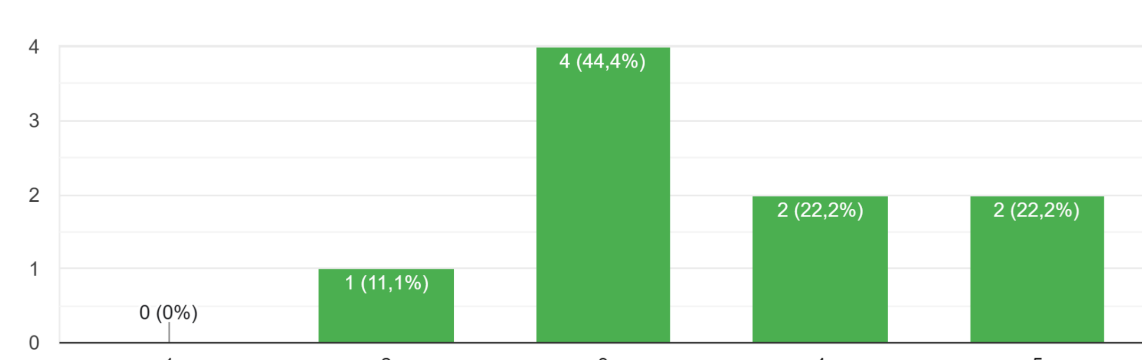


Figura 40: Avaliação da possibilidade de uso da Clara no dia a dia (escala de 1 a 5).

5.3. Desafios e lições aprendidas

A jornada de desenvolvimento e validação da Clara serviu como um campo de provas realista, revelando tanto os obstáculos técnicos inerentes a sistemas modernos quanto os aprendizados estratégicos que deles surgiram. No âmbito técnico, o primeiro desafio percebido foi a orquestração de um ecossistema de serviços distribuídos e assíncronos. A necessidade de conectar componentes como BullMQ, Trigger.dev, Prisma e a API da

OpenAI exigiu uma depuração cuidadosa das interfaces entre eles, com especial atenção ao tratamento de falhas em um ambiente onde as operações não são sequenciais. Contudo, a maior fonte de complexidade técnica residiu na integração com o WhatsApp por meio da biblioteca não oficial Baileys. Sua natureza, baseada em engenharia reversa, resultou em instabilidades frequentes, documentação escassa e a falta de um caminho claro para a escalabilidade horizontal. A necessidade de autenticação via *QR code* ou código único e a exigência de manter uma instância ativa dificultam o uso em ambientes que precisam ser escaláveis. Isso demandou a implementação de mecanismos de reconexão e monitoramento reativo, confirmando que, embora viável para prototipagem, tal abordagem representa um gargalo significativo para um ambiente de produção estável.

Em paralelo aos desafios de infraestrutura, a criação de uma experiência do usuário eficaz apresentou seus próprios obstáculos, sobretudo no *design* da interação conversacional. O processo de engenharia de *prompts* para a OpenAI foi iterativo e complexo, buscando um equilíbrio entre respostas simples, precisas e de baixo custo computacional. Além disso, projetar fluxos de diálogo que fossem ao mesmo tempo intuitivos para o usuário e robustos o suficiente para lidar com a ambiguidade da linguagem natural mostrou-se uma tarefa não trivial, mostrando que a inteligência da ferramenta deve ser acompanhada por um *design* de interação cuidadosamente planejado.

Essa experiência mostrou lições para o projeto. A mais importante foi a confirmação do valor inestimável de uma *stack* de observabilidade robusta. Ferramentas como Sentry, Helicone e a Bull Dashboard não foram apenas úteis na fase de produção, mas indispensáveis durante o desenvolvimento, permitindo a rápida detecção e correção de erros em um sistema intrinsecamente complexo. Adicionalmente, a decisão de aplicar padrões de *design* como *Pipeline* e *Strategy* foi validada, pois proporcionaram a organização e a manutenibilidade necessárias para gerenciar a lógica do *chatbot* de forma escalável. Por fim, o *feedback* direto dos usuários foi crucial, atuando como o elo final que conectou a implementação técnica às necessidades reais, direcionando prioridades e revelando falhas de usabilidade que métricas técnicas sozinhas não capturariam. A complexidade da integração mostra o desafio que reside em orquestrar

diferentes sistemas (APIs externas, filas, IA, agendadores) de forma harmoniosa e confiável, tornando a robustez da arquitetura e a qualidade da observabilidade fatores de sucesso.

6. Conclusão

O projeto Clara demonstrou com sucesso a viabilidade técnica de construir um assistente financeiro baseado em IA operando dentro do WhatsApp, utilizando uma *stack* tecnológica moderna centrada em TypeScript e Node.js. A arquitetura implementada, empregando um monorepo gerenciado por Turborepo, filas assíncronas com BullMQ/Redis, agendamento com Trigger.dev, validação com Zod e acesso a dados *type-safe* com Prisma, provou ser capaz de suportar as funcionalidades propostas, incluindo interação conversacional via OpenAI e tratamento de diferentes tipos de mensagens.

Os testes realizados em um ambiente de produção simulado e com usuários reais indicaram um desempenho satisfatório em termos de resposta e processamento, dentro das limitações esperadas devido ao uso de uma biblioteca não oficial para a integração com o WhatsApp. O *feedback* dos usuários, embora com uma amostra limitada (9 respostas de 20 testadores), validou o interesse na conveniência oferecida pelo canal de mensagens para interações financeiras, mas também sublinhou a necessidade crítica de aprimorar a inteligência conversacional, a clareza das informações e, fundamentalmente, construir a confiança do usuário. Os desafios encontrados, particularmente na integração de sistemas complexos e no *design* da experiência conversacional, forneceram aprendizados valiosos sobre a implementação de tais soluções no mundo real.

É importante, contudo, reconhecer as limitações deste projeto. Trata-se de um protótipo funcional, e os testes, embora realizados com usuários reais, foram limitados em escopo e duração. A avaliação de escalabilidade foi baseada em testes sob carga típica, não em testes de estresse exaustivos. Além disso, a dependência de APIs de terceiros (WhatsApp, OpenAI, etc.) introduz fatores externos que podem afetar a performance e a disponibilidade. A análise de segurança e conformidade com a LGPD, embora abordada tecnicamente, requer uma auditoria formal para validação completa em um cenário de produção real.

Apesar dessas limitações, os resultados e aprendizados abrem um vasto campo para desenvolvimentos futuros. A evolução natural do protótipo reside na expansão funcional e no aprimoramento de sua inteligência. Isso inclui a incorporação de recursos mais sofisticados de gestão financeira, como a possibilidade de categorizar despesas futuras e de cartões de crédito; a possibilidade de edição e remoção de transações registradas; o acompanhamento de orçamentos e a integração com o ecossistema *Open Finance* para uma visão consolidada. Paralelamente, seria crucial investir em técnicas mais avançadas de processamento de linguagem natural para refinar a compreensão de intenções complexas, personalizar as interações e oferecer *insights* financeiros proativos, sempre respeitando os limites regulatórios e éticos.

Além do aprimoramento do produto, futuras investigações deveriam explorar a adaptação da solução para múltiplas interfaces, como assistentes de voz, como por exemplo a Alexa da Amazon, e conduzir testes em larga escala. Um estudo com um público maior e mais diversificado, por um período prolongado, forneceria dados robustos sobre usabilidade e padrões de uso, essenciais para guiar o desenvolvimento contínuo. Tais esforços convergiriam, em última análise, para a produtização da Clara, transformando o protótipo em um produto comercialmente viável, o que demandaria uma análise aprofundada de modelos de negócio e estratégias de monetização.

Em suma, o projeto Clara estabeleceu uma base técnica sólida e validou o interesse inicial em assistentes financeiros conversacionais no WhatsApp no contexto brasileiro. Os próximos passos envolvem refinar a experiência do usuário, expandir as funcionalidades e abordar os desafios inerentes à construção de confiança e à navegação no complexo ecossistema de APIs. Para garantir escalabilidade, estabilidade e conformidade a longo prazo, será fundamental migrar para a API oficial do WhatsApp, que oferece suporte formal, infraestrutura robusta e recursos específicos para negócios.

7. Referências Bibliográficas

BAILEYS. (2025) **"Lightweight full-featured typescript/javascript WhatsApp Web API"**, https://github.com/WhiskeySockets/Baileys. Acesso em: 06 jun. 2025.

BR.DID. (2025) **"BR DID - Números virtuais VOIP"**, https://brdid.com.br/. Acesso em: 06 jun. 2025.

BULLMQ.IO. (2025) **“What is BullMQ | BullMQ“**, https://docs.bullmq.io. Acesso em: 06 jun. 2025.

Campos, Álvaro. (2024) **“Fintech Magie recebe aporte de R$ 28 milhões liderado pelo fundo Lux Capital“**, https://valor.globo.com/financas/noticia/2024/08/22/fintech-magie-recebe-aporte-de-r28-milhes-liderado-pelo-fundo-lux-capital.ghtml. Acesso em: 8 jun. 2025.

CNDL. (2024) **“35% dos inadimplentes não fazem controle das contas e dos gastos, revela pesquisa CNDL/SPC Brasil“**, https://site.cndl.org.br/35-dos-inadimplentes-nao-fazem-controle-das-contas-e-dos-gastos-revela-pesquisa-cndlspc-brasil-2/. Acesso em: 8 jun. 2025.

DRAGONFLY. (2025) **“Question: What is the difference between BullMQ and RabbitMQ?“**, https://www.dragonflydb.io/faq/bullmq-vs-rabbitmq. Acesso em: 07 jun. 2025.

DOCKER. (2025) **“What is Docker?“**, https://docs.docker.com/get-started/docker-overview/. Acesso em: 06 jul. 2025.

DOCKER COMPOSE. (2025) **“Overview of Docker Composer”**, https://docs.docker.com/compose/. Acesso em: 06 jul. 2025.

DOCKER COMPOSE. (2025) **“Multiple Compose files“**, https://docs.docker.com/compose/production/#use-multiple-compose-files. Acesso em: 06 jul. 2025.

GRAND VIEW RESEARCH. (2024) **“Chatbot market size, share & growth, 2024–2030“**, https://www.grandviewresearch.com/industry-analysis/chatbot-market. Acesso em: 04 maio 2025.

HELICONE. (2025) **“Helicone Getting Started“**, https://docs.helicone.ai/getting-started/quick-start. Acesso em: 06 jun. 2025.

IBM. (2025) **“O que é um message broker“**, https://www.ibm.com/br-pt/topics/message-brokers. Acesso em: 06 jul. 2025.

Ihnatovich, Dzmitry. (2025) **“Monorepos with Turborepo“**, https://medium.com/@ignatovich.dm/monorepos-with-turborepo-6aa0852708ee. Acesso em: 09 jun. 2025.

Kemp, Simon. (2025) **“Digital 2025: Brazil“**, https://datareportal.com/reports/digital-2025-brazil. Acesso em: 05 abr. 2025.

MONOREPO.TOOLS. (2025) **“Monorepo Explained: What is a Monorepo?“**, https://monorepo.tools/#what-is-a-monorepo. Acesso em: 06 jul. 2025.

MYSQL. **“MySQL.com“**. https://www.mysql.com/. Acesso em: 07 jul. 2025.

NODE.JS. (2025) **“Node.js - Run JavaScript Everywhere“**, https://nodejs.org/en. Acesso em: 07 jul. 2025.

OPENAI. (2025) **“Function Calling | OpenAI API“**, https://platform.openai.com/docs/guides/function-calling. Acesso em: 09 jun. 2025.

OPENAI. (2025) **“Speech to text | OpenAI API“**, http://platform.openai.com/docs/guides/speech-to-text. Acesso em: 08 jun. de 2025.

OPENAI. (2025) **“Structured Outputs | OpenAI API“**, https://platform.openai.com/docs/guides/structured-outputs. Acesso em: 09 jun. 2025.

PNPM. (2025) **“pNpm: Gestor de pacotes rápidos e eficientes do espaço em disco“**, https://pnpm.io/pt/. Acesso em: 06 jun. 2025.

PRISMA.IO. (2025) **“Prisma Documentation“**, https://www.prisma.io/docs. Acesso em: 06 jun. 2025.

REDIS. (2025) **“Redis - The Real-time Data Platform“**, https://redis.io/. Acesso em: 07 jul. 2025.

SENTRY. (2025) **“Application Performance Monitoring & Error Tracking Software“**, https://sentry.io/. Acesso em: 08 jun. 2025.

SERASA. (2025) **“Mapa da Inadimplência e Negociação de Dívidas no Brasil“**, https://www.serasa.com.br/limpa-nome-online/blog/mapa-da-inadimplencia-e-renegociacao-de-dividas-no-brasil/. Acesso em: 8 jun. 2025.

Souza, Leandro Miguel. (2024) **“"Powered by AI", Magie quer ser banco dentro do WhatsApp“**, https://startups.com.br/negocios/fintech/powered-by-ia-magie-quer-ser-banco-dentro-do-whatsapp/. Acesso em: 8 jun. 2025.

TRIGGER.DEV. (2025) **“How it works: the build system“**, https://trigger.dev/docs/how-it-works#the-build-system. Acesso em: 10 jun. 2025.

TRIGGER.DEV. (2025) **“Welcome to the Trigger.dev docs“**, https://trigger.dev/docs. Acesso em: 06 jun. 2025.

TURBOREPO. (2025) **“Introduction | Turborepo“**, https://turborepo.com/docs. Acesso em: 06 jun. 2025.

TYPESCRIPT. (2025) **“TypeScript: JavaScript with syntax for types“**, https://www.typescriptlang.org/. Acesso em: 07 jul. 2025.

META. (2025) **“API de nuvem do WhatsApp“**, https://developers.facebook.com/docs/whatsapp/cloud-api/?locale=pt_BR. Acesso em: 06 jun. 2025.

VSCODE. (2025) **“The open source AI code editor“**, https://code.visualstudio.com/. Acesso em: 07 jul. 2025.

WHATSAPP. (2025) **“Preços da Plataforma WhatsApp Business“**, https://business.whatsapp.com/products/platform-pricing. Acesso em: 07 jul. 2025.

WHATSAPP. (2025) **“Termos de Serviço do WhatsApp“**, 4 jan. 2021. https://www.whatsapp.com/legal/terms-of-service. Acesso em: 09 jun. 2025.

YUGASA. (2025) **“Whatsapp Chatbots: The new digital revolution“**, https://yugasa.com/frontend/images/WhatsApp-Chatbots-vs.-Mobile-Apps.pdf. Acesso em: 04 maio 2025.

ZOD. (2025) **“TypeScript-first schema validation with static type inference“**, https://zod.dev/. Acesso em: 06 jun. 2025.