UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO
INSTITUTO MULTIDISCIPLINAR

VICTOR REZENDE DE LIMA

Observator: Um Gateway para Observabilidade de LLM

Prof. Filipe Braida do Carmo, D.Sc.
Orientador

Nova Iguaçu, Dezembro de 2025

Observator: Um Gateway para Observabilidade de LLM

Victor Rezende de Lima

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto Multidisciplinar da Universidade Federal Rural do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Apresentado por:

_____

Victor Rezende de Lima

Aprovado por:

_____

Prof. Filipe Braida do Carmo, D.Sc.

_____

Prof. Bruno José Dembogurski, D.Sc.

_____

Prof. Fellipe Ribeiro Duarte, D.Sc.

NOVA IGUAÇU, RJ - BRASIL

Dezembro de 2025

DOCUMENTOS COMPROBATÓRIOS Nº 33250/2025 - CoordCGCC (12.28.01.00.00.98)

(Nº do Protocolo: NÃO PROTOCOLADO)

(Assinado digitalmente em 17/12/2025 12:55)
BRUNO JOSE DEMBOGURSKI
PROFESSOR DO MAGISTERIO SUPERIOR
DeptCC/IM (12.28.01.00.00.83)
Matrícula: ###249#4

(Assinado digitalmente em 16/12/2025 10:39)
FELLIPE RIBEIRO DUARTE
PROFESSOR DO MAGISTERIO SUPERIOR
DeptCC/IM (12.28.01.00.00.83)
Matrícula: ###890#4

(Assinado digitalmente em 16/12/2025 08:53)
FILIPE BRAIDA DO CARMO
PROFESSOR DO MAGISTERIO SUPERIOR
DeptCC/IM (12.28.01.00.00.83)
Matrícula: ###295#4

(Assinado digitalmente em 15/12/2025 23:57)
VICTOR REZENDE DE LIMA
DISCENTE
Matrícula: 2020######2

*Visualize o documento original em https://sipac.ufrrj.br/documentos/ informando seu número: **33250**, ano: **2025**, tipo: **DOCUMENTOS COMPROBATÓRIOS**, data de emissão: **15/12/2025** e o código de verificação: **e1a1eb851c***

Agradecimentos

Agradeço, primeiramente, a Deus que em Sua infinita misericórdia me sustentou e fortaleceu em cada etapa desta caminhada. Cada conquista alcançada e cada obstáculo superado foram frutos de Sua ação e providência em minha vida. Toda honra e glória seja dada a Deus.

Agradeço a minha amada esposa, Jéssica, pelo apoio incondicional, pelo companheirismo e, acima de tudo, pela paciência durante as longas horas de estudo e dedicação que esta graduação exigiu. Você foi fundamental para que eu chegasse até aqui.

Agradeço a minha filha, Mariana que mesmo com pouca idade foi minha maior fonte de inspiração. Seu sorriso puro e sua alegria renovaram minhas forças nos dias difíceis e me motivaram diariamente a concluir esta jornada, buscando sempre ser um exemplo para você.

Agradeço aos meus pais, Marcelo e Adriana, por tudo que fizeram por mim, não há palavras suficientes para expressar minha gratidão por cada conselho, correção e pelo suporte incondicional que moldaram meu caráter e meus valores. O sacrifício de vocês foram o alicerce que me permitiu sonhar alto e ter a segurança necessária para perseguir meus objetivos.

Agradeço a todos os professores do Departamento de Ciência da Computação da UFRRJ pelo conhecimento e suporte ao longo da graduação. Em especial, ao Professor Filipe Braida, pela orientação neste trabalho e pela excelência e dedicação demonstradas durante todo o curso, fundamentais para minha formação acadêmica e profissional.

E por fim agradeço aos amigos que a universidade me presenteou, pela parceria nos trabalhos, pelos estudos em grupo e pelos momentos de descontração que tornaram a graduação mais leve e memorável. Estendo também minha gratidão aos colegas e familiares que, direta ou indiretamente, contribuíram para a realização deste sonho.

RESUMO

Observator: Um Gateway para Observabilidade de LLM

Victor Rezende de Lima

Dezembro/2025

Orientador: Filipe Braida do Carmo, D.Sc.

A inteligência artificial generativa consolidou-se como um componente fundamental no desenvolvimento de software moderno, impulsionando uma rápida integração de capacidades cognitivas em aplicações comerciais através de *Application Programming Interfaces* (APIs). No entanto, essa dependência de modelos externos criou um novo paradigma operacional, onde o núcleo de inteligência das aplicações reside fora da infraestrutura controlada pelas organizações. Esse cenário resulta em um problema crítico de pouca visibilidade e falta de governança, caracterizado pela imprevisibilidade de custos gerados a partir do consumo de *tokens*, riscos de privacidade no tráfego de dados sensíveis e dificuldade de auditoria. Com o objetivo de mitigar esses riscos e retomar o controle infraestrutural, este trabalho propõe e implementa o Observator, um *gateway* de observabilidade de código aberto projetado para orquestrar, auditar e gerenciar as interações entre clientes e provedores de *Large Language Model* (LLM). A solução validou a eficácia do padrão arquitetural de AI Gateway com o padrão de projeto *adapter*, demonstrando-se uma alternativa viável às plataformas *Software as a Service* (SaaS) proprietárias existentes no mercado, ao oferecer soberania total dos dados e baixo atrito de integração sem os custos de licenciamento ou os riscos de privacidade inerentes a ecossistemas fechados.

ABSTRACT

Observator: Um Gateway para Observabilidade de LLM

Victor Rezende de Lima

Dezembro/2025

Advisor: Filipe Braida do Carmo, D.Sc.

Generative artificial intelligence has established itself as a fundamental component in modern software development, driving the rapid integration of cognitive capabilities into commercial applications via APIs. However, this reliance on external models has created a new operational paradigm where the application's intelligence core resides outside the infrastructure controlled by organizations. This scenario results in a critical problem of low visibility and lack of governance, characterized by the unpredictability of costs generated from token consumption, privacy risks in sensitive data traffic, and auditing difficulties. Aiming to mitigate these risks and regain infrastructural control, this work proposes and implements Observator, an open-source observability gateway designed to orchestrate, audit, and manage interactions between clients and LLM providers. The solution validated the effectiveness of the AI Gateway architectural pattern combined with the adapter design pattern, proving to be a viable alternative to the proprietary SaaS platforms existing in the market by offering full data sovereignty and low integration friction without the licensing costs or privacy risks inherent to closed ecosystems.

Lista de Figuras

Figura 3.1: Diagrama Entidade-Relacionamento do Sistema 39

Figura 4.1: Tela de Cadastro de Usuários 49

Figura 4.2: Tela de Login 50

Figura 4.3: Tela de Recuperação de Senha 51

Figura 4.4: Tela de Dashboard 52

Figura 4.5: Tela de Histórico de Interações 53

Figura 4.6: Tela de Detalhamento de Interação 54

Figura 4.7: Tela do Painel Administrativo 55

Figura 4.8: Tela de Registro Centralizado de Contas 55

Figura 4.9: Tela de Edição de Usuário 56

Figura 4.10: Tela de Configuração de Perfil 57

Figura 4.11: Tela de Alteração de Senha 57

Figura 4.12: Tela do Gerenciamento de Tokens de Acesso à API 58

Lista de Tabelas

3.1 Tabela Comparativa das Soluções Representativas de Observabilidade 23

3.2 Tabela de Requisitos Funcionais 28

3.3 Tabela de Regras de Negócio 31

Lista de Abreviaturas e Siglas

API	*Application Programming Interface*
BART	*Bidirectional and Auto-Regressive Transformers*
BERT	*Bidirectional Encoder Representations from Transformers*
BPE	*Byte Pair Encoding*
CSS	*Cascading Style Sheets*
GPU	*Graphics Processing Unit*
GPT	*Generative Pre-trained Transformer*
GUI	*Graphical User Interface*
HMR	*Hot Module Replacement*
HMM	*Hidden Markov Models*
HTTP	*Hypertext Transfer Protocol*
IA	Inteligência Artificial
JSON	*JavaScript Object Notation*
JSONB	*JSON Binary*
LLM	*Large Language Model*
LM	*Language Model*
LSTM	*Long Short-Term Memory*
MVC	*Model-View-Controller*
ORM	*Object-Relational Mapping*

PLN	Processamento de Linguagem Natural
RAG	*Retrieval-Augmented Generation*
RBAC	*Role-Based Access Control*
REST	*Representational State Transfer*
RF	Requisitos Funcionais
RNN	*Recurrent Neural Networks*
RN	Regras de Negócio
SaaS	*Software as a Service*
SDK	*Software Development Kit*
SGBD	Sistema Gerenciador de Banco de Dados
SPA	*Single-Page Application*
SQL	*Structured Query Language*
T5	*Text-to-Text Transfer Transformer*
TBT	*Time Between Tokens*
TPU	*Tensor Processing Unit*
TTFT	*Time To First Token*
TTLT	*Time To Last Token*
UI	*User Interface*
URL	*Uniform Resource Locator*

Sumário

Agradecimentos i

Resumo iii

Abstract iv

Lista de Figuras v

Lista de Tabelas vi

Lista de Abreviaturas e Siglas vii

1 Introdução 1

2 Fundamentação 3

2.1 *Large Language Model* 4

2.1.1 Das Abordagens Estatísticas à Arquitetura *Transformer* . . . 5

2.2 Arquiteturas de Acesso a LLM 7

2.2.1 SaaS 8

2.2.1.1 Interação Humano-Modelo 8

2.2.1.2 Interação Sistema-Modelo 9

2.2.2 Fluxos de Geração 9
2.2.2.1 Geração Síncrona 9
2.2.2.2 Geração por *Streaming* 10
2.2.2.3 Embeddings 10
2.2.2.4 RAG 10
2.3 Engenharia de *Prompt* e Observabilidade de LLM 11
2.3.1 Engenharia de *Prompt* 12
2.3.2 Observabilidade de LLM 14
2.3.2.1 Observabilidade de Custo 14
2.3.2.2 Observabilidade de Qualidade 15
2.3.2.3 Observabilidade de Latência 15

3 Proposta **17**

3.1 Motivação 17
3.2 Trabalhos Relacionados 19
3.2.1 LangSmith 20
3.2.2 Langfuse 21
3.2.3 Helicone 21
3.2.4 Phoenix 22
3.2.5 Síntese Comparativa 23
3.3 Proposta de Sistema 24
3.3.1 Módulos do Sistema 24
3.3.1.1 Autenticação e Autorização 25

3.3.1.2 Gateway de LLM 25
3.3.1.3 Auditoria e Engenharia de Custos 26
3.3.1.4 *Dashboard* 27
3.3.2 Requisitos Funcionais 27
3.3.3 Regras de Negócio 30
3.3.4 Casos de Uso 33
3.3.5 Modelagem de Dados 38

4 Observator **41**

4.1 Tecnologias Utilizadas 41
4.1.1 AdonisJS 42
4.1.1.1 Lucid 42
4.1.1.2 Auth 43
4.1.1.3 Bouncer 44
4.1.1.4 Adonis Jobs 44
4.1.2 Padrão Arquitetural: AI Gateway 46
4.1.3 Arquitetura de Projeto e *Front-end* 46
4.2 Implementação do Observator 48
4.2.1 Registro, Autenticação e Recuperação 48
4.2.2 Dashboard 51
4.2.3 Histórico de Interações 52
4.2.4 Detalhes da Interação 53
4.2.5 Painel de Governança de Usuários 54

4.2.6 Gestão de Conta e Configurações 56

4.2.6.1 Perfil e Identidade Visual 56

4.2.6.2 Segurança e Credenciais 57

4.2.6.3 Gestão de Tokens 58

5 Conclusão **60**

5.1 Considerações Finais 60

5.2 Limitações e Trabalhos Futuros 61

5.2.1 Limitações Atuais 61

5.2.2 Trabalhos Futuros 62

Referências **64**

Capítulo 1

Introdução

A última década marcou uma transformação singular na indústria de tecnologia, impulsionada pela consolidação da Inteligência Artificial (IA) como eixo central da inovação. O surgimento e a rápida evolução dos LLMs, aliados à popularização da IA generativa, catalisada pelo lançamento de plataformas como o ChatGPT[1], superaram barreiras técnicas historicamente desafiadoras. Esses avanços permitiram que sistemas computacionais passassem a processar, interpretar e gerar linguagem natural com níveis inéditos de fluência, robustez e eficiência (MASLEJ et al., 2023) . A disponibilização desses modelos por meio de APIs ou fazendo uso de plataformas SaaS democratizou o acesso a essa tecnologia, intensificando a corrida pela integração de capacidades cognitivas em softwares comerciais.

No entanto, a integração com LLMs introduziu um impacto operacional significativa, ao consumir esses modelos como serviços em nuvem, as equipes de engenharia passaram a operar com visibilidade limitada sobre os custos gerados por essa integração (DIAZ-DE-ARCAYA et al., 2024). Os provedores de LLMs não disponibilizam informações detalhadas sobre as interações com softwares, o que acrescenta novos riscos à engenharia de software, como a imprevisibilidade orçamentária decorrente da cobrança por *tokens*, a dificuldade em diagnosticar falhas semânticas e a ausência de governança sobre os dados enviados (LIAO; VAUGHAN, 2023). Além disso, ferramentas tradicionais de monitoramento, projetadas para sistemas determinísticos,

[1] *https* : *//chatgpt.com*

mostram-se insuficientes para lidar com a natureza probabilística e financeiramente sensível dessas novas operações (DIAZ-DE-ARCAYA et al., 2024).

Diante desse cenário, torna-se imperativo o desenvolvimento de mecanismos que devolvam às equipes de engenharia o controle sobre sua infraestrutura de inteligência. As soluções existentes no mercado frequentemente impõem dilemas entre simplicidade e soberania, exigindo ou o envio de dados sensíveis para plataformas proprietárias ou a configuração de infraestruturas complexas e onerosas. Há, portanto, uma lacuna para ferramentas que ofereçam transparência operacional sem sacrificar a privacidade dos dados ou a agilidade do desenvolvimento.

Nesse contexto, este trabalho propõe o projeto e a implementação do Observator, uma plataforma de observabilidade e governança baseada em arquitetura de *gateway*. O objetivo central é construir uma solução de código aberto capaz de interceptar, auditar e gerenciar as interações entre aplicações clientes e provedores de LLM. A proposta busca viabilizar a integração transparente com aplicações existentes por meio da compatibilidade com *Software Development Kit* (SDK)s oficiais, minimizando o atrito de adoção.

O presente trabalho está organizado em cinco capítulos. O Capítulo 2 apresenta a fundamentação teórica sobre LLMs, iniciando pela definição conceitual desses modelos, passando pelas formas de acesso e consumo, e culminando na discussão sobre engenharia de observabilidade aplicada a LLMs. O Capítulo 3 discute a motivação, analisa os trabalhos relacionados e apresenta a proposta de solução. O Capítulo 4 descreve as tecnologias utilizadas e detalha o processo de implementação da plataforma. Por fim, o Capítulo 5 apresenta as considerações finais, destacando as limitações do trabalho e sugerindo direções para investigações futuras.

Capítulo 2

Fundamentação

Os avanços recentes em LLMs redefiniram o campo do Processamento de Linguagem Natural (PLN), estabelecendo novos paradigmas de desempenho, generalização e aplicabilidade (BOMMASANI et al., 2021). Esses modelos, conhecidos como LLMs, incorporam arquiteturas de aprendizado profundo capazes de analisar e gerar texto com elevado grau de coerência contextual, fluidez linguística e adaptação a múltiplas tarefas (ZHOU et al., 2023).

Entretanto, a compreensão dos LLMs exige mais do que a observação de seu comportamento atual, implicando em revisitar a trajetória histórica que levou à construção de arquiteturas cada vez mais expressivas e eficientes, passando dos modelos estatísticos tradicionais aos mecanismos que fundamentam a era da arquitetura *transformer* (JURAFSKY; MARTIN, 2025). Além disso, compreender como esses sistemas são disponibilizados ao público, seja por meio de interfaces diretas ou por integrações via API, permite contextualizar suas limitações, capacidades e implicações para o desenvolvimento de soluções baseadas em IA.

Assim, este capítulo tem como objetivo oferecer uma visão abrangente da evolução conceitual e arquitetural que culminou nos LLMs, bem como dos modos de acesso e operação desses sistemas. Tal panorama estabelece as bases necessárias para discutir, nos capítulos subsequentes, os desafios práticos enfrentados no uso profissional dessas tecnologias.

2.1 *Large Language Model*

Antes de delimitar a especificidade dos grandes modelos, é necessário definir o conceito de *Language Model* (LM). Em sua essência, um *Language Model* é um sistema probabilístico treinado para determinar a verossimilhança de sequências de palavras. O objetivo primordial desses modelos é prever o próximo elemento de uma sentença com base no histórico do texto, capturando as regularidades estatísticas, gramaticais e semânticas da língua alvo (JURAFSKY; MARTIN, 2025).

Os LLMs constituem uma evolução dessa premissa, configurando-se como sistemas avançados de PLN fundamentados na tarefa de geração condicional, na qual o modelo mapeia contextos complexos para respostas textuais coerentes. Segundo Jurafsky e Martin (2025), LLMs tratam-se de modelos probabilísticos treinados em vastos volumes de dados para aprender a distribuição estatística da linguagem. A operacionalização desses sistemas ocorre através da predição iterativa de *tokens*, que são as unidades atômicas discretas que compõem o vocabulário do modelo, cuja probabilidade de ocorrência é calculada condicionalmente a partir de uma instrução ou contexto de entrada fornecido pelo usuário, tecnicamente denominado *prompt*.

Enquanto para Bommasani et al. (2021), estes sistemas enquadram-se na definição de modelos fundacionais, e que podem ser definidos como modelos treinados em escala massiva sobre dados abrangentes e que podem ser adaptados para uma vasta gama de tarefas subsequentes. Essa característica permite que uma única arquitetura generalize o aprendizado para funções distintas, como tradução, sumarização e geração de código, eliminando a necessidade de desenvolver modelos especializados do zero para cada aplicação

Porém, o alcance desse patamar de generalização e escala reflete uma evolução não linear, impulsionada pela necessidade de superar restrições computacionais e dificuldades na modelagem de contexto. Para dimensionar a transformação radical representada pelos modelos atuais, faz-se necessário examinar as arquiteturas precursoras, compreendendo como a busca pela captura eficiente de dependências de longo alcance motivou a migração dos métodos estatísticos para as abordagens de

aprendizado profundo.

2.1.1 Das Abordagens Estatísticas à Arquitetura *Transformer*

Historicamente, antes da predominância do aprendizado profundo, o PLN foi regido por paradigmas simbólicos e estatísticos que não utilizavam redes neurais. Essa era foi marcada pelo uso de modelos de *n-gram*, que operavam baseando-se estritamente na contagem de frequência de palavras em janelas curtas de contexto para estimar qual palavra deveria vir a seguir (JURAFSKY; MARTIN, 2025). Outras abordagens clássicas, como os *Hidden Markov Models* (HMMs), foram fundamentais para tarefas como reconhecimento de fala, baseando-se em tabelas de probabilidade de transição entre estados. No entanto, essas técnicas dependiam de probabilidades fixas calculadas sobre o texto de treinamento e sofriam com a escassez de dados ao tentar lidar com sequências de palavras nunca vistas antes (JURAFSKY; MARTIN, 2025).

Posteriormente, a introdução de *Recurrent Neural Networks* (RNN) possibilitou modelar dependências temporais e suas variantes, em especial *Long Short-Term Memory* (LSTM), proposta por Hochreiter e Schmidhuber (1997), mitigou problemas de gradiente e ampliou a capacidade de capturar dependências de maior alcance, embora ambas as classes de modelo ainda sofressem limitações de paralelização devido ao processamento sequencial.

Para mitigar a rigidez dos modelos estatísticos baseados em símbolos discretos, houve a adoção de representações distribuídas, técnicas como word2vec de Mikolov et al. (2013) e GloVe de Pennington, Socher e Manning (2014) passaram a codificar relações semânticas em vetores densos. Isso permitiu uma generalização semântica mais robusta, embora ainda dependesse de arquiteturas sequenciais para processar o contexto.

Um marco importante nessa trajetória foi a criação do IBM Watson[1], segundo Ferrucci et al. (2010) ele combinava PLN, recuperação de informação e análise probabilística em domínio aberto. Em 2011, o IBM Watson venceu competidores

[1] *https* : *//www.ibm.com/watson*

humanos no programa televisivo Jeopardy![2], um quiz show de perguntas e respostas. Embora Watson não fosse um modelo generativo, seu sucesso demonstrou a viabilidade de sistemas capazes de interpretar perguntas complexas em linguagem natural.

Outro marco significativo ocorreu com AlphaGo[3] em 2016, que venceu uma partida do jogo de tabuleiro Go contra o campeão mundial Lee Se-dol[4] evidenciou o poder do *Deep Reinforcement Learning* combinado a redes neurais profundas e infraestruturas de alto desempenho. Embora AlphaGo não seja um sistema de PLN, seu impacto demonstrou o potencial de redes profundas e de hardware massivamente paralelo como *Graphics Processing Unit* (GPU) e *Tensor Processing Unit* (TPU) (SILVER et al., 2016).

A transformação arquitetural decisiva ocorreu em 2017 com o artigo Attention Is All You Need, de Vaswani et al. (2017), que introduziu a arquitetura *Transformer* e o mecanismo de *self-attention*, permitindo processamento paralelo eficiente e a captura de dependências de longo alcance sem recorrência. As principais vantagens desta arquitetura residem em sua capacidade de paralelização massiva, o que viabiliza o treinamento de modelos gigantescos com alta eficiência em GPU e TPU. Além disso, a arquitetura destaca-se pela modelagem eficiente de contextos de longo alcance e por sua modularidade arquitetural, característica que permite a criação de variantes flexíveis, como *encoder-only*, *decoder-only* e *encoder-decoder*.

A partir dos *Transformers*, sucederam-se marcos que definiram a era dos LLM, transformando esses desenvolvimentos em infraestrutura cognitiva reutilizável para diversas aplicações, o que Bommasani et al. (2021) caracteriza como modelos fundacionais. O ano de 2018 foi crucial para essa consolidação: o *Generative Pre-trained Transformer* (GPT) inaugurou o paradigma de pré-treinamento em grandes conjuntos de dados seguido de *fine-tuning* supervisionado (RADFORD et al., 2018), enquanto o *Bidirectional Encoder Representations from Transformers* (BERT) revolucionou a compreensão textual ao introduzir o mascaramento de *tokens* e arquiteturas bidirecionais (DAVIS, 2018). Na sequência, o lançamento do GPT-2 em 2019 demonstrou,

[2] $https://www.ibm.com/history/watson-jeopardy$
[3] $https://deepmind.google/research/alphago/$
[4] $https://www.bbc.com/news/technology-35785875$

pela primeira vez, a capacidade de geração coerente em larga escala, incitando debates iniciais sobre segurança e riscos de modelos generativos (RADFORD et al., 2019).

A evolução arquitetural prosseguiu com propostas de unificação e generalização. Modelos como o *Text-to-Text Transfer Transformer* (T5) estabeleceram que todas as tarefas de PLN poderiam ser tratadas como problemas de transformação de texto para texto (RAFFEL et al., 2020), enquanto o *Bidirectional and Auto-Regressive Transformers* (BART) combinou os princípios de *encoder-decoder* para unificar reconhecimento e geração em uma só arquitetura (LEWIS et al., 2020). O ápice dessa fase ocorreu em 2020 com o GPT-3, que introduziu o conceito de *in-context learning*, permitindo que o modelo aprendesse novas tarefas apenas através de exemplos fornecidos no próprio *prompt*, eliminando a necessidade de *fine-tuning* adicional para diversos casos de uso (BROWN et al., 2020).

2.2 Arquiteturas de Acesso a LLM

A crescente complexidade computacional dos modelos fundacionais e, em especial, dos modelos de fronteira, define um cenário em que apenas grandes organizações dispõem de recursos suficientes para treinar e executar tais modelos localmente. Os modelos de fronteira representam as instâncias mais avançadas desses sistemas, alcançando o nível mais elevado de desempenho multitarefa, capacidade de generalização e eficiência no processamento de grandes quantidades de informação.

O treinamento e a manutenção operacional de modelos, como o GPT-5.1[5] ou o Gemini 2.5 Pro[6] , dependem de infraestrutura massivamente paralela, baseada em *clusters* de GPUs e TPUs, além de investimentos financeiros extremamente elevados, inacessíveis para a maior parte das organizações. Strubell, Ganesh e McCallum (2019) destacam que tanto o custo energético quanto a exigência de hardware especializado para sustentar modelos dessa magnitude criam barreiras financeiras significativas, ultrapassando as capacidades típicas de empresas de médio porte e centros de pesquisa independentes.

[5]https://platform.openai.com/docs/guides/latest-model
[6]https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro?hl=pt-br

2.2.1 SaaS

Diante dessa barreira econômica e técnica, o modelo de distribuição de software SaaS consolidou-se como o principal mecanismo de disponibilização desses sistemas ao público. Nesse paradigma, os provedores hospedam os modelos em infraestrutura de nuvem e os disponibilizam por meio de interfaces acessíveis via internet, geralmente mediante assinatura ou cobrança por uso *token*.

Um *token* representa o menor elemento linguístico processado pelo modelo, podendo corresponder a palavras inteiras, subpalavras ou fragmentos de caracteres. Em modelos modernos a contagem de *tokens* é realizada usando o algoritmo*Byte Pair Encoding* (BPE), que equilibra eficiência computacional e preservação semântica (SENNRICH; HADDOW; BIRCH, 2016).

Choudhary (2007) observa que o SaaS reduz a necessidade de instalação, manutenção e escalabilidade local, democratizando o acesso a serviços altamente especializados. A partir desse modelo de distribuição, consolidaram-se dois modos principais de acesso a esses sistemas, diferenciados fundamentalmente pelo ator que inicia e gerencia a requisição.

2.2.1.1 Interação Humano-Modelo

Esta modalidade refere-se ao uso direto dos modelos por usuários finais através das aplicações oficiais mantidas por provedores, como ChatGPT[7] e Gemini[8]. Nesse caso, a comunicação é mediada por uma *Graphical User Interface* (GUI), dispensando configurações técnicas complexas.

Todo o fluxo de interação, da construção da mensagem ao recebimento da resposta, é abstraído pela aplicação que gerencia o estado da conversa e a formatação visual. O objetivo aqui é oferecer uma experiência de uso acessível, intuitiva e exploratória, onde a latência e a precisão são percebidas subjetivamente pelo usuário humano.

[7] *https://chatgpt.com/*
[8] *https://gemini.google.com/app*

2.2.1.2 Interação Sistema-Modelo

A interação sistema-modelo corresponde ao consumo do LLM como um componente de *software* integrado a aplicações de terceiros via API. Nesse cenário, o LLM não responde a um humano diretamente, mas atua como um módulo de processamento dentro da arquitetura de um sistema cliente.

Diferentemente da interação humano-modelo, essa abordagem possibilita automação em larga escala, integração com fluxos de trabalho corporativos e controle arquitetural rigoroso. É o padrão utilizado para construir assistentes virtuais customizados, analisadores de documentos e agentes autônomos.

2.2.2 Fluxos de Geração

As APIs utilizadas para acessar LLMs não constituem sistemas monolíticos. Pelo contrário, são compostas por múltiplos *endpoints*, cada um projetado para atender diferentes demandas de interação, desempenho e contexto de uso. Esses padrões de comunicação estruturam como o *back-end* cliente realiza requisições e como o servidor do LLM devolve as respostas. Para os propósitos deste trabalho, destacam-se quatro fluxos principais: Geração Síncrona, Geração por Streaming, Embeddings e Arquitetura *Retrieval-Augmented Generation* (RAG).

2.2.2.1 Geração Síncrona

O fluxo de geração síncrona segue o paradigma clássico da web baseado no modelo *Representational State Transfer* (REST), nesse formato o *back-end* cliente envia uma requisição *Hypertext Transfer Protocol* (HTTP) contendo o *prompt* completo. A API do LLM processa a solicitação e mantém a conexão aberta até concluir totalmente a geração do texto, retornando o resultado em um único bloco. Trata-se de um fluxo simples, amplamente compatível com arquiteturas já existentes e adequado para operações não interativas. Entretanto, como todo o processamento ocorre antes da devolução da resposta, o usuário final pode experimentar alta latência perceptível em *prompts* mais complexos (XIAO; YANG, 2025).

2.2.2.2 Geração por Streaming

Para cenários que exigem maior interatividade, como *chatbots*, sistemas de atendimento ou aplicações que precisam apresentar respostas progressivamente, as API oferecem o modo *streaming*. Nesse padrão, o servidor do LLM envia partes da resposta, muitas vezes *token* a *token*, à medida que a geração ocorre esse mecanismo é viabilizado por recursos do protocolo HTTP. O *back-end* cliente pode então retransmitir imediatamente esses fragmentos ao *fron-tend*, proporcionando sensação de fluidez e reduzindo a latência percebida pelo usuário (XIAO; YANG, 2025).

2.2.2.3 Embeddings

Diferentemente dos fluxos de geração textual, o *endpoint* de *embeddings* retorna uma representação matemática do texto. *Embeddings* são vetores densos de alta dimensionalidade que capturam propriedades semânticas, permitindo medir similaridade e relações contextuais entre trechos de texto (JURAFSKY; MARTIN, 2025). Nesse fluxo, o *back-end* cliente envia uma entrada textual e recebe o vetor correspondente, que pode ser usado em aplicações como mecanismos de busca semântica, recomendação, categorização e detecção de similaridade contextual.

2.2.2.4 RAG

A abordagem RAG não corresponde a um *endpoint* específico, mas sim a um padrão arquitetural que combina geração de texto com recuperação de conhecimento externo, buscando superar limitações de memória e atualização dos LLM (GAO et al., 2024). O padrão RAG é especialmente importante para aplicações que dependem de dados privados, informação atualizada ou conteúdos altamente especializados.

O fluxo típico de uma abordagem RAG é gerenciado por um componente orquestrador e envolve: (i) o envio do *prompt* inicial ao *endpoint* de *embeddings* para sua conversão em um vetor de busca; (ii) a consulta desse vetor a uma base externa, geralmente um banco vetorial, para identificar documentos semanticamente relevantes; (iii) a seleção e incorporação dos documentos recuperados como contexto

adicional ao *prompt*; (iv) o envio do *prompt* enriquecido ao *endpoint* de geração, de forma síncrona ou em *streaming*, permitindo respostas mais precisas, atualizadas e contextualizadas.

A compreensão dos modelos de fronteira, das restrições infraestruturais que definem seu uso e dos diferentes modos de interação oferecidos pelos provedores estabelece a base necessária para analisar como sistemas clientes podem explorar essas capacidades de forma eficiente. Seja por meio de interfaces humanas ou integrações automatizadas via API, o comportamento dos LLMs depende diretamente da estrutura do *prompt*, da escolha do fluxo de geração e do gerenciamento adequado do contexto. Esses elementos, embora muitas vezes abstraídos no uso cotidiano, tornam-se cruciais quando a aplicação exige previsibilidade, controle e desempenho consistente.

Nesse cenário, torna-se evidente que o simples acesso ao modelo não é suficiente para garantir resultados de qualidade ou estabilidade operacional. O desenvolvimento de soluções apoiadas em LLMs demanda práticas rigorosas de engenharia de *prompts*, mecanismos de monitoramento contínuo e estratégias de observabilidade capazes de capturar métricas, falhas e variações de comportamento ao longo do tempo.

Esses fatores levam à próxima seção, que aprofunda as metodologias de construção, avaliação e acompanhamento de interações com modelos generativos, evidenciando como tais práticas são essenciais para o uso confiável e sustentável dessas tecnologias em ambientes de produção.

2.3 Engenharia de *Prompt* e Observabilidade de LLM

Uma vez estabelecida na seção anterior a interação sistema-modelo, esta seção aprofunda-se nos desafios operacionais e nos novos paradigmas de gerenciamento que emergem dessa interação. Diferentemente de sistemas de *software* tradicionais, cujo comportamento é deterministicamente definido por regras explícitas, métodos formais e instruções rígidas, a interação com LLMs possui natureza probabilística, contextual e dependente de linguagem natural. Essa característica modifica de ma-

neira significativa os mecanismos tradicionais de controle, depuração e previsibilidade do *software*.

Nesse novo cenário, duas disciplinas tornam-se centrais: Engenharia de Prompt e Observabilidade de LLM. A primeira atua como mecanismo primário de controle qualitativo e funcional sobre o comportamento do modelo; a segunda fornece os meios quantitativos e analíticos para aferir, monitorar e otimizar esse comportamento em ambientes produtivos.

2.3.1 Engenharia de *Prompt*

A Engenharia de *Prompt* é a disciplina dedicada ao projeto, construção e otimização das entradas textuais destinadas a orientar um modelo fundacional a produzir respostas alinhadas ao objetivo do usuário (LIU et al., 2023b).

Em sistemas baseados em LLMs, o *prompt* funciona como uma interface de programação conceitual, na qual o desenvolvedor especifica comportamentos, restrições, estilo e contexto por meio de linguagem natural. A estrutura, a clareza e o grau de contexto incluído no *prompt* influenciam diretamente a qualidade, a precisão e o custo operacional associado à geração do modelo. O desempenho do modelo é altamente sensível à formulação da entrada, o que motivou o surgimento de técnicas formais de engenharia de *prompt* (ZHOU et al., 2023) . Entre as principais abordagens, destacam-se:

- Instrução Direta: O modelo é instruído a realizar uma tarefa sem nenhum exemplo prévio como um *prompt*, *e.g.*, “Traduza este texto para o francês: [...]” (LIU; NEUBIG; ANDREAS, 2024).
- Instrução com Exemplos: O modelo recebe, no próprio *prompt*, uma pequena quantidade de exemplos de entrada e saída, *e.g.*, “Q: ’maçã’ é ’apple’. Q: ’uva’ é [...]”. Esta técnica aumenta significativamente a performance em tarefas específicas (LIU; NEUBIG; ANDREAS, 2024).
- Indução a Cadeia de Pensamento: Consiste em instruir o modelo a “pensar passo a passo” ou “explicar seu raciocínio” antes de dar a resposta final. Wei et

al. (2022) demonstram que esta simples adição ao *prompt* melhora significativamente a capacidade do modelo em tarefas de raciocínio complexo, como problemas matemáticos ou lógicos.

- Padrões de *Prompt* Avançados: Técnicas mais recentes, como ReAct, combinam a Indução a Cadeia de Pensamento com o uso de ferramentas, permitindo ao LLM decidir quando consultar uma API externa como parte de seu processo de pensamento para formular uma resposta (YAO et al., 2022).

Um ponto essencial da engenharia de *prompt* é que ela não se limita a fazer a pergunta certa, mas constitui um processo iterativo de design, cujo impacto é simultaneamente técnico e econômico. O conteúdo do *prompt* determina a quantidade de *tokens* de entrada, a extensão e complexidade da resposta, a latência da geração de resposta e o custo financeiro da requisição

Prompts baseados na técnica de Indução a Cadeia de Pensamento tendem a produzir respostas significativamente mais longas, ampliando o custo de saída. De modo semelhante, fluxos baseados em RAG formam *prompts* compostos, integrando instruções, pergunta do usuário e documentos recuperados, o que expande substancialmente o número de *tokens* de entrada. Assim, cada decisão de design de *prompt* envolve *trade-offs* explícitos entre clareza, precisão, latência e custo. O *prompt*, portanto, constitui a principal alavanca de controle disponível ao desenvolvedor para regular o comportamento do modelo. Essa relação direta entre design, qualidade e custo cria a necessidade de uma disciplina complementar: a observabilidade, que permite mensurar, diagnosticar e otimizar sistematicamente os efeitos das estratégias de engenharia de *prompt* em ambientes reais.

A medida que as técnicas de engenharia de *prompt* evoluem de instruções simples para estruturas complexas como Indução a Cadeia de Pensamento e ReAct, observa-se um aumento proporcional na imprevisibilidade do comportamento do sistema. O *prompt* deixa de ser uma variável estática para se tornar um componente dinâmico que impacta diretamente o consumo de recursos e a latência da aplicação. Essa correlação direta entre a sofisticação do design do *prompt* e seus custos operacionais evidencia que o controle qualitativo, por si só, é insuficiente. Torna-se imperativo, portanto,

estabelecer mecanismos quantitativos rigorosos para monitorar essas interações, dando origem à disciplina de Observabilidade de LLM.

2.3.2 Observabilidade de LLM

O conceito de observabilidade tem origem na teoria de controle de Kálmán (1960), mas ganhou relevância contemporânea na engenharia de *software* e em práticas DevOps, onde é definido como a capacidade de inferir o estado interno de um sistema a partir de suas saídas externas. Em sistemas tradicionais, essa observabilidade é estruturada nos chamados três pilares: métricas, *logs* e *traces* (SRIDHARAN, 2018).

Entretanto, quando aplicada a sistemas baseados em LLM, a observabilidade assume uma complexidade inteiramente nova. As ferramentas tradicionais de observabilidade não foram projetadas para lidar com as características probabilísticas, semânticas e financeiras que emergem do uso de APIs de LLMs. Esse novo cenário exige a ampliação da observabilidade para três dimensões adicionais: custo, qualidade semântica e latência (HUYEN, 2022).

2.3.2.1 Observabilidade de Custo

Em sistemas tradicionais, o custo de uma chamada de API é computacionalmente marginal e faturado de forma agregada, *e.g.*, custo por hora do servidor. Em SaaS integrados a LLM, o custo financeiro é variável e direto, faturado por chamada com base nos *tokens* processados.

A observabilidade de custo exige que o sistema registre a quantidade de *tokens* no *prompt* e a quantidade de *tokens* na resposta para cada interação, pois o custo é determinado com base no consumo total *tokens*. O mapeamento de custos deve ser realizado multiplicando o total de *tokens* pelo preço específico do modelo, dessa forma convertendo dados de uso técnico em valores financeiros (VELASCO; TSIRTSIS; GOMEZ-RODRIGUEZ, 2025; SUN et al., 2025).

2.3.2.2 Observabilidade de Qualidade

Um sistema de *software* tradicional falha de forma binária e explícita retornando o código HTTP de *status* de falha ou sucesso. Uma API de LLM, por outro lado, pode ser sintaticamente bem-sucedida, mas semanticamente falha. Para Ji et al. (2023) a falha semântica mais comum é a "alucinação", onde o modelo gera informações factualmente incorretas, mas de forma plausível e confiante. Para mitigar esses riscos, Liang et al. (2022), Chang et al. (2023) sugerem métricas de avaliação holística e *feedback* humano.

A observabilidade de qualidade deve, portanto, ir além dos códigos de status HTTP, buscando armazenar a carga útil do *prompt* e resposta de cada interação para análises posteriores, correlacionar a avaliação do usuário a cada interação além de monitorar o tráfego em busca de dados sensíveis, toxicidade ou falhas em seguir as instruções do sistema.

2.3.2.3 Observabilidade de Latência

A latência em uma API de LLM é inerentemente superior e mais volátil do que em uma API tradicional, sendo influenciada por fatores dinâmicos como o tamanho do *prompt*, a complexidade instrucional e a quantidade de *tokens* gerados. Essa característica exige uma mudança de paradigma na medição de performance, priorizando a percepção do usuário em detrimento apenas do tempo total de execução. Para compreender o impacto real na experiência do usuário, a medição de performance é decomposta em três métricas (GOEL et al., 2025).

A *Time To Last Token* (TTLT) corresponde ao tempo total decorrido entre o envio do *prompt* e o recebimento do último caractere da resposta. É a métrica padrão para chamadas síncronas, onde o sistema aguarda a conclusão total do processamento. No entanto, como o TTLT é proporcional ao tamanho da resposta gerada, ele pode variar drasticamente sem indicar degradação na saúde do servidor (GOEL et al., 2025).

Outra métrica importante é a *Time To First Token* (TTFT) que representa o

intervalo de tempo que o usuário aguarda até visualizar o primeiro fragmento de conteúdo. Em interfaces conversacionais, esta é a métrica crítica de percepção de velocidade. Um TTFT baixo cria uma sensação de resposta imediata, influenciando o usuário a cerca da capacidade de resposta do LLM (GOEL et al., 2025).

Uma terceira métrica auxilia na avaliação de latência, conhecida como *Time Between Tokens* (TBT) ela mede o intervalo entre a geração de *tokens* de saída consecutivos de uma solicitação e afeta a fluidez percebida da resposta, o que é particularmente importante para aplicativos interativos, nos quais os usuários esperam um fluxo contínuo e ininterrupto de conteúdo gerado (GOEL et al., 2025).

A implementação eficiente desse sistema de observabilidade multifacetada exige um ponto centralizado de coleta, agregação e análise de dados. Delegar essa responsabilidade de forma distribuída entre microsserviços ou equipes de desenvolvimento distintas não apenas introduziria redundâncias e inconsistências, como também ampliaria o risco operacional ao dificultar auditorias, diagnósticos e o monitoramento contínuo da qualidade das respostas geradas. Em um cenário no qual os LLMs operam como componentes críticos de aplicações sensíveis, a ausência de um mecanismo unificado de visão e controle compromete diretamente a confiabilidade do sistema como um todo.

Essa constatação evidencia um desafio estratégico que embora o ecossistema atual ofereça ferramentas consolidadas, ele carece de soluções que conciliem simplicidade operacional, autonomia organizacional e suporte nativo às particularidades do uso de modelos de linguagem em produção. É justamente nessa lacuna que este trabalho se insere.

No capítulo seguinte, é apresentada a proposta de solução, detalhando como uma plataforma de observabilidade especializada para LLMs pode unificar métricas, registros e fluxos de análise, fornecendo uma camada essencial de governança, rastreabilidade e confiabilidade para aplicações que dependem desses modelos.

Capítulo 3

Proposta

Com base na fundamentação teórica apresentada no Capítulo 2, que estabeleceu os conceitos de LLMs e a necessidade crítica de observabilidade, este capítulo detalha a proposta da solução desenvolvida neste trabalho. O objetivo central é especificar uma plataforma de *gateway* de observabilidade capaz de interceptar, auditar e gerenciar as interações entre aplicações clientes e provedores de IA, visando mitigar os desafios operacionais de baixa visibilidade, imprevisibilidade de custos e latência através de uma arquitetura que prioriza a soberania dos dados e a simplicidade de integração.

A estrutura do capítulo organiza-se de forma progressiva, iniciando pela análise das motivações práticas e de ferramentas de observabilidade de LLMs, para identificar as lacunas que justificam o desenvolvimento da ferramenta. Na sequência, define-se a arquitetura modular da solução, culminando na formalização da especificação técnica através dos requisitos funcionais, regras de negócio, casos de uso e modelagem de dados, que estabelecem o contrato de comportamento esperado para a implementação descrita no capítulo seguinte.

3.1 Motivação

Os LLMs transcenderam o ambiente acadêmico para se tornar um motor de inovações disruptivas na indústria. No entanto, a consolidação dos LLM como

ferramenta de uso comercial em massa, catalisada pelo lançamento de ferramentas como o ChatGPT[1] em 2022, inaugurou um novo paradigma de engenharia de software (RIO-CHANONA; LAURENTSYEVA; WACHS, 2024). Empresas de todos os portes, desde *startups* até gigantes tecnológicas, aceleraram a integração desses modelos em produtos que variam de copilotos de produtividade a sistemas de decisão complexos.

Organizações que integram LLM a seus softwares enfrentam um cenário de "caixa-preta", onde as interações com as APIs de LLMs são fechadas e difíceis de monitorar, uma vez que a natureza proprietária dos modelos impede o acesso ao seu funcionamento interno, comprometendo desta forma a auditoria dessa integração (LIAO; VAUGHAN, 2023). Os desafios decorrentes dessa falta de visibilidade manifestam-se em diversas frentes críticas.

A dependência de API de terceiros, cobradas por *token*, introduz uma variabilidade financeira inexistente em softwares determinísticos tradicionais. Sem uma visibilidade granular do consumo de recursos por requisição, as organizações perdem a capacidade de otimizar *prompts* ou alocar orçamentos de forma eficiente, resultando em despesas imprevistas que podem comprometer a viabilidade econômica de projetos (DIAZ-DE-ARCAYA et al., 2024).

Não menos crítica é a questão da confiabilidade e da qualidade semântica, LLMs são mecanismos probabilísticos sem compreensão do mundo real, propensos a "alucinações" que consiste na geração de informações factualmente incorretas com alta convicção (JI et al., 2023). Adicionalmente, Bender et al. (2021) argumentam que esses modelos operam como "papagaios estocásticos", copiando padrões linguísticos e replicando vieses sociais presentes nos dados de treinamento. Essa tendência de gerar conteúdo enviesado ou incorreto representa um risco de reputação e ético severo para aplicações corporativas.

No âmbito da proteção de dados, emergem vulnerabilidades críticas de segurança e privacidade, uma vez que os modelos podem inadvertidamente revelar informações sensíveis ou facilitar ataques cibernéticos (LIAO; VAUGHAN, 2023). A arquitetura de interação dos LLM expõe novas superfícies de ataque, como a injeção de

[1] *https* : *//chatgpt.com/*

prompt, classificada como a vulnerabilidade número um no *OWASP Top 10 for LLM Applications*[2]. Segundo Greshake et al. (2023), a incapacidade do modelo em distinguir instruções de sistema de entradas de usuário permite que comandos maliciosos sobrescrevam diretrizes de segurança, criando riscos de vazamento de dados e manipulação de comportamento.

As aplicações atuais raramente se limitam a uma única chamada de LLM, elas são construídas como cadeias ou agentes autônomos que interagem com múltiplas ferramentas em ciclos iterativos de raciocínio e ação (XI et al., 2023). Essa arquitetura gera históricos de contexto extensos, exacerbando problemas como o fenômeno *lost in the middle*, onde o modelo falha em recuperar informações relevantes no meio de um contexto longo (LIU et al., 2023a). Rastrear a origem de erros em tais sistemas distribuídos utilizando apenas ferramentas de *logs* convencionais torna-se inviável, uma vez que a interação dinâmica entre múltiplos componentes dificulta a descoberta da causa das falhas (LIAO; VAUGHAN, 2023).

Esses desafios interconectados, custos imprevisíveis, alucinações, vulnerabilidades de segurança e complexidade de agentes, formam a motivação central para este trabalho. Eles evidenciam a necessidade urgente de uma nova classe de ferramentas de engenharia: plataformas de observabilidade especializadas em LLMs, capazes de transformar a "caixa-preta" probabilística das interações com LLMs em um sistema transparente, gerenciável e confiável.

3.2 Trabalhos Relacionados

Para endereçar os complexos desafios operacionais de baixa visibilidade, custo e latência descritos na seção anterior, o mercado de tecnologia desenvolveu um vasto ecossistema de ferramentas de observabilidade. O panorama atual incluindo desde extensões de plataformas de monitoramento tradicionais, como Datadog[3], até soluções especializadas em avaliação e depuração, como Opik[4] e Lunary[5]. No

[2] $https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/$
[3] $https://docs.datadoghq.com/llm_observability/$
[4] $https://www.comet.com/site/products/opik/$
[5] $https://lunary.ai/$

segmento de orquestração e *gateways*, ferramenta como Portkey[6] disputa espaço com funcionalidades avançadas de roteamento.

No entanto, uma análise exaustiva de todas as ferramentas disponíveis excederia o escopo deste trabalho. Foram selecionadas quatro plataformas que ilustram de forma representativa as principais abordagens de engenharia vigentes: a abordagem ecossistêmica proprietária, a plataforma de engenharia de código aberto, a arquitetura baseada em *gateway* e a instrumentação baseada em padrões abertos.

3.2.1 LangSmith

O LangSmith[7] representa a referência atual em soluções proprietárias fortemente acopladas a um ecossistema de desenvolvimento. Projetado para operar em conjunto com os *frameworks* LangChain[8] e LangGraph[9], sua arquitetura distingue-se por implementar um modelo de dados hierárquico capaz de renderizar a árvore de execução completa de cadeias complexas e agentes autônomos. Diferentemente de ferramentas tradicionais, o LangSmith permite a inspeção profunda de passos intermediários de raciocínio e chamadas de ferramentas , além de suportar nativamente *datasets* de referência para testes de regressão automatizados.

Contudo, essa integração profunda cobra seu preço em flexibilidade. Embora ofereça uma experiência fluida para usuários do ecossistema LangChain, a ferramenta apresenta riscos significativos de aprisionamento tecnológico. Por ser fundamentalmente uma solução SaaS de código fechado, exige que dados sensíveis trafeguem para a infraestrutura da LangChain Inc., o que pode violar requisitos de conformidade em setores regulados. Além disso, sua eficácia e facilidade de instrumentação diminuem substancialmente quando utilizada com *stacks* tecnológicas agnósticas.

[6] *https* : *//portkey.ai/*
[7] *https* : *//www.langchain.com/langsmith/observability*
[8] *https* : *//www.langchain.com/*
[9] *https* : *//www.langchain.com/langgraph*

3.2.2 Langfuse

Posicionando-se como a alternativa agnóstica e de código aberto, a Langfuse[10] adota uma arquitetura projetada para desacoplar a instrumentação da análise. A plataforma utiliza SDKs assíncronos que enviam eventos de telemetria em segundo plano, garantindo que o monitoramento não introduza latência na aplicação principal. Seu modelo de dados é centrado no *trace*, enriquecido com metadados de custo, latência e pontuações de qualidade, permitindo avaliações híbridas via `‘‘LLM-as-a-judge’’`, que consiste na avaliação da resposta por um LLM diferente do qual consultado, ou *feedback* humano.

A grande vantagem da Langfuse reside na soberania de dados, por ser *open source* suporta a auto-hospedagem via contêineres Docker, possibilitando que empresas mantenham todos os *prompts* e respostas dentro de sua própria nuvem privada. Essa flexibilidade, no entanto, transfere a complexidade operacional para o usuário. No modelo de auto hospedagem, a responsabilidade pela manutenção do banco de dados e pela escalabilidade da ingestão de eventos recai inteiramente sobre a equipe de engenharia da organização, elevando o custo total de propriedade em cenários de alto volume.

3.2.3 Helicone

O Helicone[11] materializa o conceito de *gateway*, posicionando-se como uma camada de infraestrutura crítica que atua como um orquestrador de tráfego ativo entre as aplicações clientes e os provedores de LLM. Ao centralizar múltiplas integrações em um ponto único de entrada, a ferramenta abstrai a complexidade de gestão de APIs e injeta funcionalidades avançadas diretamente na camada de rede. Construído sobre uma infraestrutura de *edge computing*, o Helicone implementa estratégias de *caching* na borda, permitindo que requisições frequentes sejam respondidas instantaneamente para reduzir custos e latência, além de gerenciar políticas de resiliência, como novas tentativas e *fallbacks*, e governança de acesso. Toda essa gestão é realizada de forma

[10] $https://langfuse.com/$
[11] $https://www.helicone.ai/$

transparente, sem a necessidade de instrumentação intrusiva no código da aplicação, bastando o redirecionamento da URL base do cliente HTTP.

Entretanto, essa arquitetura de intermediação introduz compensações arquiteturais importantes. A consolidação do tráfego transforma o *gateway* em um ponto único de falha na topologia da rede, onde sua indisponibilidade pode interromper o acesso aos provedores de LLMs. Além disso, a necessidade operacional de descriptografar e inspecionar o conteúdo das mensagens para funcionalidades de *cache* e observabilidade exige um alto nível de confiança no provedor ou a opção mandatória pela auto-hospedagem para preservar a confidencialidade de dados sensíveis em ambientes corporativos.

3.2.4 Phoenix

Desenvolvida pela Arize[12], a Phoenix[13] representa a abordagem de observabilidade focada na Ciência de Dados e na depuração profunda de sistemas RAG. Diferente de plataformas puramente de engenharia, a Phoenix adota uma filosofia "local-first", sendo frequentemente executada diretamente em ambientes de desenvolvimento como Jupyter Notebooks antes de ser implantada em produção. Sua arquitetura é projetada para a visualização de dados de alta dimensionalidade, permitindo a inspeção não apenas do texto gerado, mas dos vetores subjacentes.

O grande diferencial arquitetural da Phoenix é seu motor de visualização de *embeddings*. Isso permite que engenheiros identifiquem visualmente *clusters* de alucinações ou lacunas na base de conhecimento recuperada, algo impossível de detectar apenas com logs textuais. Contudo, sua origem focada em *notebooks* pode torná-la menos intuitiva para engenheiros de software tradicionais que buscam apenas monitoramento de latência. Embora a versão *open source* seja poderosa para análise local, a persistência de dados de longo prazo e a colaboração em equipe direcionam o usuário para a plataforma comercial, criando uma barreira de entrada para o monitoramento contínuo em produção sem custos associados.

[12] *https* : *//arize.com/*
[13] *https* : *//phoenix.arize.com/*

3.2.5 Síntese Comparativa

A análise das plataformas LangSmith, Langfuse, Helicone e Phoenix revela que o ecossistema de observabilidade para LLMs é marcado por abordagens distintas, cada uma otimizando um conjunto específico de *trade-offs* entre integração, soberania de dados, facilidade de instrumentação e profundidade analítica.

Essas plataformas demonstram que não existe uma solução universal: cada abordagem resolve uma parte distinta do problema. Conforme detalhado na Tabela 3.1, enquanto LangSmith e Helicone priorizam simplicidade operacional por meio de centralização seja de ecossistema ou de tráfego, Langfuse e Phoenix apostam em maior abertura e flexibilidade, porém transferindo responsabilidades ao usuário. Essa diversidade expõe uma lacuna no ecossistema: a ausência de uma solução que concilie leveza operacional, soberania de dados, independência de ecossistema e suporte nativo às necessidades específicas de aplicações baseadas em LLMs. É justamente nessa interseção que a proposta deste trabalho se insere.

Tabela 3.1: Tabela Comparativa das Soluções Representativas de Observabilidade

Característica	**LangSmith**	**Langfuse**	**Helicone**	**Phoenix**
Funcionalidade diferencial	Integração com LangChain	Avaliação de *prompt*	Otimização de custo	Diagnóstico profundo
Soberania de Dados	Baixa	Alta	Baixa	Alta
Complexidade de Integração	Alta	Média	Muito baixa	Média
Auto-hospedagem	Não	Sim	Sim	Sim

Fonte: Elaborado pelo autor.

3.3 Proposta de Sistema

Após a análise dos desafios operacionais gerados pela integração de softwares com LLMs e do ecossistema de ferramentas existentes, este trabalho propõe o desenvolvimento de uma plataforma de observabilidade com diferenciais estratégicos. O estudo dos trabalhos relacionados revelou um mercado composto por soluções robustas, porém frequentemente associadas a alta complexidade, modelos de negócio restritivos ou forte dependência de ecossistemas específicos. Esses fatores evidenciam uma lacuna para uma solução que priorize simplicidade, soberania dos dados e acessibilidade.

Diante desse cenário, a proposta deste trabalho concentra-se na criação de um *gateway* de observabilidade leve e de código aberto, projetado para implantação flexível e que permita às equipes manter controle sobre seus dados e infraestrutura. A solução terá foco no monitoramento das métricas críticas que afetam a viabilidade de aplicações baseadas em LLMs, como custo, latência e consumo de *tokens*, com o objetivo de reduzir a barreira de entrada e proporcionar visibilidade operacional sem que equipes precisem lidar com custos elevados ou complexidade excessiva de integração.

Embora a arquitetura de referência aqui proposta tenha sido concebida para ser agnóstica e extensível a múltiplos provedores de LLM, este trabalho delimita o escopo de implementação e validação a API Gemini[14]. Esta decisão estratégica visa viabilizar a prova de conceito focando na profundidade da observabilidade em um único ecossistema, servindo como base para futuras expansões de adaptadores para outros provedores

3.3.1 Módulos do Sistema

Para materializar esses objetivos, a solução proposta adota uma abordagem centrada em governança e observabilidade de interações com LLMs. O sistema atua como um *gateway* de API centralizado que fornece controle de acesso, auditoria

[14] *https://aistudio.google.com/*

detalhada e monitoramento de custos. A arquitetura foi concebida de forma modular, separando claramente as responsabilidades do sistema, conforme detalhado nas subseções seguintes.

3.3.1.1 Autenticação e Autorização

A segurança da plataforma é alicerçada em um módulo robusto responsável por verificar a identidade dos usuários e governar suas permissões. Este componente centraliza o fluxo de entrada no sistema e a distribuição de privilégios, garantindo que funcionalidades sensíveis sejam acessadas apenas por entidades legítimas.

O sistema implementa um mecanismo de autenticação híbrido, capaz de operar tanto com credenciais locais, como e-mail e senha, quanto com provedores externos de identidade. Essa flexibilidade permite integrar diferentes fluxos de autenticação sem comprometer a segurança ou a compatibilidade com ambientes corporativos.

A arquitetura de permissões é estruturada segundo o modelo de *Role-Based Access Control* (RBAC), que define dois perfis nativos responsáveis por moldar a interface de gerenciamento. O perfil de administrador possui acesso irrestrito às funcionalidades de governança, podendo auditar, criar e revogar contas de usuários. Já o usuário possui escopo limitado à administração de seus próprios recursos, como dados pessoais e credenciais de segurança, acessíveis por meio de seu painel dedicado.

Integrado a este módulo está o gerenciamento de *tokens* de API. Diferentemente do acesso via interface web, a interação com o *gateway* de LLM exige credenciais de máquina de longa duração. A plataforma delega aos usuários a autonomia para gerar e revogar seus próprios *tokens* de API, que atuam como as chaves de autenticação para as requisições programáticas processadas pelo sistema.

3.3.1.2 Gateway de LLM

Este módulo constitui o núcleo funcional da solução, atuando como um *gateway* centralizado para as interações com modelos de linguagem. A arquitetura do processamento foi projetada priorizando a resiliência, o sistema utiliza internamente uma

fila de tarefas para encapsular a lógica de comunicação externa. Essa abordagem permite a implementação de mecanismos de tolerância a falhas, como novas tentativas automáticas e *backoff exponencial*, mantendo, contudo, a interface síncrona para o cliente. O controlador aguarda a conclusão do processamento interno antes de retornar a resposta, preservando a semântica padrão do ciclo de requisição-resposta do protocolo HTTP.

Neste módulo, a plataforma atua como um componente de mediação direta para aplicações, operando através de um fluxo contínuo que se inicia com a interceptação da requisição formatada nativamente pelo SDK. Simultaneamente ao tráfego de dados, o sistema realiza o processamento interno, extraindo o *payload* e registrando os dados de observabilidade antes de encaminhar a solicitação à API real do LLM. O ciclo encerra-se com a adaptação de resposta, etapa crucial na qual o sistema formata o retorno do modelo para espelhar com exatidão a estrutura de dados esperada pelo SDK, assegurando uma integração transparente. Especificamente para este trabalho, o módulo de adaptação será configurado para mimetizar os contratos de interface da API Gemini, interceptando e traduzindo suas estruturas de requisição e resposta específicas

Essa abordagem assegura que aplicações projetadas para o uso de SDKs de LLMs possam adotar a plataforma de observabilidade sem necessidade de refatoração de código, bastando a reconfiguração da *baseURL* no cliente HTTP do SDK.

3.3.1.3 Auditoria e Engenharia de Custos

Imediatamente após o processamento da requisição, o sistema aciona o *pipeline* de auditoria, operacionalizando o objetivo central de transformar a baixa visibilidade de interações com um LLM em um processo transparente e auditável. Diferentemente de um *proxy*, que apenas direciona os dados, a plataforma garante a persistência estruturada de cada interação, independentemente de seu desfecho, seja ele sucesso ou falha.

O registro de auditoria não será apenas um *log* textual, mas um modelo de dados relacional que captura três dimensões críticas para a observabilidade: os dados da

transação, que incluem o identificador do usuário, o *payload* de entrada e a resposta completa gerada pelo modelo, garantindo soberania e possibilidade de auditoria forense ao serem mantidos na própria infraestrutura; as métricas de desempenho, que abrangem o código de *status* HTTP e a latência *end-to-end* em milissegundos, permitindo identificar gargalos de rede ou degradações do provedor; e as métricas de custo, produzidas por um motor de precificação que, que após extrair a contagem total *tokens*, consulta um catálogo interno que diferencia valores entre *tokens* de entrada e saída para calcular e registrar com precisão o custo financeiro de cada transação.

3.3.1.4 Dashboard

Para converter os dados brutos de auditoria em diagnóstico, a plataforma disponibiliza um *dashboard* analítico. Esta interface atua como a camada de apresentação, permitindo que gestores e desenvolvedores visualizem o comportamento das aplicações de IA em tempo real.

A interface organiza as informações em três níveis de granularidade: os indicadores de desempenho, que apresentam métricas agregadas como custo total acumulado, volume de requisições, taxa de erro e latência média para uma avaliação imediata da saúde do sistema; a análise temporal e de tendências, composta por gráficos de séries temporais que revelam padrões de consumo e possíveis anomalias, incluindo a evolução diária de custos, a distribuição dos códigos de status e o comportamento da latência ao longo do tempo; e a inspeção granular, oferecida por uma interface de exploração de *logs* que exibe o histórico completo das interações, permitindo examinar cada transação individual com seus *prompts* e respostas, o que é essencial tanto para auditorias de qualidade quanto para processos de depuração.

3.3.2 Requisitos Funcionais

Os Requisitos Funcionais (RF) constituem a especificação comportamental do software. Segundo Sommerville (2011), estes requisitos são declarações dos serviços que o sistema deve fornecer, de como o sistema deve reagir a entradas específicas e

de como deve se comportar em situações particulares. Eles definem as capacidades operacionais da plataforma, cobrindo desde a autenticação até a visualização de métricas. A Tabela 3.2 apresenta a lista completa das funcionalidades implementadas no sistema.

Tabela 3.2: Tabela de Requisitos Funcionais

ID	**Requisito**	**Descrição**
RF-01	Cadastrar Usuário	O sistema deve permitir que um novo usuário se cadastre fornecendo nome, e-mail e senha.
RF-02	Recuperar Senha	O sistema deve permitir que o usuário solicite e redefina sua senha através de um *link* de recuperação seguro enviado por e-mail.
RF-03	Encerrar Sessão	O sistema deve permitir que um usuário autenticado encerre sua sessão.
RF-04	Diferenciar Papéis	O sistema deve diferenciar os usuários em dois papéis: usuário e administrador.
RF-05	Listar Usuários	O sistema deve permitir que administradores listem todos os usuários cadastrados na plataforma.
RF-06	Criar Usuário	O sistema deve permitir que administradores criem novos usuários manualmente.
RF-07	Editar Usuário	O sistema deve permitir que administradores editem as informações de qualquer usuário.
RF-08	Excluir Usuário	O sistema deve permitir que administradores excluam usuários da plataforma.
RF-09	Personificar Conta	O sistema deve permitir que administradores "personifiquem" a conta de um usuário para fins de suporte.
RF-10	Gerenciar Perfil	O sistema deve permitir que qualquer usuário autenticado gerencie seu próprio perfil.
RF-11	Gerar *Token*	O sistema deve permitir que um usuário autenticado gere *tokens* de API para interagir com o *gateway*.

ID	Requisito	Descrição
RF-12	Listar *Tokens*	O sistema deve permitir que um usuário autenticado liste todos os seus *tokens* de API gerados.
RF-13	Revogar *Token*	O sistema deve permitir que um usuário autenticado revogue seus *tokens* de API a qualquer momento.
RF-14	Autenticar Requisições	O sistema deve autenticar todas as requisições de API através do *token* gerado.
RF-15	Disponibilizar Interface	O sistema deve disponibilizar uma interface de comunicação compatível com o SDK que mimetize o formato oficial do provedor.
RF-16	Adaptar Requisição	O sistema deve ser capaz de interpretar a requisição no formato nativo, extrair os dados e formatar a resposta preservando o contrato do SDK.
RF-17	Consultar Cache	Antes de encaminhar a requisição ao provedor, o sistema deve consultar o banco de dados buscando por uma requisição que tenha o mesmo *prompt* e modelo, retornando a resposta armazenada se encontrada.
RF-18	Aguardar Processamento	O sistema deve aguardar o resultado do processamento interno para retornar a resposta na mesma requisição HTTP.
RF-19	Registrar Auditoria	O sistema deve registrar cada requisição feita a API do LLM em um histórico de auditoria, associando-a ao usuário.
RF-20	Persistir *Payload*	O registro de auditoria deve armazenar os dados completos da requisição e da resposta gerada.
RF-21	Registrar Desempenho	O sistema deve registrar as métricas de desempenho, incluindo *status* e latência da API externa.
RF-22	Contabilizar *Tokens*	O sistema deve registrar as métricas de uso de *tokens* de entrada e saída para cada requisição.

ID	Requisito	Descrição
RF-23	Calcular Custos	O sistema deve calcular o custo monetário exato de cada requisição com base nos *tokens* e na tabela de preços.
RF-24	Apresentar *Dashboard* Analítico	O sistema deve apresentar um *dashboard* contendo estatísticas agregadas e gráficos temporais sobre custos, volume de requisições e latência.
RF-25	Visualizar Histórico	O sistema deve permitir que o Usuário visualize seu histórico pessoal de requisições em lista.
RF-26	Inspecionar Detalhes	O sistema deve permitir que o Usuário inspecione os detalhes de uma requisição específica.
RF-27	Visualizar Dados Globais	O sistema deve permitir que o Administrador visualize o *dashboard* com dados agregados de todos os usuários.
RF-28	Filtrar Métricas	O sistema deve permitir que o Administrador filtre os dados do *dashboard* para visualizar métricas de um usuário específico.

3.3.3 Regras de Negócio

Para assegurar que a plataforma atenda aos objetivos de governança e observabilidade estabelecidos, o comportamento do sistema foi estruturado sobre um conjunto de Regras de Negócio (RN). Conforme define Sommerville (2011), estas regras derivam das políticas organizacionais e do domínio da aplicação, impondo restrições sobre o funcionamento do sistema para garantir que ele opere em conformidade com os processos e padrões da empresa. A Tabela 3.3 apresenta a especificação consolidada das regras e sua relação direta com os RF correspondentes.

Tabela 3.3: Tabela de Regras de Negócio

ID	**Regra de Negócio**	**Descrição**	**RFs**
RN001	Autenticação Híbrida	O sistema deve aceitar autenticação via provedor de identidade federada e credenciais locais.	RF01, RF05
RN002	Unicidade de Conta	Não é permitido o registro de múltiplos usuários com o mesmo endereço de e-mail.	RF01
RN003	Recuperação Segura	A redefinição de senha deve ocorrer exclusivamente via *link* temporário e de uso único enviado ao e-mail cadastrado.	RF02
RN004	Hierarquia de Papéis	O sistema deve distinguir estritamente as permissões entre os papéis de administrador e usuário.	RF06, RF09, RF12
RN005	Gestão Administrativa	Apenas usuários com papel de administrador podem listar, criar ou personificar outros usuários.	RF07, RF08, RF09, RF10, RF11
RN006	Soberania de Tokens	Usuários autenticados devem ter autonomia para gerar e revogar seus próprios *tokens* de API .	RF14, RF15, RF16
RN007	Validação de Token	O sistema deve interceptar requisições e exigir autenticação via cabeçalho *Authorization*, validando integridade antes do processamento.	RF15, RF16
RN008	Campos Obrigatórios	Requisições devem conter, obrigatoriamente, o corpo da mensagem e a identificação do modelo de destino.	RF16

ID	Regra de Negócio	Descrição	RFs
RN009	Estratégia de Cache	O sistema deve verificar a existência de *prompt* prévio idêntico. Em caso de correspondência, deve retornar a resposta armazenada imediatamente.	RF17, RF20
RN010	Resiliência e Retries	Em caso de falha de comunicação com o provedor, o sistema deve realizar até três tentativas de reexecução com estratégia de *exponential backoff*.	RF18
RN011	Processar Interação em Fila	O processamento da interação externa deve ser encapsulado em uma fila de *jobs* interna, mantendo a interface síncrona para o cliente.	RF18
RN012	Falha Definitiva	Após o esgotamento das tentativas de reexecução, o sistema deve registrar o erro permanentemente e retornar uma mensagem de falha formatada.	RF18, RF24
RN013	Persistência Mandatória	Todas as transações processadas pelo sistema, sejam elas bem-sucedidas, falhas ou provenientes de cache, devem ser registradas de forma persistente no banco de dados.	RF21
RN014	Metadados de Auditoria	O registro deve conter, minimamente: ID do usuário, *timestamp*, latência , *status code*, origem da resposta, *prompt* e resposta.	RF23, RF24, RF28
RN015	Cálculo de Custo	O sistema deve calcular o custo financeiro baseando-se na contagem de *tokens* multiplicada pela tabela de preços vigente. Respostas de cache devem ter custo zero ou reduzido.	RF23, RF24

ID	Regra de Negócio	Descrição	RFs
RN016	Privacidade e Segregação	O administrador tem acesso a métricas globais e individuais. Contudo, o acesso ao conteúdo bruto das transações, histórico de requisições e respostas, é restrito exclusivamente ao usuário proprietário.	RF25, RF26, RF27, RF28

3.3.4 Casos de Uso

Para Sommerville (2011), casos de uso são uma técnica fundamental de modelagem de requisitos que descreve as interações entre os usuários e o sistema, focando em como o software deve responder a estímulos externos para alcançar um objetivo específico. Eles servem como uma ponte entre os requisitos funcionais abstratos e a implementação técnica, fornecendo um contexto narrativo para a validação das regras de negócio. A seguir segue os principais casos de uso do sistema.

UC01: Cadastrar Usuário

Ator Principal: Usuário

Descrição: Permitir que um usuário crie uma conta fornecendo nome, e-mail e senha válidos.

Pré-condição: O usuário não possuir conta no sistema.

Pós-condição: Conta criada no sistema.

Fluxo Principal:

1. O usuário acessa a página de cadastro.
2. O sistema exibe o formulário com os campos de nome, e-mail e senha.
3. O usuário preenche os dados e submete o formulário.

4. O sistema valida a unicidade do e-mail e a complexidade da senha.

5. O sistema cria a nova conta com perfil usuário.

6. O sistema autentica o usuário e o redireciona para a página de *dashboard*.

Fluxo Alternativo (5): E-mail inválido ou já cadastrado

1. O sistema detecta que o e-mail informado já está cadastrado ou possui formato inválido, exibe uma mensagem de erro e destaca o campo para correção retornando ao passo 2 do Fluxo Principal.

Fluxo Alternativo (5): Senha não atende os critérios mínimos de segurança

1. O sistema detecta que que a senha não atingiu os critérios mínimos, destacando o campo para correção e retorna para passo 2 do Fluxo Principal.

UC02: Fazer Login

Ator Principal: Usuário

Descrição: Permitir que um usuário inicie uma sessão no sistema.

Pré-condição: O usuário deve possuir conta ativa no sistema ou conta do Google ativa.

Pós-condição: Usuário autenticado e direcionado para a página de *Dashboard*.

Fluxo Principal:

1. O usuário acessa a página de login.

2. O sistema apresenta o formulário de credenciais com os campos para e-mail e senha e o botão “Entrar com Google”.

3. O usuário insere as credenciais locais, e-mail e senha, e confirma.

4. O sistema valida as credenciais locais.

5. O sistema gera a sessão de acesso e redireciona para a página de *dashboard*.

Fluxo Alternativo (2): Login via Google

1. No passo 2 do Fluxo Principal, o usuário seleciona a opção "Entrar com Google".
2. O sistema redireciona o usuário para a autenticação do provedor externo.
3. O sistema recebe o *token* de confirmação do Google.
4. O sistema valida o *token* e identifica a conta do usuário.
5. O fluxo retorna ao passo 5 do Fluxo Principal.

Fluxo Alternativo (3): Credenciais Inválidas

1. No passo 3 do Fluxo Principal, o usuário insere e-mail ou senha incorretos.
2. O sistema tenta validar as credenciais.
3. O sistema identifica que as credenciais são inválidas.
4. O sistema exibe a mensagem: "E-mail ou senha inválidos.".
5. O sistema retorna ao passo 2 do Fluxo Principal.

UC03: Gerar *Token* de API

Ator Principal: Usuário

Descrição: Permitir que um usuário crie *token* para realizar a integração com o sistema de observabilidade.

Pré-condição: O usuário deve possuir sessão ativa no sistema.

Pós-condição: *Token* gerado.

Fluxo Principal:

1. O usuário acessa a página de"*Tokens* de API" nas configurações.
2. O sistema lista os *tokens* ativos.
3. O usuário seleciona "Gerar Novo *Token*" e define um nome para identificação.
4. O sistema gera um novo *Bearer Token* e o exibe uma única vez.
5. O usuário copia e armazena o *token* em local seguro.

UC04: Enviar Requisição

Ator Principal: Sistema Cliente

Descrição: Permitir que um sistema cliente envie requisições para uma API de LLM através do sistema.

Pré-condição: O sistema cliente deve possuir um *token* de API do sistema e ter ele integrado a sua requisição.

Pós-condição: O sistema cliente recebeu a resposta de seu *prompt*.

Fluxo Principal:

1. O sistema cliente envia uma requisição para o *endpoint* do sistema usando o SDK do seu provedor de LLM.
2. O sistema intercepta a chamada e verifica se existe *token* do sistema ativo no cabeçalho da chamada.
3. O sistema verifica se no banco de dados não existe uma requisição bem sucedida que contenha o mesmo *prompt* e modelo.
4. O sistema enfileira a requisição em um *job* interno para processamento.
5. O *worker* processa o *job*, repassando a chamada à API oficial LLM.
6. O sistema recebe a resposta do provedor de LLM.

7. O sistema calcula o custo baseado nos *tokens* do LLM trafegados e persiste os *logs* de auditoria e as métricas de latência.

8. O sistema retorna a resposta ao cliente no formato *JavaScript Object Notation* (JSON) exato esperado pelo SDK.

Fluxo Alternativo (3): Existe Cache para a Requisição

1. Se existir requisição anterior bem sucedida que possui *prompt* e modelo idênticos, o sistema busca a resposta que ela teve e retorna para a requisição atual.

Fluxo Alternativo (5): Falha e Retentativa

1. Se a API do LLM falhar, o sistema aplica a estratégia de *Backoff Exponencial* e retenta a operação até 3 vezes.

2. Se a falha persistir após as tentativas, o sistema registra o erro definitivo e retorna o código do erro ao cliente.

UC05: Visualizar Detalhes de uma Interação

Ator Principal: Usuário

Descrição: Permitir que um usuários visualize a requisição e resposta integral de uma interação com LLM.

Pré-condição: O usuário deve estar autenticado no sistema, e ter consumido um *token* de API.

Pós-condição: O usuário visualiza o conteúdo completo da requisição e resposta.

Fluxo Principal:

1. O usuário acessa a página de histórico.

2. O sistema exibe a lista de todas as interações que o usuário teve com uma API de LLM em que usou o *token* do sistema.

3. O usuário encontra a interação desejada
4. O usuário clica no botão de ações da interação que direciona para a página de detalhes.
5. O usuário é direcionado para a página de detalhes da interação.

UC06: Promover Usuário a Administrador

Ator Principal: Administrador

Descrição: Permitir que um administrador promova um usuário a administrador.

Pré-condição: O administrador deve estar autenticado no sistema.

Pós-condição: Um usuário recebe privilégios se tornando um novo administrador.

Fluxo Principal:

1. O administrador acessa o painel "Usuários".
2. O sistema lista todos os usuários cadastrados na plataforma.
3. O administrador seleciona um usuário e clica no botão de editar.
4. o administrador edita as configurações de perfil do usuário elevando seus privilégios para administrador
5. O sistema aplica a alteração solicitada e atualiza as configurações do usuário.

3.3.5 Modelagem de Dados

A modelagem de dados desempenha, um papel fundamental na arquitetura deste sistema de observabilidade, sendo responsável por traduzir as necessidades abstratas de auditoria, custos e governança em um esquema lógico coerente. A natureza crítica dos dados processados envolvem desde credenciais de segurança e controle de acesso

até o conteúdo sensível de *prompts* e respostas de LLMs,o modelo foi projetado para assegurar a atomicidade das transações e a consistência dos relacionamentos.

Figura 3.1: Diagrama Entidade-Relacionamento do Sistema

A entidade central desta modelagem é a tabela `users`, ela é a raiz de integridade referencial do sistema, armazenando de forma segura as credenciais de acesso e os dados de perfil. Para garantir a governança e a segregação de funções, esta entidade relaciona-se diretamente com a tabela `roles`, que define a hierarquia de permissões baseada em papéis, distinguindo administradores de usuários comuns. Ainda neste contexto de segurança, a entidade `reset_password_tokens` suporta os fluxos de

recuperação de conta, armazenando *hashes* temporários para permitir a redefinição de senhas sem intervenção administrativa.

A modelagem prevê ainda uma separação clara entre o acesso humano e o acesso de máquina. Para isso, a entidade `auth_access_tokens` gerencia as chaves de integração geradas pelos usuários. Diferente de uma sessão de navegador, esses registros controlam o acesso de aplicações externas, permitindo a revogação granular de permissões e garantindo que o tráfego de API seja autenticado e rastreável até sua origem.

O núcleo funcional da plataforma reside na interação entre as entidades de auditoria e custo. A tabela `prompt_requests` registra cada interação processada pelo sistema, estrutura dados críticos como o tempo de latência, a contagem exata de *tokens* de entrada e saída, e o conteúdo das mensagens, viabilizando auditorias forenses e de qualidade. Para suportar a engenharia de custos associada a esses registros, o sistema utiliza a entidade `language_models` como um catálogo dinâmico de parametrização. Esta tabela armazena as configurações e os preços vigentes de cada modelo, permitindo que o cálculo financeiro das requisições na `prompt_requests` seja realizado com base em dados atualizáveis.

Capítulo 4

Observator

Este capítulo detalha o processo de construção e a implementação técnica da plataforma de observabilidade de LLMs, conforme a arquitetura e os requisitos definidos no Capítulo 3. A solução foi implementada como uma aplicação web completa, focada em modularidade, segurança e desempenho.

4.1 Tecnologias Utilizadas

A materialização da arquitetura de referência proposta no capítulo anterior exigiu a seleção criteriosa de uma pilha tecnológica visando garantir a manutenibilidade, a segurança de tipos e a escalabilidade. A implementação fundamentou-se no ecossistema NodeJs, adotando o TypeScript como linguagem unificadora para assegurar a integridade e robustez do código em todas as camadas da aplicação.

Optou-se por uma abordagem arquitetural de monolito modular, integrando um *back-end* estruturado e coeso com uma interface reativa moderna. Esta estratégia visa reduzir a complexidade operacional, mantendo, contudo, a separação lógica de responsabilidades. A seguir, são detalhadas as decisões de engenharia e as ferramentas selecionadas para compor cada subsistema, justificando sua adoção frente às alternativas de mercado.

4.1.1 AdonisJS

O núcleo da aplicação foi desenvolvido sobre o AdonisJS[1], um *framework* web para Node.js que adota a premissa *TypeScript-first*. Sua arquitetura adere estritamente ao padrão *Model-View-Controller* (MVC), oferecendo uma estrutura de diretórios opinativa e organizada, além de um ambiente de desenvolvimento integrado com suporte a *Hot Module Replacement* (HMR) para o código de *back-end*.

A escolha desta tecnologia fundamentou-se em sua filosofia *batteries included*. Diferentemente de *micro-frameworks* como Express[2] que exigem a montagem manual de componentes de terceiros, o AdonisJS fornece nativamente um ecossistema coeso e robusto. Esta abordagem foi decisiva para acelerar o ciclo de desenvolvimento, pois provê soluções integradas e testadas para os desafios centrais da plataforma.Dentre os componentes nativos que justificam essa escolha arquitetural, destacam-se o Lucid[3] que realiza a modelagem de dados e abstração *Structured Query Language* (SQL), Bouncer para o controle de autorização granular, Auth para um gerenciamento seguro de autenticação e Adonis Jobs para o processamento de de filas.

Para a inicialização da infraestrutura, optou-se pela utilização de um *starter kit*[4] desenvolvido por membro da comunidade Adonis, em detrimento do padrão oficial. A escolha deste artefato específico justifica-se pela disponibilização de configurações arquiteturais avançadas pré-configuradas, estendendo as funcionalidades nativas com uma camada adicional de abstração. Essa decisão estratégica eliminou a necessidade de configurações triviais iniciais, permitindo foco imediato nas regras de negócio do *gateway*.

4.1.1.1 Lucid

O *framework* AdonisJS oferece flexibilidade na integração com *Object-Relational Mappings* (ORMs), esta tecnologia atua como uma camada de abstração responsável por mapear as tabelas de um banco de dados relacional para classes e objetos

[1] $https://adonisjs.com/$
[2] $https://expressjs.com/$
[3] $https://lucid.adonisjs.com/docs/introduction$
[4] $https://github.com/filipebraida/adonisjs-starter-kit$

da aplicação, eliminando a necessidade de manipulação direta de instruções SQL para operações padrão. A arquitetura do *framework* permite que o desenvolvedor selecione a biblioteca de sua preferência, com a documentação oficial listando suporte a alternativas de mercado como Prisma[5] e TypeORM[6].

Para este projeto, no entanto, optou-se pela utilização do Lucid. Esta decisão justifica-se pelo fato de a biblioteca já vir pré-configurada na estrutura base do AdonisJS e, crucialmente, por ser desenvolvida pela mesma equipe do *framework*, o que minimiza riscos de incompatibilidade e assegura maior estabilidade nas atualizações. O Lucid destaca-se também por abstrair a complexidade das consultas mantendo o acesso ao potencial do SQL, oferecendo uma API para abstrair operações avançadas.

Para a implementação desta plataforma, selecionou-se o PostgreSQL como o Sistema Gerenciador de Banco de Dados (SGBD) devido sua robustez, confiabilidade e, principalmente, em seu suporte avançado a tipos de dados não-estruturados, especificamente o tipo *JSON Binary* (JSONB)

4.1.1.2 Auth

O gerenciamento de identidade foi implementado através do módulo nativo de Autenticação do *AdonisJS*. Dada a natureza híbrida da plataforma, que opera simultaneamente como uma aplicação web interativa e um *gateway*, foi necessário orquestrar duas estratégias de autenticação distintas.

- Guard Web: Configurado para proteger a *Single-Page Application* (SPA), este guardião utiliza o mecanismo tradicional de sessões baseadas em *cookies* seguros. Ele é responsável por manter o estado de *login* dos administradores e usuários nas páginas privadas dos sistema.
- Guard API: Configurado especificamente para proteger os *endpoints* do *gateway* de LLM. Este guardião utiliza *tokens* de acesso opacos, permitindo que aplicações externas e SDKs se autentiquem de forma programática. Esta

[5] *https://www.prisma.io/*
[6] *https://typeorm.io/*

estratégia dissocia a autenticação da máquina da sessão do usuário, garantindo que as integrações não expirem com o fechamento do navegador e possam ser revogadas individualmente.

4.1.1.3 Bouncer

O controle de acesso baseado em papéis foi implementado por meio do *Bouncer*, o módulo oficial de autorização do *AdonisJS*. Esse modelo de controle define permissões com base nos papéis atribuídos aos usuários, em vez de associar permissões individualmente a cada um. Assim, um usuário herda automaticamente as permissões do seu papel, como administrador ou usuário comum, tornando o gerenciamento de acessos mais seguro, escalável e consistente.

O *Bouncer* fornece uma camada robusta para o gerenciamento dessas permissões, permitindo a centralização das regras de autorização em *policies*, que encapsulam as decisões de segurança sobre quem pode executar determinadas ações no sistema. Cada *policy* define métodos que representam ações específicas, como visualizar, criar, atualizar ou remover, e retorna um valor *boolean* indicando se o usuário tem ou não permissão para realizá-las.

4.1.1.4 Adonis Jobs

A interação com LLM caracteriza-se por uma latência intrínseca elevada e variável, aumentada pela extensão dos *prompts* e por eventuais congestionamentos de rede. A execução síncrona dessas operações no fluxo principal da requisição HTTP pode bloquear o processo do servidor, impedindo o atendimento de novas solicitações e conduzindo ao esgotamento de recursos. Este cenário configura um clássico problema do tipo produtor-consumidor, onde a taxa de entrada de requisições supera a capacidade imediata de processamento, resultando em degradação de desempenho.

Para mitigar esses efeitos e gerenciar operações de longa duração a solução proposta incorpora um sistema robusto de filas orquestrado pelo pacote Adonis Jobs. Construído sobre a biblioteca de sistema de filas BullMQ[7], o pacote fornece uma

[7] *https* : *//bullmq.io/*

integração limpa e idiomática dessa tecnologia ao ecossistema AdonisJS, utilizando o Redis[8] para armazenamento e gerenciamento das filas.

A adoção deste mecanismo foi determinante para a resiliência e escalabilidade do *gateway*. Ao delegar o processamento intensivo a *workers* dedicados, reduz-se drasticamente a carga sobre o servidor web principal. A persistência e orquestração dessas tarefas são suportadas pelo Redis, que fornece um canal de comunicação de baixa latência e alta vazão, requisito crítico para ambientes de alto volume de tráfego.

Em termos de implementação, o *job* encapsula toda a lógica da tarefa: executa a chamada ao provedor de LLM, coleta métricas de telemetria, latência, contagem de *tokens* e conteúdo, e estrutura a resposta. O controlador, por sua vez, atua apenas como despachante, submetendo a tarefa à fila e aguardando a resolução do processamento para retornar a resposta ao cliente. Em cenários de erro, o *job* captura o contexto da falha, garantindo a rastreabilidade completa da operação.

Embora o sistema mantenha uma interface síncrona para o cliente, o ganho em escalabilidade horizontal é significativo. A arquitetura permite que os *workers* operem de forma independente, possibilitando a adição dinâmica de novas instâncias de processamento conforme a demanda aumenta. Isso permite distribuir a carga de trabalho entre múltiplas máquinas, assegurando a absorção eficiente de picos de tráfego com eficiência de recursos.

A robustez operacional é reforçada pelo suporte nativo a política de retentativa. Falhas transitórias, como instabilidades de rede ou *timeouts* do provedor, são tratadas automaticamente através do reprocessamento do *job*. Para evitar a saturação do serviço externo em momentos de instabilidade, aplica-se a estratégia de *exponential backoff*, que aumenta progressivamente o intervalo entre as tentativas. Isso transforma o tratamento de erros em um processo controlado e determinístico.

Em síntese, este modelo materializa a solução para o problema produtor-consumidor: o servidor HTTP atua como produtor ágil, enfileirando demandas, enquanto os *workers* atuam como consumidores resilientes, processando a carga conforme a capacidade

[8] *https* : *//redis.io/*

disponível. Esse desacoplamento arquitetural é o pilar que garante o equilíbrio de carga, a tolerância a falhas e a manutenção dos níveis de serviço mesmo sob condições adversas.

4.1.2 Padrão Arquitetural: AI Gateway

Sobre a fundação tecnológica do AdonisJS e do sistema de filas, o núcleo da aplicação foi desenhado para materializar o padrão arquitetural de *AI Gateway*. Conceitualmente, um *AI Gateway* atua como um ponto único de entrada e controle entre as aplicações clientes e os modelos fundacionais, abstraindo a complexidade de múltiplas APIs e centralizando políticas de tráfego e auditoria. A implementação deste componente no Observator foi projetada para operar como um *middleware* de alta performance, interceptando requisições para injetar observabilidade sem adicionar latência significativa.

Para garantir a compatibilidade com sistemas legados que já possuem integração via SDK com provedores de LLMs, o *gateway* implementa o padrão de projeto *adapter*. Para o escopo desta implementação, o desenvolvimento foi direcionado especificamente para o suporte a API Gemini. Desta forma, a interface exposta mimetiza estritamente o contrato de dados esperado pelo SDK oficial deste provedor. Isso permite que a plataforma atue como um componente de substituição direta: a aplicação cliente envia a requisição formatada para o LLM, o *gateway* a intercepta, processa a auditoria e devolve a resposta no formato exato que o SDK espera, tornando a camada de observabilidade transparente para o código consumidor.

4.1.3 Arquitetura de Projeto e *Front-end*

A organização do código-fonte foi estruturada visando a manutenibilidade a longo prazo e a escalabilidade modular. Para tal, adotou-se uma estrutura de monorepo, gerenciada pelo gerenciador de pacotes PNPM[9] e orquestrada pela ferramenta de *build system* Turborepo[10]. Esta estratégia permite o controle centralizado de dependências,

[9] *https://pnpm.io/*
[10] *https://turborepo.com/*

otimização do tempo de compilação através de armazenamento em cache remoto e a segregação clara de responsabilidades, a estrutura de diretórios reflete essa divisão lógica.

- `apps/web`: Abriga a aplicação principal, contendo a lógica de negócio e as camadas de apresentação.
- `packages/ui`: Contém o *design system* proprietário, isolado como uma biblioteca interna de componentes visuais reutilizáveis.

Esta separação física favorece a padronização da interface. A aplicação principal consome o pacote de *User Interface* (UI) como uma dependência, garantindo que elementos gráficos, estilizados uniformemente via TailwindCSS[11] e componentes base shadcn/ui, mantenham a consistência visual e evitem a duplicação de código *Cascading Style Sheets* (CSS).

A arquitetura da interface é definida pela integração de três tecnologias fundamentais: (i) React[12] que é utilizado como biblioteca de renderização, permitindo a construção de interfaces reativas complexas, essenciais para os painéis de *analytics* e tabelas de dados do sistema. (ii) o Inertia[13] atuando como protocolo de ligação entre o *back-end* e o *front-end*. Ele permite que os controladores do servidor renderizem componentes React diretamente, injetando dados via *props*. Isso viabiliza a experiência de usuário de uma SPA sem a sobrecarga arquitetural de desenvolver e manter uma aplicação apartada exclusivamente para o *front-end*. (iii) Vite[14] que é empregado como ferramenta de empacotamento. Por utilizar módulos ES nativos e suportar HMR, o Vite acelera significativamente o ciclo de desenvolvimento e *feedback* visual.

O resultado é uma arquitetura de monolito modular robusta, que combina a simplicidade de implantação de uma aplicação tradicional com a interatividade moderna de uma SPA baseada em componentes.

[11] *https://tailwindcss.com/*
[12] *https://react.dev/*
[13] *https://inertiajs.com/*
[14] *https://vite.dev/*

4.2 Implementação do Observator

Após a definição da arquitetura e das tecnologias, esta seção apresenta a materialização da plataforma Observator. A seguir, são detalhados os principais fluxos de interação e as interfaces desenvolvidas, evidenciando como os requisitos funcionais e as regras de negócio foram traduzidos em componentes visuais e mecanismos de controle. A apresentação segue a jornada típica do usuário, desde o provisionamento de acesso até a análise avançada de métricas de observabilidade.

4.2.1 Registro, Autenticação e Recuperação

A arquitetura do módulo de gestão de identidade foi projetada para harmonizar requisitos não funcionais de segurança e usabilidade, abrangendo os ciclos de registro, autenticação e recuperação de credenciais. Estas funcionalidades formam a barreira de controle de acesso primária, assegurando que apenas entidades autorizadas interajam com os recursos protegidos do sistema, garantindo assim a confidencialidade e a integridade dos dados.

O processo de registro de contas constitui o ponto de entrada para novos usuários. Nesta etapa, o sistema coleta e processa as credenciais essenciais como e-mail e senha, além do nome de usário. A implementação da tela de registro incorpora mecanismos rigorosos de validação de dados de entrada, aplicando regras de negócio que asseguram a unicidade do identificador e a conformidade com políticas de complexidade de senha. A figura 4.1 exibe o formulário de registro.

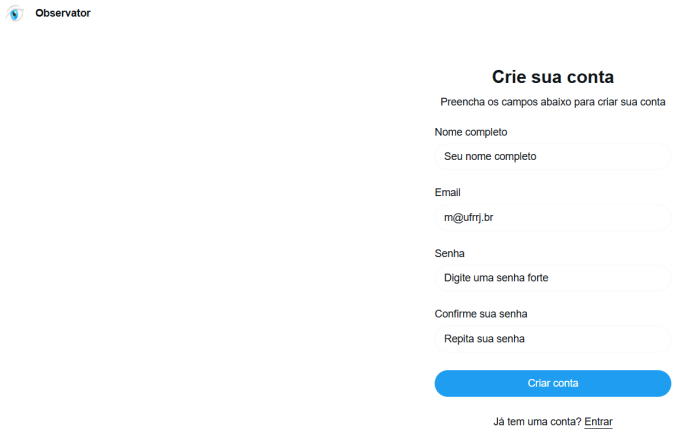

Figura 4.1: Tela de Cadastro de Usuários

Adicionalmente, visando a eficiência operacional, o sistema implementa uma estratégia de autenticação implícita pós-registro: após a persistência bem-sucedida da nova conta, a sessão é estabelecida automaticamente, eliminando a redundância de um login subsequente. O tratamento de exceções é realizado em tempo real na camada de apresentação, fornecendo retorno semântico ao usuário em casos de violação das regras de validação ou conflitos de dados.

A materialização do mecanismo de controle de acesso é visualizada na Figura 4.2, que exibe a tela de login desenvolvida. O layout reflete a estratégia de autenticação híbrida adotada na arquitetura, segregando visualmente as credenciais locais do provedor de autenticação externo.

Esta disposição busca otimizar a usabilidade ao oferecer um caminho de menor resistência via autenticação social, mitigando a fadiga de senhas sem excluir a opção tradicional. Após a submissão bem-sucedida nesta tela, o *back-end* estabelece o *token* de sessão, garantindo a persistência segura do estado de autenticação.

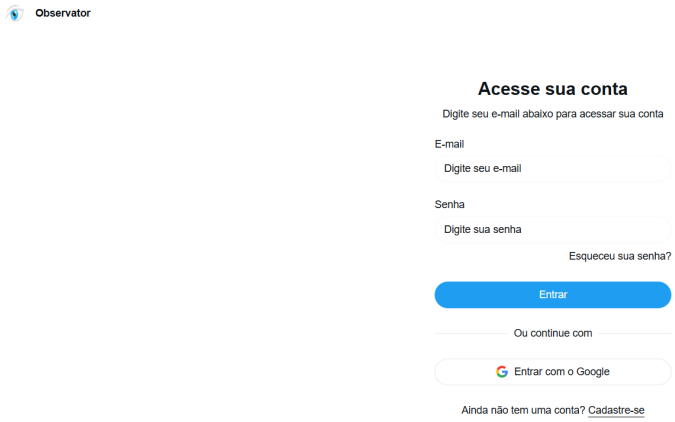

Figura 4.2: Tela de Login

Por fim, o ciclo de gestão de credenciais é completado pelo módulo de recuperação, projetado para restaurar o acesso em cenários de perda de senha sem comprometer a segurança da conta. A tela de solicitação, apresentada na Figura 4.3, atua como o gatilho inicial deste processo.

Arquiteturalmente, o fluxo é desacoplado: a solicitação do usuário dispara um evento interno que enfileira o envio de um e-mail transacional de forma assíncrona. Este e-mail contém uma *Uniform Resource Locator* (URL) assinada com criptografia que possui tempo de expiração curto. Esta abordagem garante que apenas o detentor do e-mail possa acessar a tela de redefinição dentro de uma janela temporal restrita, mitigando riscos de ataques de força bruta ou interceptação de endereços antigos

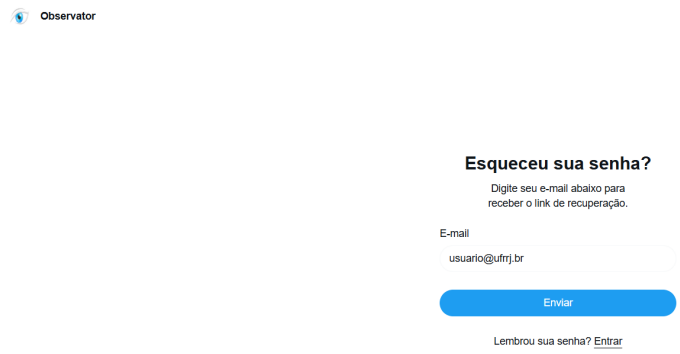

Figura 4.3: Tela de Recuperação de Senha

4.2.2 Dashboard

Imediatamente após a autenticação, o sistema apresenta o *dashboard*, conforme demonstrado na Figura 4.4. Esta tela constitui o ponto central de observabilidade, consolidando métricas críticas de consumo e desempenho em uma visão unificada.

A arquitetura de informação foi estruturada hierarquicamente para oferecer diagnósticos rápidos. O topo da tela destaca indicadores de desempenho que quantificam o volume transacional total, o custo financeiro acumulado e a latência média. Na camada subsequente, a visualização gráfica possibilita a análise de tendências temporais, como a evolução de custos diária e tempo de resposta por requisição, bem como a avaliação da confiabilidade sistêmica através da proporção entre requisições bem-sucedidas e falhas.

Adicionalmente, para o perfil com privilégios administrativos, a tela disponibiliza mecanismos de filtragem granular, essenciais para isolar as métricas de usuários específicos e subsidiar processos de auditoria detalhada.

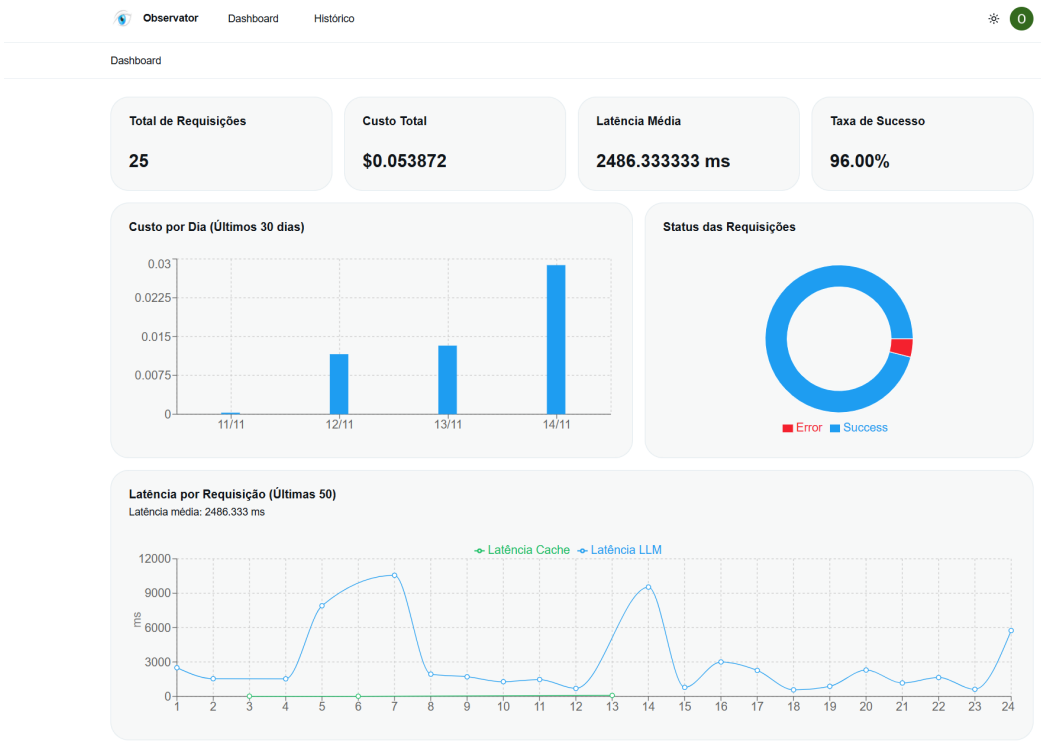

Figura 4.4: Tela de Dashboard

4.2.3 Histórico de Interações

Complementando a visão agregada do *dashboard*, o histórico de interações oferece a capacidade de rastreabilidade granular. A tela, ilustrada na Figura 4.5, implementa uma visualização tabular projetada para a auditoria detalhada de cada interação com o *gateway*.

Cada registro na tabela apresenta os metadados essenciais para a identificação e análise da transação, organizados em dimensões complementares. O sistema exibe o contexto e o conteúdo da operação, incluindo a data da execução, o modelo utilizado e um fragmento do *prompt* para identificação visual rápida. Em termos de telemetria, são apresentados o *status* da operação, a latência da resposta e a contagem de *tokens*. Por fim, para a análise de eficiência financeira, a tabela detalha o custo exato da transação e o indicador de origem, cache ou API LLM, permitindo verificar a eficácia da estratégia de economia de recursos.

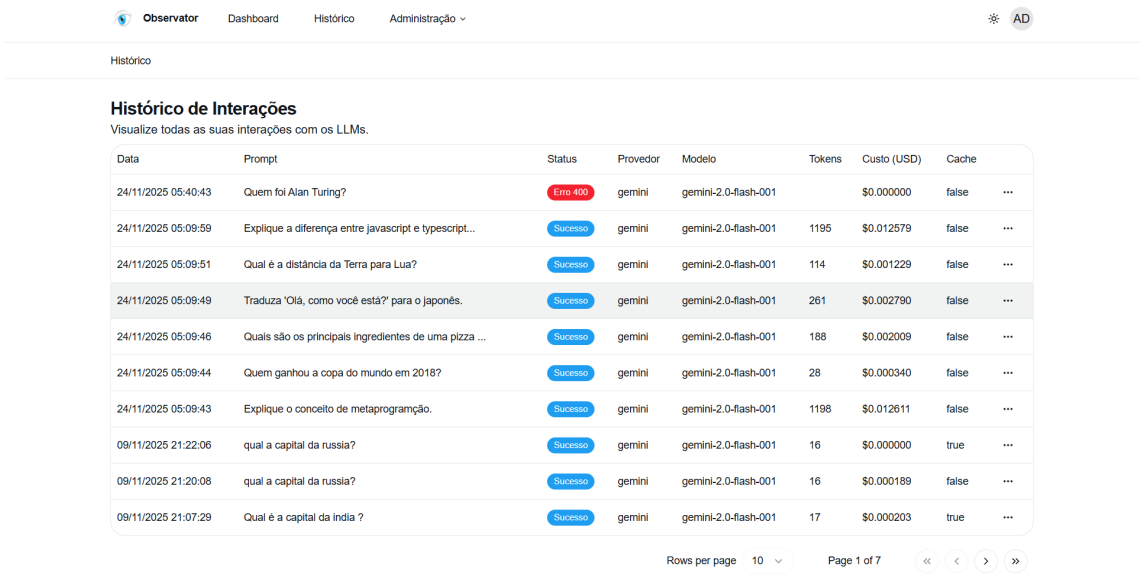

Figura 4.5: Tela de Histórico de Interações

4.2.4 Detalhes da Interação

A partir da tabela de histórico, o usuário pode navegar a até à tela de detalhes da interação. Esta tela foi projetada para oferecer uma visão completa e granular do ciclo de vida da interação entre o sistema e o modelo de linguagem.

O componente central desta tela é o visualizador de dados estruturados, que apresenta os objetos completos de *request* e *response*. Este formato permite que usuários inspecionem cada campo da interação, incluindo parâmetros de configuração, cabeçalhos de segurança, tempo de execução e metadados técnicos retornados pelo provedor.

Adicionalmente, a tela processa o JSON bruto para apresentar uma seção dedicada ao conteúdo da resposta, onde o texto principal produzido pelo LLM é extraído e renderizado de forma legível. Esta funcionalidade é crítica para a validação semântica, facilitando a leitura humana sem a poluição visual da sintaxe JSON.

A relevância desta funcionalidade no contexto de observabilidade reside em seu papel como ferramenta de depuração e análise forense. Ao expor integralmente o fluxo de comunicação, conforme ilustrado na Figura 4.6, o sistema permite investigar

a raiz de falhas, comparar comportamentos entre diferentes modelos e auditar o conteúdo exato trafegado.

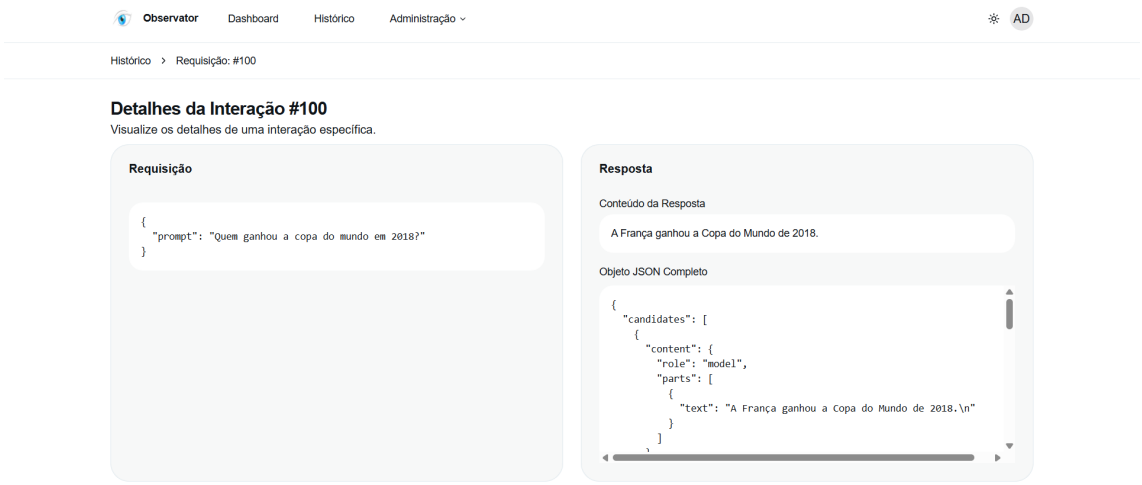

Figura 4.6: Tela de Detalhamento de Interação

4.2.5 Painel de Governança de Usuários

Restrito ao perfil de administrador, o módulo de gerenciamento de usuários implementa os requisitos de controle centralizado sobre o ciclo de vida das identidades. A tela principal, apresentada na Figura 4.7, organiza a base de usuários em uma estrutura tabular, expondo metadados críticos como nome, e-mail e nível de privilégio.

Além da opção de listagem de usuários, o sistema integra a este painel o mecanismo de registro centralizado, detalhado na Figura 4.8. Esta funcionalidade permite o registro completo de novos usuários, incluindo nome, e-mail, senha e definição imediata do papel sem a necessidade de cadastro manual pelo próprio utilizador. Esse processo é estratégico para ambientes corporativos, assegurando que a gestão de acessos permaneça centralizada e que nenhuma credencial seja gerada sem a supervisão explícita da administração.

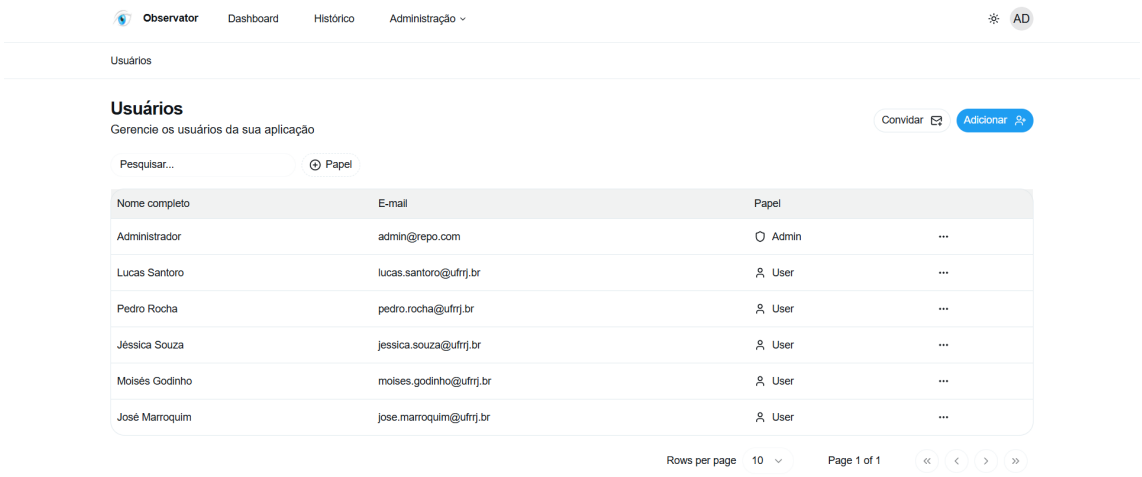

Figura 4.7: Tela do Painel Administrativo

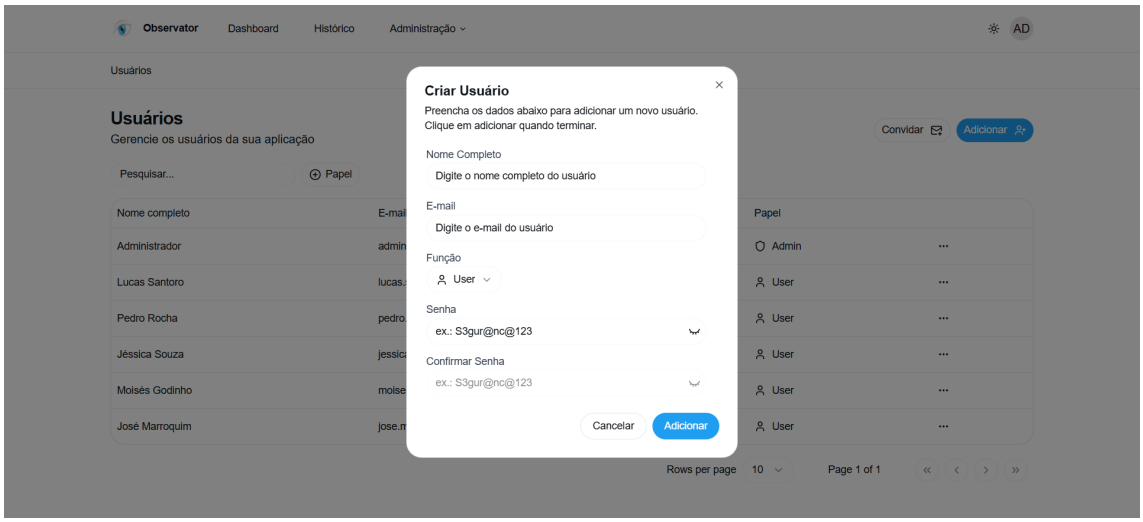

Figura 4.8: Tela de Registro Centralizado de Contas

A governança é complementada pela tela de edição, que materializa na prática o modelo RBAC. Conforme ilustrado na Figura 4.9, o administrador tem a capacidade de modificar os atributos de identidade e o nível de privilégio atribuído ao usuário.

Esta funcionalidade confere dinamismo à gestão de segurança, permitindo o ajuste de acessos em resposta a mudanças organizacionais. O sistema viabiliza tanto a elevação de privilégios quanto a restrição de acesso, assegurando que as permissões permaneçam sempre alinhadas às responsabilidades atuais de cada colaborador.

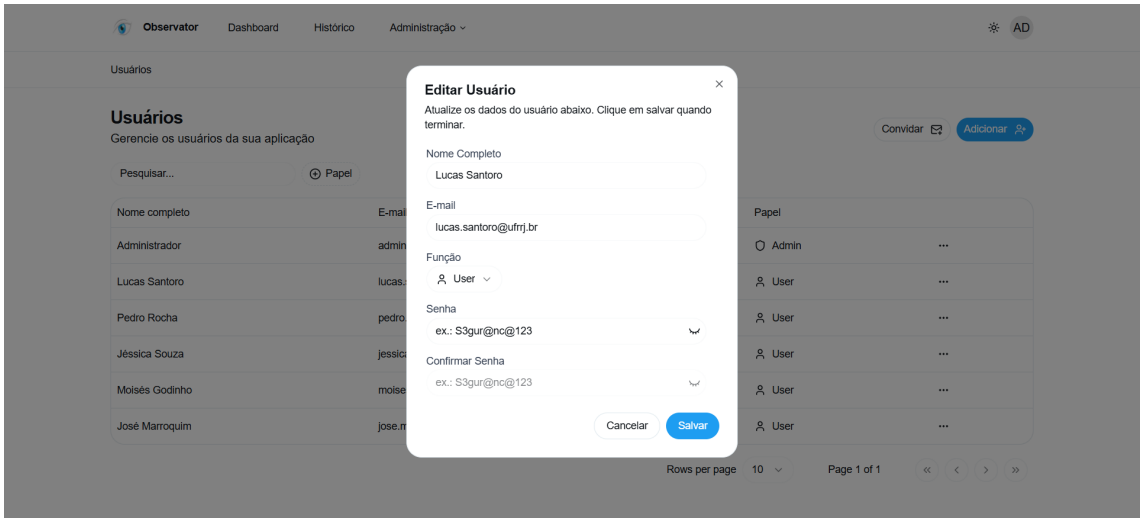

Figura 4.9: Tela de Edição de Usuário

4.2.6 Gestão de Conta e Configurações

O módulo de configurações foi projetado sob uma arquitetura de navegação unificada , compartilhando um *layout* lateral persistente. Esta decisão de *design* visa centralizar as funcionalidades de personalização, segurança e integração, garantindo uma experiência coesa e reduzindo a carga cognitiva necessária para alternar entre contextos de gestão da conta.

4.2.6.1 Perfil e Identidade Visual

A tela de configurações de perfil, apresentada na Figura 4.10, permite a gestão dos dados cadastrais e da identidade visual do usuário. A implementação foca na autonomia, permitindo a atualização dinâmica de atributos como nome de exibição e avatar. Essas informações são persistidas no banco de dados e propagadas para o cabeçalho da aplicação e para os *logs* de auditoria, garantindo consistência na identificação do usuário em todo o sistema.

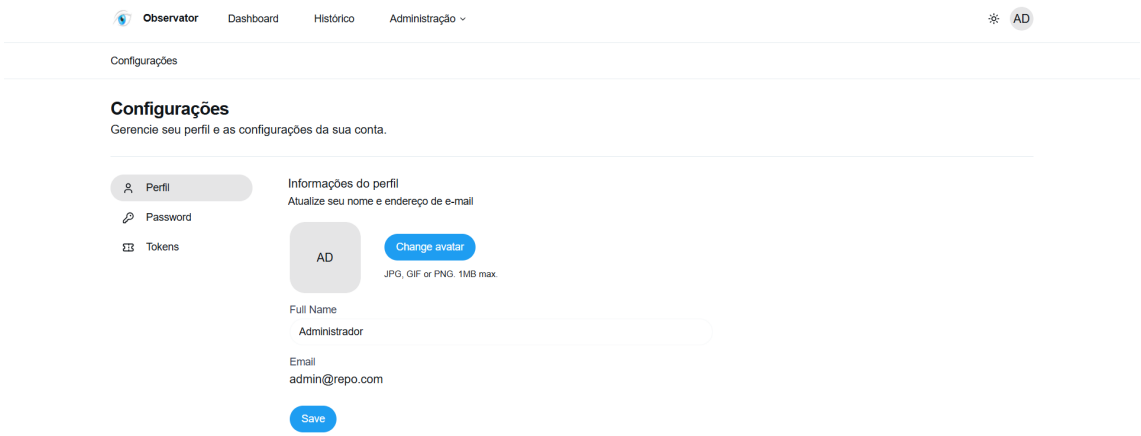

Figura 4.10: Tela de Configuração de Perfil

4.2.6.2 Segurança e Credenciais

A seção de segurança implementa os controles de redefinição de credenciais. Conforme ilustrado na Figura 4.11, o formulário impõe regras estritas de validação: a exigência da senha atual para autorizar a mudança e a aplicação de políticas de complexidade para a nova senha. Esta funcionalidade é vital para a manutenção da higiene de segurança da conta.

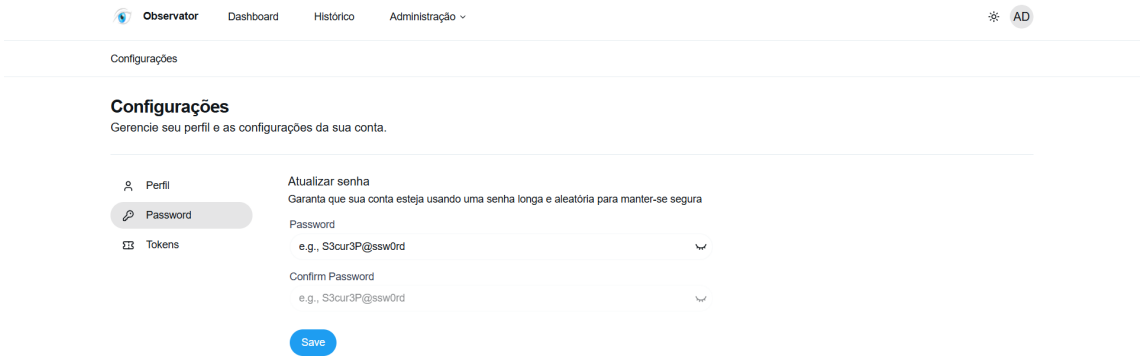

Figura 4.11: Tela de Alteração de Senha

4.2.6.3 Gestão de Tokens

Por fim, a área de *tokens* de API representa o ponto de convergência entre a tela humana e a automação. Esta tela, exibida na Figura 4.12, instrumentaliza o requisito de soberania de acesso, permitindo que o usuário gerencie o ciclo de vida de suas chaves de acesso programático. Essa interface disponibiliza uma listagem de *tokens* ativos e ações para a geração de novas chaves e a revogação imediata de credenciais.

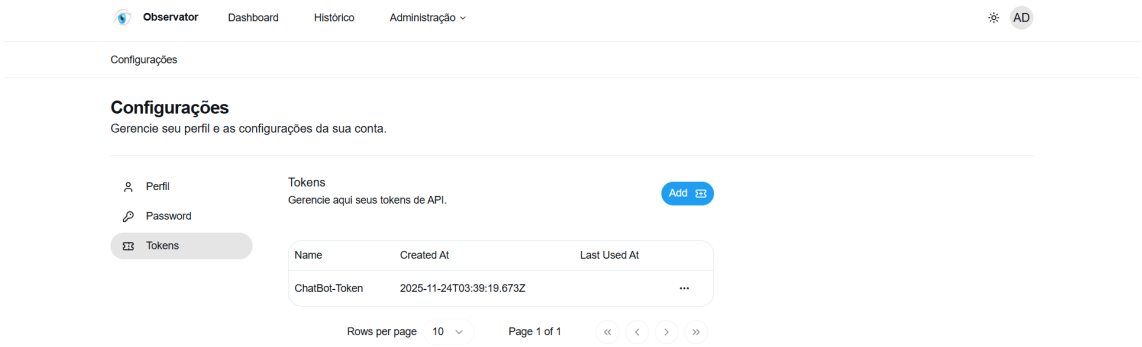

Figura 4.12: Tela do Gerenciamento de Tokens de Acesso à API

É através do *token* gerado nesta tela que desenvolvedores e sistemas externos realizam a autenticação junto ao *gateway*. Para consumir os recursos de um provedor LLM juntamente com o Observator, a aplicação cliente deve incorporar este segredo ao cabeçalho HTTP padrão de autorização, conforme demonstrado no exemplo de requisição abaixo:

Código 4.1: Exemplo de Integração via SDK Google

```
1 import { GoogleGenAI } from '@google/genai'
2
3 const genAI = new GoogleGenAI({
4   apiKey: `${process.env.GOOGLE_API_KEY}`,
5   httpOptions: {
```

```
6       baseUrl: `${process.env.OBSERVATOR_APP_URL}`,
7       headers: {
8         'Authorization': `Bearer ${authTokenGateway}`,
9         'x-api-key': `${process.env.GOOGLE_API_KEY}`,
10        'targetURL': 'https://generativelanguage.googleapis.com',
11      },
12    },
13  });
14
15  async function generateContent() {
16    const response = await genAI.models.generateContent({
17      model: "gemini-2.0-flash-001",
18      contents: "Por que o céu é azul?",
19    });
20  }
21
22  generateContent();
```

Capítulo 5

Conclusão

O presente trabalho abordou os desafios emergentes na engenharia de software voltada para Inteligência Artificial, especificamente no contexto da operacionalização de LLMs. Partindo da premissa de que a integração direta com provedores de LLMs introduz riscos de imprevisibilidade de custos e dependência tecnológica, a pesquisa culminou no desenvolvimento e validação de uma plataforma de observabilidade.

Este capítulo final sintetiza os resultados alcançados, reflete sobre as implicações arquiteturais da solução proposta e delineia as fronteiras do escopo atual, apontando caminhos para a evolução contínua do projeto.

5.1 Considerações Finais

A materialização da plataforma Observator demonstrou que a adoção de um padrão arquitetural de *gateway* é uma estratégia eficaz para retomar o controle sobre aplicações baseadas em LLMs. Ao interpor uma camada de infraestrutura controlada entre o cliente e o provedor, foi possível transformar um processo de "caixa-preta" em um fluxo transparente e auditável, sem impor complexidade excessiva ao desenvolvimento.

Do ponto de vista técnico, a implementação validou a robustez da pilha tecnológica escolhida. O uso do ecossistema Node.js com TypeScript e AdonisJS provou-se

adequado para lidar com operações de *input* e *output* intensivo, enquanto a arquitetura de processamento baseada em filas garantiu a resiliência necessária para mitigar a instabilidade inerente às APIs de LLMs. A estratégia de substituição direta, mimetizando o contrato de interface do SDK oficial, confirmou-se como um diferencial de usabilidade, reduzindo drasticamente o atrito de integração.

Em termos de negócio e governança, o sistema atingiu os objetivos de soberania de dados e controle financeiro. A capacidade de persistir *prompts* e respostas em infraestrutura própria, aliada ao cálculo preciso de custos por transação e ao mecanismo de cache, oferece uma alternativa viável e econômica frente às soluções *SaaS* proprietárias de mercado, que muitas vezes exigem o envio de dados sensíveis para terceiros.

Conclui-se, portanto, que a observabilidade não é apenas uma funcionalidade acessória, mas um requisito não funcional crítico para qualquer sistema que possua alguma integração com LLMs em produção. A solução apresentada oferece uma fundação básica para equipes que buscam equilibrar inovação com responsabilidade operacional.

5.2 Limitações e Trabalhos Futuros

O desenvolvimento de uma plataforma de engenharia de software é um processo iterativo e contínuo. A validade desta pesquisa reside não apenas nos resultados alcançados, mas também na delimitação clara de suas fronteiras experimentais. Esta seção analisa as restrições de escopo assumidas durante a implementação do protótipo e projeta um roteiro estratégico para a evolução da plataforma, visando sua maturação para ambientes produtivos de larga escala.

5.2.1 Limitações Atuais

Embora a solução proposta tenha atingido seus objetivos fundamentais, o escopo deste trabalho apresenta limitações inerentes ao tempo e aos recursos de um projeto acadêmico. Primeiramente, a implementação atual restringe-se exclusivamente à

integração com o a API Gemini. Embora a arquitetura tenha sido concebida para ser extensível, o sistema ainda não suporta nativamente outros provedores com presença significativa no mercado, como OpenAI[1] ou Anthropic[2], limitando sua aplicação imediata em ambientes de múltiplos modelo.

Do ponto de vista da experiência em tempo real, o *gateway* opera atualmente apenas no modo de resposta completa, não suportando o retorno de dados via *streaming* ou RAG. Essa restrição, decorrente da complexidade de manter conexões persistentes e gerenciar o fluxo contínuo de dados através da camada de processamento intermediária, pode impactar a percepção de latência em interfaces de *chat* que exigem interatividade imediata.

Além disso, o sistema carece de mecanismos avançados de controle de consumo, como a gestão automatizada de orçamentos, que permitiria bloquear o uso de uma chave de API após atingir um teto financeiro mensal, não foi contemplada nesta versão. Por fim, o mecanismo de cache implementado baseia-se na correspondência exata do texto, não possuindo capacidades de busca vetorial para realizar cache semântico, o que reduz a eficiência em cenários onde perguntas distintas possuem a mesma intenção semântica.

5.2.2 Trabalhos Futuros

Como roteiro para a evolução da plataforma, a prioridade reside na expansão da arquitetura para um modelo agnóstico e multi-provedor. A generalização do padrão *adapter* é fundamental para suportar a normalização de requisições e respostas de outros ecossistemas líderes, como OpenAI e Anthropic. Essa evolução permitiria que as aplicações clientes alternassem entre diferentes modelos de linguagem dinamicamente, sem a necessidade de refatoração de código, concretizando a visão de um *gateway* universal.

Em paralelo, busca-se elevar a eficiência e a inteligência do sistema através da integração de bancos de dados vetoriais. A implementação de *cache semântico*,

[1] *https* : *//openai.com/*
[2] *https* : *//www.anthropic.com/*

suportada pelo armazenamento de *embeddings* dos *prompts*, permitiria reutilizar respostas baseadas na similaridade de intenção e não apenas na correspondência textual exata. Adicionalmente, propõe-se a criação de um módulo de avaliação automatizada *LLM-as-a-Judge*, onde um modelo auxiliar auditaria a qualidade, toxicidade e relevância das respostas geradas em um *pipeline* de pós-processamento, automatizando a garantia de qualidade.

Por fim, para consolidar a governança operacional, sugere-se o desenvolvimento de um sistema de alertas em tempo real. A implementação de canais de notificação como *webhooks* ou e-mail que disparem alertas imediatos quando limites orçamentários ou taxas de erro críticas forem atingidos transformaria a plataforma de uma ferramenta passiva de análise histórica em um mecanismo ativo de controle e resposta a incidentes.

Referências

BENDER, E. M. et al. On the dangers of stochastic parrots: Can language models be too big? . In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. New York, NY, USA: Association for Computing Machinery, 2021. (FAccT '21), p. 610–623. ISBN 9781450383097. Disponível em: <https://doi.org/10.1145/3442188.3445922>.

BOMMASANI, R. et al. On the opportunities and risks of foundation models. *CoRR*, abs/2108.07258, 2021. Disponível em: <https://arxiv.org/abs/2108.07258>.

BROWN, T. et al. Language models are few-shot learners. *Advances in neural information processing systems*, v. 33, p. 1877–1901, 2020.

CHANG, Y. et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, ACM, v. 15, n. 3, 2023.

CHOUDHARY, V. Software as a service: Implications for investment in software development. In: IEEE. *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. [S.l.], 2007. p. 209a–209a.

DAVIS, M. *The universal computer: The road from Leibniz to Turing*. [S.l.]: AK Peters/CRC Press, 2018.

DIAZ-DE-ARCAYA, J. et al. Large language model operations (llmops): Definition, challenges, and lifecycle management. In: *2024 9th International Conference on Smart and Sustainable Technologies (SpliTech)*. [S.l.: s.n.], 2024. p. 1–4.

FERRUCCI, D. et al. Building watson: An overview of the deepqa project. *AI Magazine*, v. 31, n. 3, p. 59–79, Jul. 2010. Disponível em: <https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2303>.

GAO, Y. et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. Disponível em: <https://arxiv.org/abs/2312.10997>.

GOEL, K. et al. *Niyama : Breaking the Silos of LLM Inference Serving*. 2025. Disponível em: <https://arxiv.org/abs/2503.22562>.

GRESHAKE, K. et al. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In: *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*. New York, NY, USA: Association for Computing Machinery, 2023. (AISec '23), p. 79–90. ISBN 9798400702600. Disponível em: <https://doi.org/10.1145/3605764.3623985>.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural computation*, MIT press, v. 9, n. 8, p. 1735–1780, 1997.

HUYEN, C. *Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications*. Sebastopol: O'Reilly Media, 2022.

JI, Z. et al. Survey of hallucination in natural language generation. *ACM Computing Surveys*, v. 55, n. 12, 2023.

JURAFSKY, D.; MARTIN, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, with Language Models*. 3rd. ed. [s.n.], 2025. Online manuscript released August 24, 2025. Disponível em: <https://web.stanford.edu/~jurafsky/slp3/>.

KÁLMÁN, R. E. On the general theory of control systems. In: *Proceedings of the First International Congress on Automatic Control*. Moscou: Butterworths, 1960. v. 1, p. 481–492.

LEWIS, M. et al. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: *Proceedings of the 58th annual meeting of the association for computational linguistics*. [S.l.: s.n.], 2020. p. 7871–7880.

LIANG, P. et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

LIAO, Q. V.; VAUGHAN, J. W. *AI Transparency in the Age of LLMs: A Human-Centered Research Roadmap*. 2023. Disponível em: <https://arxiv.org/abs/2306.01941>.

LIU, E.; NEUBIG, G.; ANDREAS, J. *An Incomplete Loop: Instruction Inference, Instruction Following, and In-context Learning in Language Models*. 2024. Disponível em: <https://arxiv.org/abs/2404.03028>.

LIU, N. F. et al. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. Disponível em: <https://arxiv.org/abs/2307.03172>.

LIU, P. et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, ACM New York, NY, v. 55, n. 9, p. 1–35, 2023.

MASLEJ, N. et al. *Artificial Intelligence Index Report 2023*. 2023. Disponível em: <https://arxiv.org/abs/2310.03715>.

MIKOLOV, T. et al. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

PENNINGTON, J.; SOCHER, R.; MANNING, C. D. Glove: Global vectors for word representation. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. [S.l.: s.n.], 2014. p. 1532–1543.

RADFORD, A. et al. Improving language understanding by generative pre-training. San Francisco, CA, USA, 2018.

RADFORD, A. et al. Language models are unsupervised multitask learners. *OpenAI blog*, v. 1, n. 8, p. 9, 2019.

RAFFEL, C. et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, v. 21, n. 140, p. 1–67, 2020.

RIO-CHANONA, R. M. del; LAURENTSYEVA, N.; WACHS, J. Large language models reduce public knowledge sharing on online q&a platforms. *PNAS Nexus*, v. 3, n. 9, p. pgae400, 09 2024. ISSN 2752-6542. Disponível em: <https://doi.org/10.1093/pnasnexus/pgae400>.

SENNRICH, R.; HADDOW, B.; BIRCH, A. Neural machine translation of rare words with subword units. In: *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*. [S.l.: s.n.], 2016. p. 1715–1725.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. *Nature*, v. 529, p. 484–503, 2016. Disponível em: <http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html>.

SOMMERVILLE, I. Software engineering 9th edition. *ISBN-10*, v. 137035152, p. 18, 2011.

SRIDHARAN, C. *Distributed Systems Observability*. Sebastopol: O'Reilly Media, 2018.

STRUBELL, E.; GANESH, A.; MCCALLUM, A. Energy and policy considerations for deep learning in nlp. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence: Association for Computational Linguistics, 2019. p. 3645–3650.

SUN, G. et al. *CoIn: Counting the Invisible Reasoning Tokens in Commercial Opaque LLM APIs*. 2025. Disponível em: <https://arxiv.org/abs/2505.13778>.

VASWANI, A. et al. Attention is all you need. *Advances in neural information processing systems*, v. 30, 2017.

VELASCO, A. A.; TSIRTSIS, S.; GOMEZ-RODRIGUEZ, M. *Auditing Pay-Per-Token in Large Language Models*. 2025. Disponível em: <https://arxiv.org/abs/2510.05181>.

WEI, J. et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, v. 35, p. 24824–24837, 2022.

XI, Z. et al. *The Rise and Potential of Large Language Model Based Agents: A Survey*. 2023. Disponível em: <https://arxiv.org/abs/2309.07864>.

XIAO, C.; YANG, Z. Streaming, fast and slow: Cognitive load-aware streaming for efficient llm serving. In: *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2025. (UIST '25), p. 1–13. Disponível em: <http://dx.doi.org/10.1145/3746059.3747721>.

YAO, S. et al. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

ZHOU, Y. et al. *Large Language Models Are Human-Level Prompt Engineers*. 2023. Disponível em: <https://arxiv.org/abs/2211.01910>.