UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO (2025)

DESENVOLVIMENTO DE UM SISTEMA DISTRIBUÍDO COM MICROSSERVIÇOS E APRENDIZADO DE MÁQUINA PARA DETECÇÃO DE FRAUDES TRANSACIONAIS EM TEMPO REAL[1]

DEVELOPMENT OF A DISTRIBUTED SYSTEM WITH MICROSERVICES AND MACHINE LEARNING FOR REAL-TIME TRANSACTIONAL FRAUD DETECTION

Afonso Simão de Góis Neto[2], Paulo Henrique Lopes Silva[3]

RESUMO

Este trabalho apresenta o desenvolvimento e a avaliação de um sistema distribuído para detecção de fraudes financeiras em tempo real, integrando conceitos de arquitetura de software e aprendizado de máquina com o objetivo de minimizar a latência para a inferência da análise transacional. O sistema foi construído visando desacoplar os componentes em microsserviços conteinerizados que utilizam comunicação híbrida: síncrona para inferência de alto desempenho e assíncrona para auditoria e persistência. O modelo preditivo supervisionado foi acoplado a um *pipeline* de automação que permite o retreinamento e atualização nas réplicas de inferência. Os resultados experimentais, obtidos por meio de testes de carga controlados, demonstraram que a arquitetura suporta alta concorrência com vazão estável e taxa de erro nula até o ponto de saturação dos recursos. Conclui-se que a solução proposta atende aos requisitos de escalabilidade, resiliência e baixa latência, validando a viabilidade técnica da aplicação em ambientes financeiros.
Palavras-chave: detecção de fraudes; sistemas distribuídos; microsserviços; aprendizagem de máquina.

ABSTRACT

This work presents the development and evaluation of a distributed system for real-time financial fraud detection, integrating software architecture and machine learning concepts with the objective of minimizing latency in transactional analysis inference. The system was built based on decoupling components into containerized microservices utilizing hybrid communication: synchronous for high-performance inference and asynchronous for auditing and persistence. The supervised predictive model was coupled with an automation pipeline that

[1] Trabalho de Conclusão de Curso defendido ao Curso de Graduação em Ciências da Computação, no dia 18 de dezembro de 2025, como requisito parcial para obtenção do título de Especialista junto a Universidade Federal Rural do Semi-Árido (UFERSA).

[2] Orientando do Curso de Graduação em Ciências da Computação, da Universidade Federal Rural do Semi-Árido (UFERSA). E-mail: afonso.neto48453@alunos.ufersa.edu.br

[3] Orientador do Curso de Graduação em Ciências da Computação, da Universidade Federal Rural do Semi-Árido (UFERSA). E-mail: phenrique@ufersa.edu.br

allows for retraining and dynamic updating of inference replicas. Experimental results, obtained through controlled load tests, demonstrated that the architecture supports high concurrency with stable throughput and zero error rate up to the resource saturation point. It is concluded that the proposed solution meets scalability, resilience, and low latency requirements, validating the technical feasibility of its application in financial environments.
Keywords: fraud detection; distributed systems; microservices; machine learning.

1 INTRODUÇÃO

A transformação digital do setor financeiro, impulsionada pela popularização dos pagamentos instantâneos e pelo crescimento exponencial do comércio eletrônico, alterou drasticamente o panorama das transações bancárias. Se por um lado essa digitalização trouxe conveniência e velocidade, por outro, ampliou a superfície de ataque para criminosos cibernéticos.

Conforme o 19º Anuário Brasileiro de Segurança Pública (FBSP, 2025), o Brasil vive uma mudança estrutural na criminalidade, com uma migração acentuada para o ambiente virtual. Em 2024, foram registrados cerca de 2,2 milhões de casos de estelionato, representando um crescimento de 408% em relação a 2018, com uma média de 4 golpes por minuto, evidenciando a incapacidade dos métodos tradicionais em conter essa onda de ataques.

Neste cenário, a resposta do setor bancário tem sido o investimento estratégico em novas tecnologias de defesa. Segundo a 34ª edição da Pesquisa Febraban de Tecnologia Bancária, o aumento da sofisticação das ameaças cibernéticas exige soluções que combinem resiliência e automação. O relatório aponta que 80% das instituições financeiras já estão implementando casos de uso de Inteligência Artificial voltados especificamente para a detecção de fraudes e prevenção à lavagem de dinheiro, enquanto 70% focam na proteção de sistemas contra ataques cibernéticos (FEBRABAN, 2025).

Entretanto, a implementação dessas tecnologias enfrenta barreiras em relação à arquitetura. Sistemas de detecção legados, baseados em regras estáticas ou modelos analíticos de processamento em lote (*batch*), mostram-se inadequados para a velocidade das transações modernas. Conforme observado por Abdallah, Maarof e Zainal, arquiteturas convencionais que dependem de processamentos periódicos falham em impedir a fraude no momento de sua ocorrência (ABDALLAH et al., 2016). O problema central reside na latência e na escalabilidade: em um ecossistema de pagamentos instantâneos, qualquer atraso na validação compromete a experiência do usuário legítimo.

Para endereçar as limitações de escalabilidade inerentes aos sistemas monolíticos, a adoção de uma arquitetura de microsserviços apresenta-se como uma evolução necessária. Contudo, essa transição impõe um desafio arquitetural conhecido: a introdução de latência de rede na comunicação entre serviços distribuídos, um custo inexistente em chamadas de função em memória de sistemas centralizados. Apesar da desvantagem, a escolha por microsserviços em cenários críticos de fraude justifica-se pela priorização da resiliência e tolerância a falhas desse tipo de sistema. Conforme destaca Newman, a capacidade de escalar componentes de forma independente e garantir que a indisponibilidade de um serviço periférico não interrompa o fluxo crítico é vital para a alta disponibilidade exigida pelo setor financeiro (NEWMAN, 2015). O desafio técnico, portanto, desloca-se para a mitigação dessa latência através do uso de protocolos de comunicação de alto desempenho.

Adicionalmente, enfrenta-se um problema conhecido como *Data Drift,* que ocorre quando as propriedades estatísticas dos dados de entrada mudam em relação aos dados nos quais o modelo foi treinado, levando a uma degradação inevitável da acurácia preditiva ao longo do tempo (GOMEDE, 2023). No contexto de fraudes, o desafio da mudança nos padrões dos dados surge quando criminosos alteram suas táticas dinamicamente para evadir regras de segurança, tornando os modelos de *Machine Learning* obsoletos mais rapidamente. Consequentemente, a ausência de estratégias de manutenção contínua de modelos mantém o sistema exposto à obsolescência e à vulnerabilidades operacionais.

A mitigação dessas vulnerabilidades exige a implementação de processos automatizados de ciclo de vida do modelo. A adoção de princípios de *Machine Learning Operations* (*MLOps*) é fundamental para transpor a barreira entre a experimentação e a produção, permitindo o retreinamento, o monitoramento e a implantação contínua de modelos de forma confiável e escalável (KREUZBERGER et al., 2023).

Complementarmente, para viabilizar o treinamento inicial do sistema sem infringir normas de sigilo bancário e proteção de dados, adotou-se a geração de dados sintéticos. Esta estratégia, fundamentada por Lopez-Rojas, Elmir e Axelsson (2016), permite a simulação de padrões comportamentais fidedignos para a validação da arquitetura e do fluxo de *MLOps*, contornando a indisponibilidade de *datasets* financeiros reais com dados sensíveis.

No que tange à abordagem algorítmica, optou-se pelo aprendizado supervisionado em detrimento de métodos não-supervisionados. Embora a detecção de anomalias não-supervisionada seja útil para identificar fraudes inéditas, ela frequentemente resulta em altas taxas de falsos positivos, o que impacta negativamente a experiência do usuário. Conforme demonstrado por Zhang et al. (2020), a aplicação do modelo XGBoost na detecção de fraudes

transacionais apresenta desempenho superior em métricas de precisão e acurácia quando comparado a outros algoritmos clássicos, sendo a escolha ideal para maximizar a detecção correta minimizando bloqueios indevidos.

Para fortalecer a tomada de decisão do modelo, a lógica de classificação fundamenta-se na heurística do *Bayes Minimum Risk,* ou risco mínimo de Bayes. Em ambientes financeiros, o custo dos erros não é uniforme: deixar passar uma fraude de alto valor (Falso Negativo) acarreta prejuízos muito superiores ao custo operacional de bloquear indevidamente uma transação legítima (Falso Positivo). Conforme proposto por Bahnsen et al. (2013), esta abordagem introduz a sensibilidade ao custo na detecção de fraudes, permitindo que o limiar de decisão seja ajustado dinamicamente com base no valor monetário de cada transação. Dessa forma, o sistema prioriza a minimização das perdas financeiras reais da instituição, transcendendo a simples otimização de métricas estatísticas puras.

1.1 Trabalhos relacionados

A literatura apresenta diversas abordagens para o problema. Jurgovsky et al. (2018) propuseram o uso de redes neurais *Long Short Term Memory* (LSTM), obtendo alta acurácia na detecção de padrões complexos. Em contrapartida, Carcillo et al. (2018) exploraram o uso de arquiteturas de *Big Data* para o processamento de eventos em larga escala.

Por fim, os desafios operacionais do ciclo de vida dos modelos foram discutidos por Paleyes et al. (2022), que destacam a importância do *MLOps*, porém alertam que muitas implementações atuais ainda permanecem fortemente atreladas a soluções proprietárias de computação em nuvem, limitando a flexibilidade arquitetural. Este trabalho diferencia-se ao propor uma arquitetura agnóstica de nuvem, baseada em contêineres e tecnologias *open-source* consolidadas, unindo a performance da Engenharia de Software com a adaptabilidade da Inteligência Artificial.

1.2 Organização do trabalho

O restante deste artigo está organizado da seguinte forma: A seção 2 apresenta o Desenvolvimento, que engloba a Fundamentação Teórica e a Metodologia, detalhando os conceitos de microsserviços, as ferramentas, o cenário de aplicação, a arquitetura proposta e as etapas de implementação. A Seção 3 apresenta os Resultados e Discussão, com a análise dos

dados obtidos nos experimentos de carga. Por fim, a Seção 4 traz as Considerações Finais e sugestões para trabalhos futuros.

2 DESENVOLVIMENTO

Esta seção apresenta o embasamento teórico necessário para a compreensão das decisões de arquitetura adotadas, bem como o detalhamento metodológico da construção do artefato proposto. A exposição está dividida em uma fundamentação teórica, que aborda os conceitos de microsserviços, protocolos de comunicação e algoritmos de aprendizado de máquina, e a metodologia, que descreve as ferramentas e etapas de implementação.

2.1 Fundamentação teórica

A construção de sistemas resilientes para o setor financeiro exige a integração de conceitos de sistemas distribuídos e ciência de dados. A seguir, detalham-se os pilares teóricos que sustentam este trabalho.

2.1.1 Pilares tecnológicos e conceituais

A arquitetura de microsserviços é definida por Newman (2015) como uma abordagem que desenvolve uma aplicação única como um conjunto de pequenos serviços, cada um executando em seu próprio processo e comunicando-se através de mecanismos leves. Diferente dos monólitos, onde a falha em um módulo pode comprometer todo o sistema, os microsserviços permitem o isolamento de falhas. Em cenários de fraude, essa característica é crítica: a indisponibilidade do serviço de auditoria, por exemplo, não deve impedir a autorização de uma transação legítima no orquestrador.

A latência de rede é o principal desafio em sistemas distribuídos. Enquanto o padrão REST (*Representational State Transfer*), segundo Fielding (2000), utiliza JSON (2017) sobre HTTP/1.1, o gRPC (*Google Remote Procedure Call*) utiliza o protocolo HTTP/2 e serialização binária via *Protocol Buffers*. Segundo Indrasiri e Kuruppu (2020), essa combinação permite uma transmissão de dados significativamente mais compacta e rápida, reduzindo o *overhead* de processamento e o uso de banda, sendo ideal para a comunicação síncrona crítica entre o orquestrador e o serviço de inferência.

Sobre o processamento de eventos e consistência eventual, para operações que não exigem resposta imediata ao usuário, como o registro de auditoria, utiliza-se o modelo orientado a eventos. O Apache Kafka (KREPS et al., 2011) atua como um *log* de *commit* distribuído, garantindo que mensagens sejam persistidas mesmo se o consumidor estiver *offline*. Esta abordagem implementa o conceito de consistência eventual, onde o sistema prioriza a disponibilidade e a partição da rede em detrimento da consistência imediata dos dados não críticos, conforme o Teorema CAP (KLEPPMANN, 2017).

O *eXtreme Gradient Boosting* (XGBoost) é um algoritmo de aprendizado de máquina baseado em árvores de decisão. De acordo com Chen e Guestrin (2016), criadores do algoritmo, o XGBoost destaca-se pela sua velocidade de execução e capacidade de lidar com dados esparsos, tornando-o ideal para problemas de classificação em dados tabulares estruturados, como é o caso de históricos transacionais bancários.

2.2 Metodologia

Para atingir os objetivos propostos, este trabalho adotou a metodologia *Design Science Research* (DSR), caracterizada pela construção e avaliação de um artefato tecnológico (WIERINGA, 2014). A pesquisa classifica-se como aplicada e quantitativa pois a validação do artefato proposto fundamenta-se na coleta e análise estatística de dados numéricos obtidos em ambiente experimental controlado. A eficácia da arquitetura foi aferida por meio de métricas objetivas de desempenho computacional, como carga, latência e vazão, enquanto a adequação do algoritmo preditivo foi fundamentada na literatura especializada.

2.2.1 Ferramentas e tecnologias

A seleção da *stack* tecnológica priorizou soluções *open-source* consolidadas:

- Docker: Para orquestração e isolamento do ambiente (MERKEL, 2014).
- Java/Spring Boot (WebFlux): Para microsserviços de I/O intensivo (WALLS, 2016).
- Python: Para o Serviço de Inferência e *scripts* de *MLOps*, devido ao suporte nativo a bibliotecas de dados e aprendizado de máquina (VAN ROSSUM; DRAKE, 2009).
- gRPC: Para comunicação de alto desempenho entre microsserviços (INDRASIRI; KURUPPU, 2020).
- Apache Kafka: Para desacoplamento e resiliência via mensageria assíncrona (NARKHEDE; SHAPIRA; PALINO, 2017).

- Redis: Banco de dados em memória, utilizado para armazenamento volátil de alta velocidade (CARLSON, 2013).
- PostgreSQL: Sistema gerenciador de banco de dados relacional (SGBD) utilizado para persistência estruturada (OBE; HSU, 2017).
- Metabase: Ferramenta de *Business Intelligence* (BI) utilizada para a criação de *dashboards* analíticos, permitindo que analistas de negócio visualizem os dados de auditoria e identifiquem novos padrões de fraude (METABASE, 2025).

A arquitetura proposta une a robustez do Java com a capacidade analítica do Python, sustentada por um ecossistema focado em desempenho e escalabilidade. O uso de gRPC e Apache Kafka otimiza, respectivamente, a comunicação direta e o processamento assíncrono entre os serviços, garantindo agilidade e desacoplamento.

2.2.2 Cenário de aplicação, integração e implantação

O artefato desenvolvido não se destina ao usuário final (pessoa física), mas sim a atuar como um *middleware* de segurança em um modelo de negócio *Business-to-Business* (B2B). Conforme definido por Turban et al. (2018), este modelo caracteriza-se por transações comerciais eletrônicas realizadas entre organizações, onde o volume financeiro e a exigência de integração superam o varejo tradicional *Business-to-Consumer* (B2C). O cenário operacional previsto consiste na integração deste sistema à esteira de pagamento de empresas contratantes, como lojas virtuais ou processadoras de cartão de crédito. Neste contexto, o sistema atua como uma ferramenta auxiliar na tomada de decisão da empresa em relação à aceitação ou rejeição de transações. O fluxo operacional inicia-se quando o sistema da empresa contratante envia os dados da transação para o *API Gateway* da solução proposta. O sistema processa a inferência internamente, isolando toda a complexidade de comunicação entre os microsserviços e retorna uma recomendação de ação (*APPROVE*, *DECLINE* ou *REVIEW*) juntamente com um *score* de risco, permitindo que o contratante decida se efetiva a cobrança.

No que tange ao modelo de implantação, o sistema adota a estratégia de hospedagem própria. O artefato de *software* é entregue à empresa contratante na forma de imagens de *containers* Docker e arquivos de orquestração. Esta abordagem agnóstica à infraestrutura permite que o sistema seja implantado tanto em servidores físicos locais (*On-Premises*), comum em instituições financeiras com rígidos requisitos de *compliance* de dados, quanto em provedores de nuvem pública ou privada, utilizando Servidores Privados Virtuais (VPS) ou *clusters* gerenciados. A responsabilidade pelo provisionamento, custos e manutenção da

infraestrutura computacional recai sobre a contratante, garantindo que os dados sensíveis das transações jamais trafeguem por ambientes fora do controle da instituição financeira.

2.2.3 Arquitetura proposta

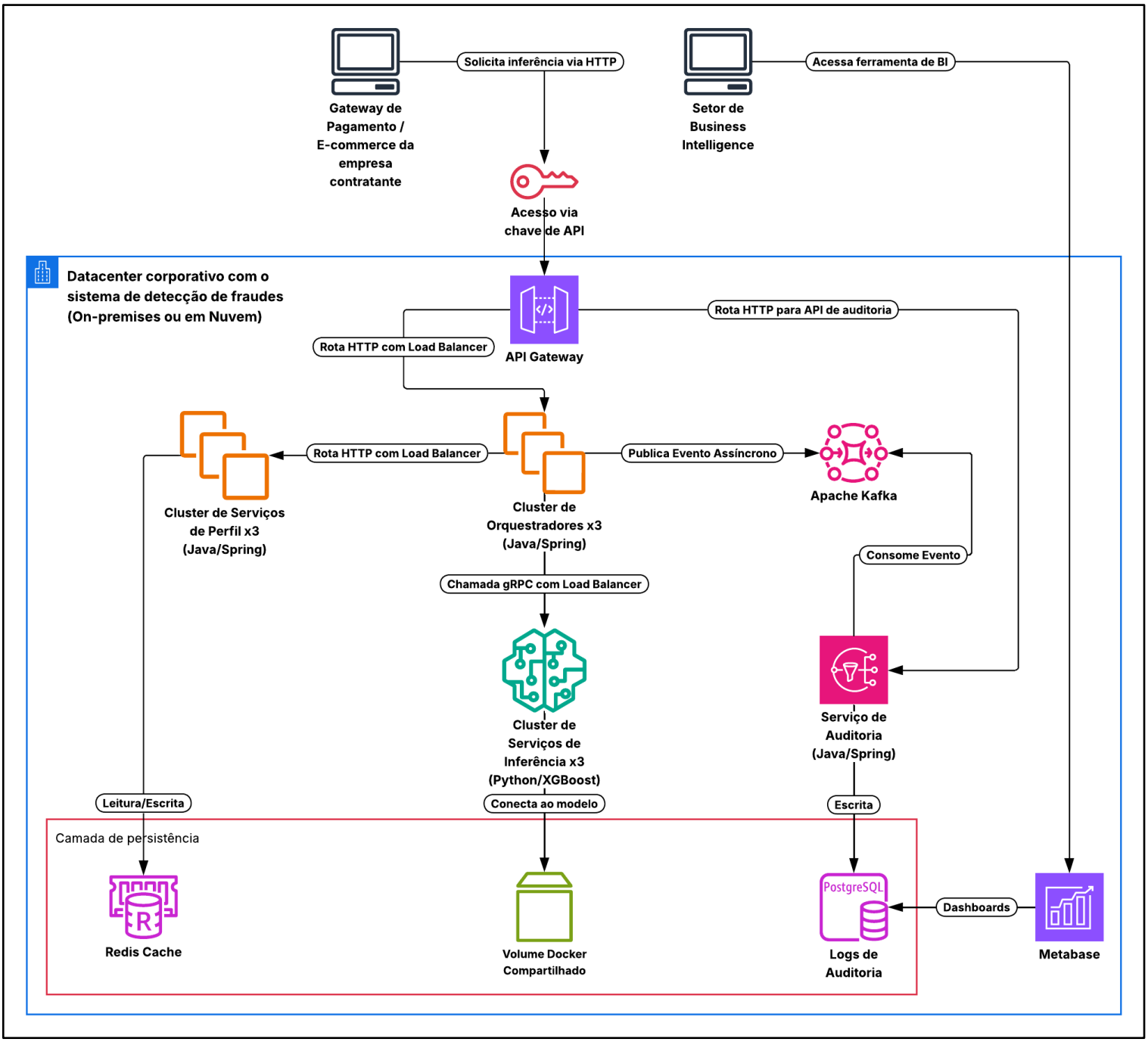
O artefato consiste em uma arquitetura híbrida composta por microsserviços interconectados, projetada para alta disponibilidade e resiliência. A solução organiza-se em dois planos de acesso distintos: o plano transacional, gerenciado pelo *API Gateway* para tráfego de alta frequência, e o plano analítico, destinado ao acesso interno para inteligência de negócios.

Para garantir a capacidade de processamento concorrente e tolerância a falhas, os serviços críticos, como o Orquestrador e os serviços de Perfil e Inferência, foram implantados em *clusters* de três réplicas cada, operando em paralelo.

A comunicação e a distribuição de tráfego entre esses componentes não dependem de balanceadores de carga externos ou servidores de descoberta de serviços. Em vez disso, optou-se pela utilização do balanceador de carga interno do ecossistema Spring, o *Spring Cloud LoadBalancer* (SPRING, 2025). Tanto o *API Gateway* quanto o Orquestrador utilizam listas estáticas de endereços de rede, mapeadas diretamente para os nomes de *host* dos *containers* Docker das réplicas correspondentes, realizando o balanceamento no lado do cliente.

Complementarmente, a arquitetura integra o Metabase (2025) como interface visual conectada ao banco de dados de auditoria. Este componente desempenha um papel estratégico ao democratizar o acesso aos dados, servindo como ponto de convergência entre o setor tecnológico e o setor financeiro/negócios. Por meio dele, analistas podem monitorar KPIs (*Key Performance Indicators*) de fraude em tempo real e realizar a triagem de casos suspeitos para rotulagem (*labeling*), alimentando o ciclo de retreinamento do modelo.

Figura 1 - Diagrama de Arquitetura do Sistema.



Fonte: Autoria própria (2025).

A seguir, detalham-se as responsabilidades de cada componente:

- *Datacenter* corporativo: Representa a fronteira de infraestrutura lógica e física onde o sistema de detecção de fraudes reside. A arquitetura foi projetada para ser agnóstica quanto ao modelo de hospedagem, suportando implantação tanto em servidores físicos locais (*On-Premises*) quanto em nuvem privada, garantindo o isolamento necessário para a proteção de dados sensíveis.

- *Gateway* de pagamento / *E-commerce*: Refere-se aos sistemas externos da empresa contratante que atuam como clientes da solução. São responsáveis por originar as requisições de análise de risco no momento do *checkout*, enviando os dados da transação via HTTP/REST e aguardando a recomendação de segurança para efetivar a cobrança.
- *API Gateway*: O ponto de entrada do sistema, responsável pela segurança através da autenticação via chave de API e pelo roteamento das requisições. O componente atua como a interface de integração B2B (*Business-to-Business*), recebendo chamadas exclusivamente dos sistemas de *E-commerce* ou *Gateways* de Pagamento da empresa contratante. Internamente, o *Gateway* distribui as requisições recebidas entre as três réplicas do Orquestrador utilizando uma estratégia de *Round-Robin* baseada na lista estática de endereços configurada na aplicação, como também expõe de forma segura os *endpoints* da API do Serviço de Auditoria, permitindo que sistemas externos ou scripts de automação consultem *logs* e submetam rótulos para o retreinamento.
- Orquestrador: Composto por três réplicas, este serviço coordena o fluxo transacional. Ele atua também como um cliente balanceado: ao necessitar de dados históricos ou de uma inferência, o Orquestrador utiliza seu balanceador interno para distribuir as chamadas HTTP e gRPC, respectivamente, entre as réplicas do Serviço de Perfil e do Serviço de Inferência.
- Serviço de Perfil: Composto por três réplicas, este serviço é responsável por fornecer o histórico de comportamento do usuário, como a contagem de transações, média de valores e última localização, com baixa latência, utilizando o Redis como camada de persistência em memória compartilhada entre as instâncias.
- Serviço de Inferência: Composto por três réplicas desenvolvidas em Python, este componente hospeda o modelo XGBoost. A comunicação com o Orquestrador é realizada via gRPC para máxima performance. As réplicas compartilham um volume de disco para permitir a atualização síncrona do modelo após o retreinamento.
- Serviço de Auditoria: Responsável pela persistência assíncrona de todas as inferências tomadas pelo sistema. Atua como consumidor de mensagens do Apache Kafka, gravando os eventos de decisão no banco de dados PostgreSQL, garantindo que o *overhead* de escrita em disco não impacte a latência da resposta de aprovação/rejeição da transação. Para garantir a integridade dos registros, foi implementado um mecanismo de assinatura digital onde cada entrada de *log* é processada pelo algoritmo de *hashing* criptográfico SHA-256 (2015). Essa abordagem assegura que qualquer tentativa

posterior de adulteração dos dados no banco possa ser detectada matematicamente. Também gera relatórios de segurança ao detectar alguma transação crítica de alta prioridade, os quais são exibidos no *console* da aplicação, podendo ser integrado futuramente à *webhooks* externos, permitindo o envio automático de notificações para canais de comunicação corporativos, como WhatsApp ou Discord, alertando proativamente o setor de negócios.

- Setor de *Business Intelligence* (BI): Representa a equipe de analistas e cientistas de dados da organização. Este grupo utiliza os *dashboards* do Metabase para monitorar KPIs de fraude, investigar falsos positivos e gerar inteligência para o negócio, sem interferir no fluxo transacional automatizado. Crucialmente, os *insights* gerados nesta etapa podem ser compartilhados com a equipe responsável pelo retreinamento do modelo, retroalimentando o *pipeline* de *MLOps* com novos dados rotulados e garantindo a adaptação contínua do sistema às estratégias de ataque emergentes.
- Metabase: Ferramenta de *Business Intelligence* conectada diretamente à réplica de leitura do banco de dados de auditoria. Diferente dos demais componentes, o Metabase não é acessado através do *API Gateway*, mas sim via acesso direto restrito à rede interna (*Intranet*) ou via rede virtual privada (VPN) corporativa. Essa segregação arquitetural impede que consultas analíticas complexas e pesadas consumam recursos de rede do *Gateway*, preservando a latência das transações financeiras críticas.

Em síntese, a arquitetura apresentada orquestra tecnologias heterogêneas para conciliar os rigorosos requisitos não-funcionais de latência e disponibilidade com as necessidades estratégicas de negócio. A segregação entre os planos transacional e analítico, aliada à redundância de componentes críticos e ao uso de protocolos de alto desempenho como gRPC, cria um ambiente resiliente capaz de suportar picos de tráfego sem degradação do serviço. Dessa forma, o desenho estrutural não apenas viabiliza a execução técnica do modelo preditivo em tempo real, mas também estabelece uma fundação segura e auditável que sustenta o ciclo contínuo de retroalimentação entre a inteligência humana e o aprendizado de máquina.

2.2.4 Etapas de desenvolvimento

A materialização da proposta arquitetural seguiu um roteiro metodológico estruturado, desenhado para reduzir a complexidade inerente à integração entre componentes de engenharia de software e modelos de aprendizado de máquina. O ciclo de vida do desenvolvimento foi

organizado de maneira sequencial e iterativa, garantindo que cada módulo fosse validado individualmente antes de sua integração ao ecossistema distribuído.

O processo de construção foi subdividido em cinco etapas incrementais:

1. Geração de dados: Criação de um algoritmo gerador de dados sintéticos para simular padrões de fraude e permitir o treinamento inicial do modelo de inferência.
2. Treinamento do modelo: Implementação do modelo XGBoost e da lógica de decisão baseada no risco mínimo de Bayes, estabelecendo o núcleo de inteligência preditiva do sistema.
3. Implementação dos componentes distribuídos: Desenvolvimento dos microsserviços em Java e Python, e configuração da comunicação híbrida REST e gRPC, consolidando a arquitetura de baixa latência.
4. Pipeline de automação: Desenvolvimento dos *scripts* para o *pipeline* de *MLOps* para retreinamento com recarregamento rápido (*hot reload*).
5. Experimentação: Execução de testes de carga com Apache JMeter (2024) para validar a vazão, latência e resiliência da arquitetura em cenários de alta concorrência.

Essa abordagem em camadas permitiu o isolamento de responsabilidades e a detecção de gargalos. Enquanto as primeiras etapas focaram na eficiência matemática e na precisão da detecção, as fases subsequentes priorizaram a eficiência computacional e a robustez do sistema, visando garantir os requisitos não funcionais de sistemas distribuídos discutidos na fundamentação teórica.

2.2.4.1 Geração de dados

Devido à natureza sensível dos dados bancários e às restrições impostas por leis de proteção de dados, como a LGPD (Lei Geral de Proteção de Dados Pessoais), o acesso a *datasets* públicos de transações financeiras com qualidade e atualidade é escasso. Para viabilizar o treinamento inicial do modelo e a validação do fluxo de *MLOps*, desenvolveu-se um algoritmo gerador de dados sintéticos utilizando a linguagem Python e as bibliotecas NumPy (2025) e Pandas (2025).

O algoritmo foi projetado para produzir um conjunto de dados com 100.000 amostras, simulando o comportamento transacional de usuários. Para garantir a similaridade estatística com dados financeiros reais, utilizou-se uma distribuição log-normal para a geração dos valores monetários. Essa escolha justifica-se pelo fato de que valores de transações financeiras são estritamente positivos e tendem a apresentar uma distribuição assimétrica à direita, onde a

maioria das compras possui valores baixos ou médios, e poucas transações possuem valores muito altos, padrão este observado em simuladores como o PaySim (LOPEZ-ROJAS et al., 2016).

A estratégia de rotulagem não foi aleatória. O algoritmo implementou a injeção deliberada de três padrões comportamentais distintos para ensinar ao modelo o que constitui uma fraude ou uma transação legítima:

1. Padrão do cliente legítimo: Representando 30% da amostra, foram gerados perfis com histórico de transações consolidado, entre 5 e 100 transações anteriores. Para estes casos, o valor da transação atual foi forçado a estar dentro de um desvio aceitável, entre 90% e 110% da média histórica do usuário, recebendo o rótulo de transação legítima.
2. Padrão de fraude por desvio de perfil: Representando cerca de 5% da amostra, foi simulado o cenário onde uma conta legítima é comprometida ou o cartão é clonado. Nestes casos, valores de transações abruptamente superiores à média histórica do usuário foram gerados, entre 5 a 10 vezes maior, recebendo o rótulo de fraude.
3. Padrão de fraude em contas novas: Simulando o ataque de criação de contas falsas ou uso imediato de cartões roubados sem histórico prévio, definiu-se uma regra onde transações sem histórico e com valor elevado, superior a 800, fossem automaticamente marcadas como fraude, garantindo uma cautela maior na inferência.

Além das variáveis numéricas, introduziu-se a variável categórica de localização, atribuindo uma probabilidade de 5% para transações internacionais, adicionando complexidade ao espaço de características que o modelo deveria interpretar.

Vale ressaltar que os valores adotados são independentes de moeda, tornando as taxas e conversões de câmbio internacionais responsabilidade do sistema externo de pagamento da empresa contratante.

Ao final do processamento, o *dataset* resultante foi dividido em conjuntos de treinamento (80%) e teste (20%) utilizando amostragem estratificada para manter a proporção original de fraudes em ambos os grupos, garantindo a integridade da avaliação do modelo na etapa seguinte.

2.2.4.2 Treinamento do modelo

Com o *dataset* preparado, procedeu-se ao treinamento do modelo utilizando o algoritmo XGBoost. A calibração do modelo seguiu as diretrizes experimentais estabelecidas por Zhang et al. (2020) para detecção de fraudes transacionais. Buscando maximizar a acurácia em um

cenário de alta complexidade, configurou-se o algoritmo com 5.000 árvores de decisão associadas a uma taxa de aprendizado conservadora de 0.02. Essa combinação permite que o modelo convirja para uma solução robusta de forma gradual, reduzindo o risco de *overfitting* comum em taxas de aprendizado mais agressivas.

A profundidade máxima das árvores foi fixada em 12, permitindo a captura de correlações não-lineares profundas entre as variáveis do perfil do usuário. Para garantir a regularização e a diversidade do *ensemble*, aplicou-se a amostragem estocástica (FRIEDMAN, 2002), utilizando apenas 80% das amostras e 40% das *features* na construção de cada árvore. Por fim, visando a eficiência computacional exigida pelos requisitos de tempo real, optou-se pelo método de construção baseado em histograma, que otimiza o uso de memória e acelera tanto o treinamento quanto a inferência. Após a definição dos hiperparâmetros, o modelo foi treinado resultando na serialização do artefato final no formato JSON. A opção por este formato aberto visa garantir a interoperabilidade e permitir a auditabilidade da estrutura de árvores de decisão.

Embora o modelo XGBoost forneça uma estimativa probabilística robusta baseada em padrões estatísticos, a arquitetura implementa uma camada intermediária de regras heurísticas para incorporar conhecimento de domínio determinístico e mitigar limitações de inferência em casos de borda. Esta lógica atua sobre a probabilidade bruta retornada pelo modelo, aplicando penalidades ou bonificações baseadas no comportamento histórico do usuário.

Foram definidas duas regras principais de ajuste:

1. Bonificação por consistência comportamental: Parte-se da premissa de que transações legítimas tendem a manter uma coerência de valor em relação ao histórico do cliente. O algoritmo verifica se o valor da transação atual se encontra dentro de um intervalo de desvio de mais ou menos 20% em relação à média histórica do usuário. Caso positivo, e havendo histórico prévio, a pontuação de risco total é reduzida pela metade, diminuindo a incidência de Falsos Positivos em compras rotineiras.
2. Penalidade por dissonância geográfica: Para endereçar o risco de credenciais comprometidas ou clonagem de cartões em localidades distintas, implementou-se uma verificação de salto geográfico. O sistema compara o país da transação atual com o país da última transação registrada. Havendo divergência em um usuário com histórico consolidado, aplica-se uma penalidade aditiva absoluta de +0.30 à probabilidade de fraude. Essa medida força o *score* para cima, aumentando a probabilidade de que a transação ultrapasse o limiar de corte e seja encaminhada para revisão ou bloqueio, mesmo que o modelo estatístico não tenha detectado anomalias nos dados numéricos.

Como mencionado na introdução, foi implementada uma camada de decisão como pós-processamento fundamentada na heurística do risco mínimo de Bayes. Diferente de classificadores tradicionais que buscam minimizar a taxa de erro global, esta implementação busca minimizar a esperança matemática da perda financeira. Esse conceito, oriundo da teoria da decisão, representa o custo médio que se espera incorrer ao tomar uma determinada ação, ponderado pela probabilidade de cada cenário possível. A lógica parte da premissa de que a matriz de custos em fraudes é assimétrica. Definiu-se o Custo do Falso Negativo (CFN) — prejuízo gerado ao aprovar uma fraude — como sendo igual ao valor monetário da transação (V). Em contrapartida, o Custo do Falso Positivo (CFP) — prejuízo operacional ao bloquear um cliente legítimo — foi fixado em uma constante (definida no código como 200.0), representando custos administrativos e de fricção com o cliente.

Diante de uma nova transação com probabilidade de fraude Pf (fornecida pelo XGBoost), o sistema calcula dois riscos concorrentes:

1. Risco esperado da aprovação (*Raprov*): É o produto da probabilidade de ser fraude pelo valor a ser perdido (Pf × V).
2. Risco esperado do bloqueio (*Rbloq*): É o produto da probabilidade de ser legítimo pelo custo operacional fixo ((1 − Pf) × 200).

A decisão de bloquear ocorre estritamente quando o risco financeiro de deixar passar a transação supera o custo operacional de interrompê-la (*Raprov* > *Rbloq*). A implementação traduz essa desigualdade no cálculo de um limiar de decisão dinâmico (T) para cada transação. Conforme a dedução matemática estabelecida por Elkan (2001) para classificação sensível ao custo, o limiar ótimo é dado por:

$$T = \frac{CFP}{CFN + CFP} = \frac{200}{V + 200} \quad (1)$$

Onde V é o valor da transação. O sistema compara a probabilidade de fraude (Pf) fornecida pelo XGBoost com este limiar T:

- Se Pf > T, a transação é considerada de risco financeiro inaceitável.
- Se Pf ≤ T, o risco é considerado tolerável dado o valor envolvido.

Na prática, isso resulta em um comportamento adaptativo: para transações de baixo valor, por exemplo 10.00, o limiar de corte é alto, aproximadamente 0.95, exigindo que o modelo tenha extrema certeza para bloquear. Para transações de alto valor, por exemplo 5000.00, o limiar cai drasticamente para aproximadamente 0.03, fazendo com que mesmo uma

leve suspeita leve a transação para revisão, protegendo o capital da empresa. Adicionalmente, foram estabelecidos limites de segurança para probabilidades extremas (> 0.90 ou < 0.15) para garantir decisões rápidas em casos óbvios, independentemente do valor.

Um aspecto arquitetural crítico implementado nesta etapa foi a estratégia de armazenamento do artefato treinado. Para suportar a arquitetura de alta disponibilidade com múltiplas réplicas, o modelo não é encapsulado isoladamente dentro do sistema de arquivos efêmero de cada contêiner. Em vez disso, estabeleceu-se um volume Docker compartilhado, mapeado do sistema hospedeiro para o diretório interno de todas as instâncias do serviço.

Esta configuração de volume compartilhado desempenha uma função dupla essencial para o fluxo de *MLOps*:

1. Consistência de estado: Garante que as três réplicas do *cluster* de inferência leiam exatamente a mesma versão física do arquivo, assegurando uniformidade nas decisões de risco. Uma mesma transação receberá o mesmo *score*, independentemente de qual réplica a processe.
2. Atualização atômica: Permite que o *script* de retreino atualize o arquivo JSON do modelo no volume Docker compartilhado, disponibilizando a nova versão da inteligência para todo o *cluster* simultaneamente. Isso viabiliza o mecanismo de *hot reload*, onde as aplicações recarregam o arquivo da memória sem a necessidade de reiniciar os contêineres ou interromper o serviço.

2.2.4.3 Implementação dos componentes distribuídos

A terceira etapa consistiu na materialização da arquitetura proposta através do desenvolvimento e integração dos componentes isolados em um ecossistema distribuído.

Os componentes de I/O intensivo (*API Gateway*, Orquestrador e Serviços de Perfil e Auditoria) foram desenvolvidos utilizando Java e o *framework* Spring Boot com o módulo WebFlux. A escolha pelo modelo de programação reativa não-bloqueante fundamentou-se na necessidade de gerenciar um alto volume de conexões simultâneas com baixo consumo de *threads*, essencial para o cenário de alta concorrência visado.

Implementou-se um filtro global personalizado no *API Gateway* para interceptar todas as requisições de entrada e re-encaminhá-las aos serviços correspondentes, que são o *Cluster* de Orquestradores e o Serviço de Auditoria. Este componente valida a presença e a integridade da chave de acesso à API no cabeçalho HTTP antes de rotear o tráfego, garantindo que apenas clientes autorizados acessem os serviços. Adicionalmente, configurou-se o *Spring Cloud*

Gateway para realizar o balanceamento de carga no lado do cliente, distribuindo as chamadas entre as três réplicas do Orquestrador.

O serviço Orquestrador foi construído como o *hub* de integração, executando um *pipeline* de decisão em três estágios para cada transação. Primeiro, realiza o enriquecimento dos dados consultando o Serviço de Perfil via cliente HTTP reativo. Em seguida, executa a ponte reativa-bloqueante para comunicar-se com o Serviço de Inferência via gRPC. Para isso foram gerados os *stubs* a partir do contrato .proto definido. Um desafio técnico superado nesta etapa foi o encapsulamento da chamada gRPC, que é bloqueante por natureza, dentro de *wrappers* reativos executados em *threads* dedicadas, evitando o travamento do *event loop* principal da aplicação reativa. Por fim, realiza a publicação assíncrona do resultado no Kafka, desacoplando a resposta ao cliente do processo de persistência.

O Serviço de Perfil foi implementado com acesso direto ao Redis através de templates reativos, garantindo latência de leitura na ordem de sub-milissegundos. Foi implementado também um ouvinte do Kafka para atualizar assincronamente as estatísticas do usuário, como contagem de transações e a média após cada transação, assegurando a consistência eventual dos dados sem penalizar o tempo de resposta da inferência.

O Serviço de Auditoria foi configurado como um consumidor do tópico Kafka. Além da persistência dos *logs* no PostgreSQL, desenvolveram-se *endpoints* REST administrativos para permitir consultas paginadas e, crucialmente, a rotulagem das transações por analistas humanos, funcionalidade que é pré-requisito para o ciclo de aprendizado de máquina. Para rastreabilidade e investigação, o *endpoint GET /audit* permite que a equipe de fraude visualize todo o *log* de transações. Para otimizar a rotina dos analistas de risco, a API oferece filtros de estado. O *endpoint GET /audit/declined* isola as transações recusadas para revisão prioritária, enquanto o *GET /audit/pending* cria uma fila de trabalho com transações que ainda aguardam rotulagem. O ciclo de retroalimentação do modelo é viabilizado pelo *endpoint* de escrita parcial *PATCH /audit/label*, que permite atribuir o rótulo real, como "*FRAUD*" ou "*LEGITIMATE*", a uma transação processada.

Complementarmente, o *endpoint GET /audit/labeled* atua como fonte de dados para o *script* de retreinamento, fornecendo apenas o *dataset* classificado, essencial para a evolução da precisão do algoritmo XGBoost.

2.2.4.4 Pipeline de automação

Paralelamente à infraestrutura transacional, foi implementado o *pipeline* de *Machine Learning Operations* (*MLOps*), materializado através de *scripts* de automação em Python responsáveis pelo ciclo de vida do modelo.

O núcleo dessa automação executa um fluxo sequencial de quatro passos críticos para garantir a evolução segura do sistema:

1. Extração e curadoria: O *script* conecta-se ao banco de dados PostgreSQL e extrai apenas as transações que possuem um rótulo confirmado por analistas.
2. Verificação de segurança: Antes de iniciar o treinamento, implementou-se um mecanismo de defesa contra o problema de *overfitting*. O algoritmo verifica se o volume de novos dados rotulados atinge um limiar estatístico mínimo de 1000 entradas. Caso a amostra seja insuficiente, o processo é abortado, preservando a versão estável do modelo anterior e evitando a degradação da acurácia.
3. Treinamento e persistência: Havendo dados suficientes, o algoritmo treina uma nova versão do classificador XGBoost e sobrescreve o arquivo JSON no volume compartilhado, disponibilizando o novo modelo instantaneamente para todo o *cluster*.
4. Sincronização de réplicas: Após a gravação em disco, o *script* executa uma rotina de gatilho que itera sobre a lista de endereços das réplicas do serviço de inferência. É enviado um comando gRPC administrativo para cada instância, instruindo-as a descartar o modelo antigo residente em memória RAM e carregar a nova versão do disco.

Essa abordagem elimina a necessidade de reinicialização dos contêineres e garante que todas as réplicas operem com a mesma versão do modelo, assegurando a consistência das inferências em um ambiente de alta concorrência.

Por fim, a automação do ciclo de vida do modelo é consolidada através da capacidade de agendamento de tarefas. A execução periódica do *script* de retreino (por exemplo, via utilitários como *Cron* no sistema operacional hospedeiro) assegura que o modelo seja atualizado em intervalos regulares, garantindo a adaptação contínua a novos padrões de fraude sem a necessidade de intervenção manual direta para o disparo do processo.

2.2.4.5 Experimentação

A etapa final do desenvolvimento consistiu na validação experimental do artefato em um ambiente controlado, com o objetivo de submeter a arquitetura a diferentes níveis de estresse para aferir a eficiência da comunicação gRPC, a vazão de resposta do *API Gateway* e

a estabilidade do sistema sob carga. Para a execução dos testes, utilizou-se a ferramenta Apache JMeter operando em modo de linha de comando, estratégia que elimina a sobrecarga de processamento da interface gráfica e assegura maior precisão na coleta das métricas. O plano de testes foi configurado para simular o comportamento de clientes externos interagindo com a aplicação através de requisições HTTP do tipo POST para o *endpoint* de análise do *API Gateway*.

A configuração do experimento priorizou a fidelidade ao cenário real de uso. O componente *HTTP Header Manager* foi ajustado para injetar a chave de API em todas as requisições, validando o mecanismo de autenticação. Para garantir a variabilidade dos dados utilizou-se o componente *CSV Data Set Config* associado a um conjunto de dados sintéticos contendo 100 registros aleatórios. Durante a execução, as *threads* do JMeter realizam a leitura sequencial deste arquivo, injetando dinamicamente os campos de identificação do usuário, valor monetário e país de origem no corpo de cada requisição, forçando o processamento completo de cada transação como um evento único e não repetitivo.

Para garantir a reprodutibilidade dos resultados, o ambiente foi isolado e orquestrado via Docker Compose (DOCKER INC., 2025) em uma estação de trabalho dedicada, redefinindo todos os volumes para seu estado inicial entre as execuções de cada teste. Além da escolha de uma versão do sistema onde todos os *logs* foram omitidos. O *hardware* hospedeiro consistiu em um notebook VAIO FE15 equipado com processador AMD Ryzen 7-5700U (8 núcleos físicos e 16 *threads* lógicas, com frequência de *burst* de até 4,30 GHz), 16GB de memória RAM DDR4 3.200 MHz e armazenamento SSD NVMe. Durante os experimentos, o sistema operacional foi configurado em perfil de alto desempenho e isolado de processos concorrentes não essenciais, minimizando interferências externas na medição da latência.

A estratégia de validação foi estruturada em dois cenários complementares. O primeiro, focado na análise de escalabilidade e degradação, consistiu na execução de baterias de testes de 60 segundos com variação de concorrência em 4, 16 e 32 *threads* simultâneas. A escolha de 16 *threads* como ponto central visou alinhar a concorrência da aplicação com a capacidade lógica do processador, buscando medir a vazão máxima com o mínimo de *overhead* de troca de contexto. O segundo cenário, focado na estabilidade, manteve essa carga ideal por um período contínuo de 5 minutos para identificar potenciais vazamentos de memória ou exaustão de recursos de rede. Em todos os casos, configurou-se um período de subida (*ramp-up period*) de 0,001 segundos para as *threads* do JMeter. O objetivo dessa parametrização foi forçar a ativação quase instantânea de toda a concorrência planejada, simulando uma entrada abrupta de tráfego para desafiar a capacidade de reação imediata e o escalonamento dos microsserviços.

3 RESULTADOS E DISCUSSÃO

A validação da arquitetura proposta visou verificar se o sistema de microsserviços, operando com comunicação híbrida (REST/gRPC), é capaz de processar fluxos de análise de risco dentro dos requisitos de tempo real exigidos por operações B2B. Os dados a seguir apresentam o comportamento do sistema sob as diferentes cargas de trabalho definidas na metodologia.

3.1 Análise de escalabilidade e curva de degradação

O primeiro experimento buscou identificar o ponto ótimo de operação e o comportamento do sistema sob saturação. A Tabela 1 a seguir sumariza as métricas de vazão e latência média obtidas ao variar o número de usuários virtuais concorrentes em janelas de execução de 60 segundos.

Tabela 1 - Comparativo de desempenho por nível de concorrência durante 1 minuto.

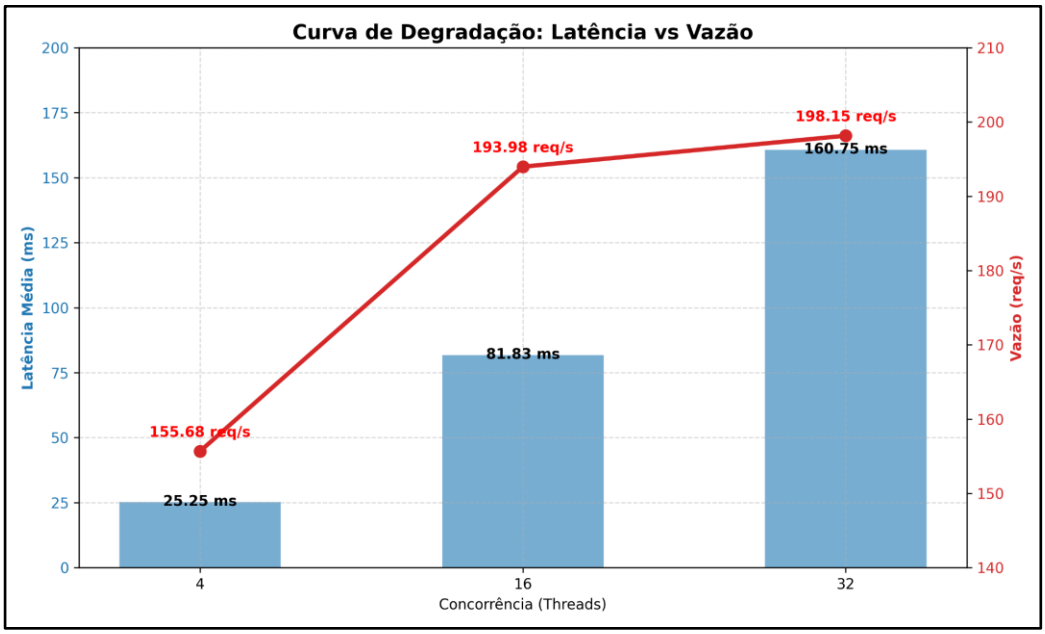
Concorrência	Total de Requisições	Vazão (req/s)	Latência Mínima (ms)	Latência Média (ms)	Latência Máxima (ms)	Erro (%)
4 *threads*	9.321	155,68	10	25,25	1.158	0
16 *threads*	11.613	193,98	20	81,83	1.619	0
32 *threads*	11.868	198,15	31	160,75	2.010	0

Fonte: Autoria própria (2025).

A análise conjunta dos dados tabulados e da curva de desempenho na Figura 2 revela três zonas distintas de operação da arquitetura. Inicialmente, no cenário de baixa concorrência com 4 *threads*, o sistema operou em subutilização, entregando a melhor latência média do experimento (25,25 milissegundos), porém, com uma vazão limitada a 155,68 requisições por segundo (req/s). Este comportamento indica a ociosidade dos recursos de computação paralela, uma vez que a carga de trabalho não foi suficiente para ocupar os *pipelines* de execução disponíveis nos múltiplos núcleos do processador (TANENBAUM, 2016). Ao alinhar a concorrência com a capacidade física do processador (16 *threads*), atingiu-se a zona de eficiência máxima: a vazão obteve um salto expressivo de 24,6% (193,98 req/s) e a latência

estabilizou-se em 81,83 ms, validando o dimensionamento da infraestrutura para operações em tempo real abaixo do limiar crítico de 100 ms.

Figura 2 - Curva de degradação: Latência média vs Vazão.



Fonte: Autoria própria (2025).

Já no cenário de 32 *threads*, o sistema entrou em zona de saturação. Observou-se que, embora a latência média tenha dobrado para 160,75 ms devido ao enfileiramento de processos pelo sistema operacional, o ganho de vazão foi marginal, de apenas 2,1%. Esse fenômeno confirma que a aplicação é limitada por processamento (*CPU-Bound*), onde o excesso de concorrência gera *overhead* de troca de contexto sem aumento real na capacidade produtiva. Conforme explica Tanenbaum e Bos (2016), à medida que o número de processos ativos excede o número de processadores físicos, o sistema operacional gasta ciclos de CPU significativos apenas para salvar e restaurar o estado dos registradores entre as trocas de tarefas, degradando o desempenho global.

Complementarmente, a análise da dispersão dos tempos de resposta oferece *insights* sobre a estabilidade operacional. Observou-se que os picos de latência máxima, que atingiram 2.010 ms, concentraram-se nos instantes iniciais da bateria de testes, caracterizando a fase de aquecimento. Sucedendo esse período de instabilidade inicial, o sistema atingiu o regime permanente, onde a latência mínima estabilizou-se em uma variação linear entre 10 ms e 31 ms. Esse comportamento evidencia fenômenos inerentes a aplicações Java, atribuídos majoritariamente ao *Garbage Collector* e à compilação JIT (*Just-In-Time*). Segundo Oaks (2020), a JVM (*Java Virtual Machine*) inicia a execução em modo interpretado e,

gradualmente, compila os trechos de código mais frequentes para código nativo otimizado; portanto, a redução da latência conforme o tempo de execução progride é o comportamento esperado após o aquecimento das caches e a otimização do código.

Por fim, o indicador mais crítico de robustez, a taxa de erros manteve-se em 0% em todos os cenários. Este resultado corrobora a eficácia do modelo de programação reativa não-bloqueante do *Spring WebFlux*. Diferente de arquiteturas bloqueantes que rejeitariam conexões ao esgotar o *pool* de *threads*, a solução proposta gerenciou o excesso de carga aceitando todas as 11.868 requisições no cenário de estresse, optando estrategicamente por degradar a latência em vez de comprometer a disponibilidade do serviço.

3.2 Análise de estabilidade

Complementarmente à análise de escalabilidade, o teste de estabilidade submeteu o sistema à carga ideal (16 *threads*) por um período contínuo de 5 minutos. Diferente do teste de estresse curto, onde o aumento de *threads* gerou gargalo de CPU, a extensão da janela de tempo neste cenário permitiu observar o comportamento da aplicação em regime permanente, onde otimizações dinâmicas da JVM e o aquecimento de *caches* têm impacto significativo.

A Figura 3 apresenta as estatísticas consolidadas geradas pelo Apache JMeter para este cenário.

Figura 3 - Estatísticas de desempenho para o teste de estabilidade (5 minutos).

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	120973	0	0.00%	39.39	8	1742	31.00	45.00	51.00	62.00	403.68	104.48	117.80

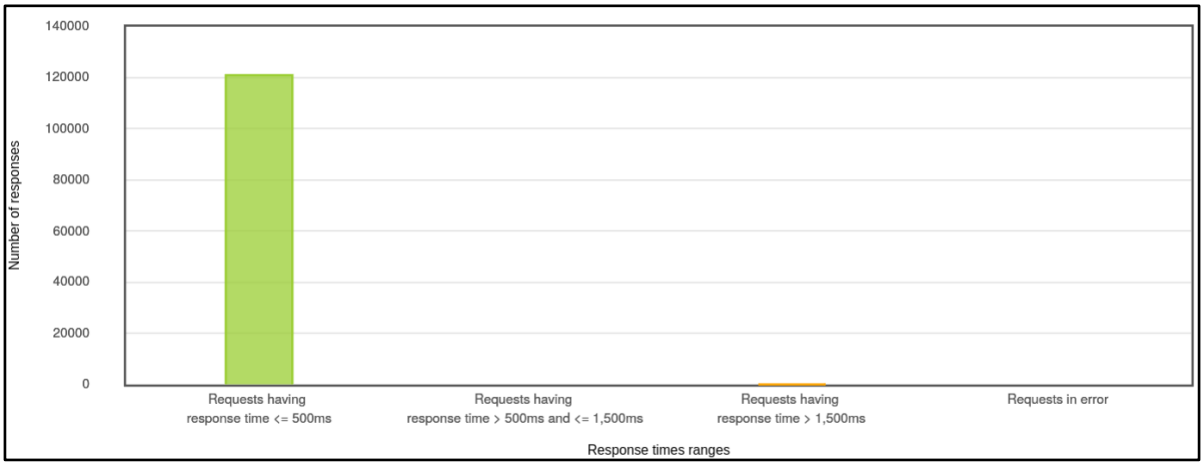
Fonte: Autoria própria (2025).

Uma comparação direta com o teste anterior de 60 segundos revela uma melhoria substancial na performance média. Enquanto o teste curto registrou uma latência média de 81,83 ms, o teste de longa duração convergiu para 39,39 ms. Essa redução de mais de 50% no tempo de resposta deve-se à maturação da execução: com o passar do tempo, o compilador do Java otimiza os caminhos de código mais frequentes para código de máquina nativo, e os dados dos usuários tornam-se disponíveis no *cache* Redis, permitindo que requisições subsequentes sejam atendidas com latência mínima de até 8 ms, sem a necessidade de processamento pesado.

A análise dos percentis reforça a consistência da arquitetura. O valor de P99 em 62 ms indica que 99% das 120 mil requisições foram processadas abaixo desse limiar, demonstrando

uma previsibilidade excelente para aplicações financeiras. O gráfico a seguir ilustra visualmente essa distribuição: a barra que representa requisições bem-sucedidas abaixo de 500 ms é predominantemente maior que as demais, confirmando que os casos de latências elevadas acima de 1.500 ms são eventos isolados, restritos aos segundos iniciais de aquecimento da aplicação.

Figura 4 - Visão geral dos tempos de resposta.



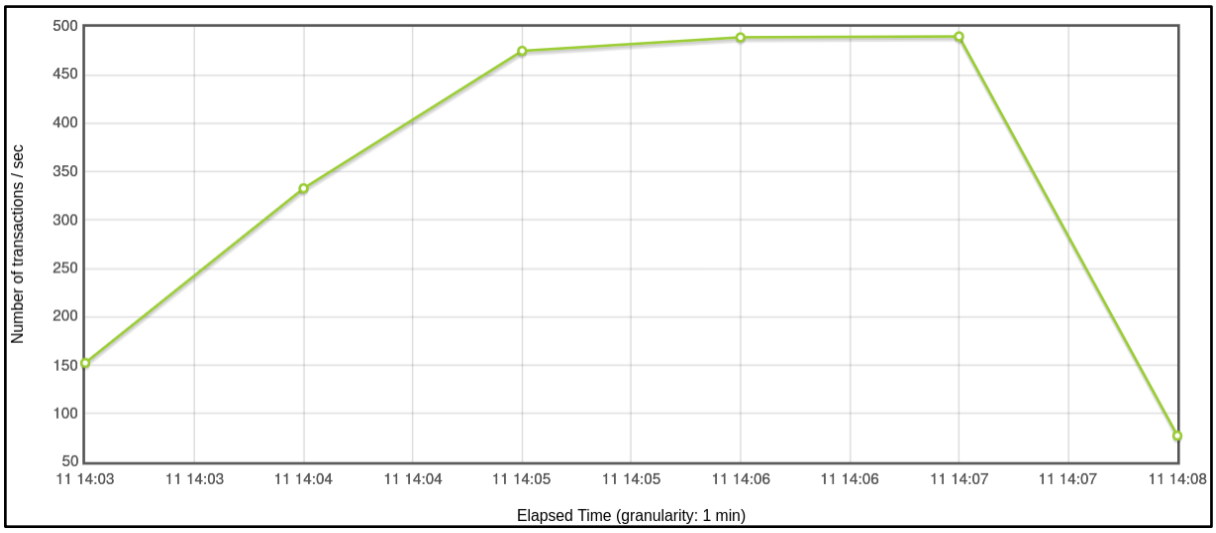
Fonte: Autoria própria (2025).

Em relação à confiabilidade, a validação foi conduzida em duas etapas para isolar gargalos de infraestrutura. Em uma execução preliminar, foram registradas 776 falhas em um total de 111.439 requisições, uma taxa de erro de 0,7%, diagnosticadas como "*java.net.SocketException: Connection reset*". A análise identificou a exaustão de portas efêmeras devido ao acúmulo de conexões no estado *TIME_WAIT*. Este estado é uma característica intrínseca do protocolo TCP, definida para garantir que pacotes antigos expirem na rede antes que a combinação de IP e porta seja reutilizada por uma nova conexão (STEVENS, 2011).

Após a mitigação desta limitação ambiental através da expansão do intervalo de portas do *kernel* Linux da máquina hospedeira, a segunda execução processou um volume superior, totalizando cerca de 120 mil requisições com taxa de erro nula (0,0%). A eliminação das falhas confirma que a instabilidade inicial não residia na lógica dos microsserviços, mas na configuração padrão do sistema operacional, validando a robustez da aplicação sob estresse.

O gráfico apresentado na figura 5 corrobora esse resultado otimizado, exibindo uma rampa de crescimento de vazão até a estabilização em um patamar de aproximadamente 490 requisições por segundo, sustentando essa performance até o encerramento do teste.

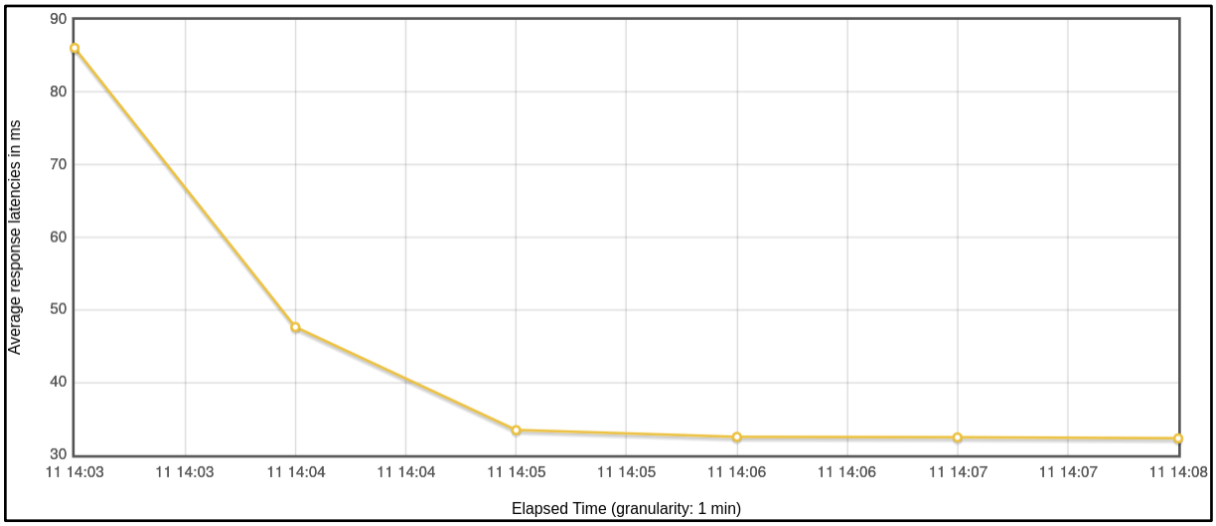
Figura 5 - Total de transações por segundo.



Fonte: Autoria própria (2025).

Por fim, o próximo gráfico demonstra o decaimento inicial e a consistência da latência da solução. Após o atraso médio de aproximadamente 85 ms na inicialização, a linha de latência média se mantém entre 40 ms e 30 ms durante boa parte do tempo, sem apresentar tendências de crescimento que indicariam vazamentos de memória ou degradação progressiva de recursos.

Figura 6 - Latências ao longo do tempo.



Fonte: Autoria própria (2025).

Em suma, os resultados da análise de estabilidade validam a solidez operacional da arquitetura proposta para regimes de execução contínua. A convergência para latências reduzidas após o período de aquecimento, aliada à consistência dos tempos de resposta ao longo do teste, demonstra que o sistema gerencia eficientemente os recursos computacionais, sem

apresentar indícios de degradação ou vazamento de memória. As intercorrências de rede, diagnosticadas como limitações estritas do ambiente de teste, não comprometem a validação lógica dos microsserviços, confirmando que a solução possui a previsibilidade e a resiliência necessárias para operar em cenários produtivos de alta demanda.

3.3 Discussão: O impacto no negócio

A aplicação da heurística do risco mínimo de Bayes, combinada com a baixa latência obtida, valida o sistema como uma alternativa viável para integração em esteiras de pagamento de alto volume.

No contexto B2B, onde o sistema atua como um oráculo de segurança para o *e-commerce* contratante, a estabilidade demonstrada no teste de 16 *threads* sugere que a infraestrutura pode ser dimensionada verticalmente, com mais recursos computacionais e poder de processamento, ou horizontalmente, com mais réplicas, com previsibilidade linear de ganho de desempenho.

Além disso, o mecanismo de *hot reload* do modelo permitiu que novas regras de fraude fossem implantadas sem interrupção do serviço. Isso resolve o problema de *Data Drift* discutido na fundamentação teórica, garantindo que a empresa contratante possa reagir a novos tipos de golpes no mesmo dia em que são detectados, sem necessidade de paradas para manutenção que afetariam o faturamento do *e-commerce*.

A análise de viabilidade econômica demonstra que o sistema é capaz de processar centenas de transações por segundo utilizando uma infraestrutura de médio porte (16 vCPUs), cujo custo mensal estimado em provedores de nuvem (VPS) varia entre USD 50.00 e USD 400.00, um valor irrisório comparado às perdas financeiras evitadas pela prevenção de fraudes.

Complementarmente ao custo de infraestrutura, o modelo de comercialização do *software* adota a estratégia de licenciamento perpétuo com suporte anual. Neste modelo, a empresa contratante realiza um investimento único para aquisição do direito de uso do artefato (*software* e *containers* Docker), garantindo a propriedade da solução em seu ambiente e total conformidade com normas de soberania de dados. Para assegurar a evolução contínua do sistema, estabelece-se um contrato de manutenção anual, tipicamente calculado como um percentual do valor da licença, que garante acesso a atualizações de segurança, novas arquiteturas de modelo e suporte técnico especializado. Essa abordagem oferece previsibilidade orçamentária e maximiza o Retorno sobre o Investimento (ROI), visto que o custo total de

propriedade (TCO) se dilui ao longo do tempo, enquanto a economia gerada pelo bloqueio de fraudes tende a crescer com a escalabilidade do negócio.

4 CONSIDERAÇÕES FINAIS

O presente trabalho atingiu seu objetivo principal ao desenvolver e validar uma arquitetura distribuída capaz de realizar a detecção de fraudes financeiras com a baixa latência exigida por sistemas de pagamentos instantâneos. A investigação confirmou a hipótese de que a combinação de microsserviços, comunicação de alto desempenho e estratégias de *Machine Learning Operations* (*MLOps*) constitui uma solução viável e resiliente para o cenário bancário moderno.

A implementação da comunicação síncrona via gRPC entre os Orquestradores e o *Cluster* de Inferência provou-se fundamental. A utilização de *Protocol Buffers* eliminou a sobrecarga de serialização típica de APIs REST tradicionais, permitindo que a inferência do modelo XGBoost fosse realizada com latência mínima, essencial para não comprometer a experiência do usuário final.

O desacoplamento do Serviço de Auditoria através do Apache Kafka garantiu que o peso das operações de escrita no banco de dados (I/O intensivo) não impactasse o tempo de resposta da autorização.

A integração do *pipeline* de automação endereçou com sucesso o problema do *Data Drift*. A capacidade de rotular transações, retreinar o modelo com dados reais e atualizar as réplicas com *hot reload* sem interrupção do serviço demonstrou que é possível manter a inteligência do sistema atualizada frente a novos padrões de fraude, reduzindo a dependência de intervenções manuais complexas; e os experimentos de carga evidenciaram que o alinhamento entre a concorrência da aplicação e a capacidade física do processador (16 *threads*) resulta no ponto ótimo de eficiência. A arquitetura foi capaz de utilizar a totalidade dos recursos computacionais disponíveis para maximizar a vazão, confirmando que a solução é escalável verticalmente.

Em suma, o artefato desenvolvido não apenas resolve o problema técnico da latência, mas também se estabelece como uma ferramenta de negócio (B2B) estratégica, permitindo que empresas contratantes terceirizem a complexidade da análise de risco com segurança e confiabilidade.

4.1 Trabalhos futuros

Apesar dos resultados promissores, identificam-se oportunidades para a evolução da pesquisa e do artefato. Uma evolução da infraestrutura envolve a migração da orquestração baseada em Docker Compose para um *cluster* Kubernetes (THE KUBERNETES AUTHORS, 2025), habilitando o escalonamento horizontal automático das réplicas de inferência baseado no consumo de CPU em tempo real. Conjuntamente, a camada de segurança deve evoluir da autenticação via chaves de API estáticas para o padrão OAuth2 (RFC 6749, 2012) com gestão de credenciais mTLS (RFC 8705, 2020), visando maior segurança na integração entre serviços em redes hostis.

No âmbito da inteligência artificial, sugere-se a implementação de estratégias de roteamento inteligente no *API Gateway* para testar diferentes versões do modelo simultaneamente em produção. Adicionalmente, para aprimorar a experiência de gerenciamento operacional, propõe-se o desenvolvimento de uma interface gráfica (*frontend*) dedicada à administração do sistema. A implementação de um painel *web* centralizado abstrai a complexidade das chamadas de API para rotulagem e auditoria, proporcionando maior agilidade e usabilidade para as equipes técnicas e de análise de risco.

Em suma, este trabalho estabelece uma fundação robusta para o desenvolvimento de sistemas de detecção de fraude com baixa latência, demonstrando que é possível conciliar a precisão de algoritmos de *Machine Learning* com a eficiência de linguagens de alto desempenho e arquiteturas distribuídas.

REFERÊNCIAS

ABDALLAH, A.; MAAROF, M. A.; ZAINAL, A. **Fraud detection system: A survey**. Journal of Network and Computer Applications, v. 68, p. 90-113, 2016. Disponível em: https://doi.org/10.1016/j.jnca.2016.04.007. Acesso em: 05 dez. 2025.

BAHNSEN, A. C.; STOJANOVIC A.; AOUADA D.; OTTERSTEN B. **Cost Sensitive Credit Card Fraud Detection Using Bayes Minimum Risk**. 12th International Conference on Machine Learning and Applications. Miami, FL, USA, 2013.

CARCILLO, Fabrizio et al. **SCARFF: a scalable framework for streaming credit card fraud detection with Spark.** Information Fusion, v. 41, p. 182-194, 2018. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S1566253517305444. Acesso em: 07 dez. 2025.

CARLSON, Josiah L. **Redis in Action.** 1. ed. Shelter Island: Manning Publications, 2013.

CHEN, T.; GUESTRIN, C. **XGBoost: A Scalable Tree Boosting System.** KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery

and Data Mining. San Francisco: ACM, 2016. p. 785–794. Disponível em: https://dl.acm.org/doi/10.1145/2939672.2939785. Acesso em: 08 dez. 2025.

DOCKER INC. **Docker Compose - Docker Docs.** 2025. Disponível em: https://docs.docker.com/compose/. Acesso em: 10 dez. 2025.

ELKAN, Charles. **The Foundations of Cost-Sensitive Learning.** In: International Joint Conference on Artificial Intelligence (IJCAI). Seattle: [s.n.], 2001. v. 17, p. 973–978. Disponível em: https://www.researchgate.net/publication/2365611_The_Foundations_of_Cost-Sensitive_Learning. Acesso em: 08 dez. 2025.

FEBRABAN. **Pesquisa Febraban de Tecnologia Bancária 2025 - Volume 1.** São Paulo: Federação Brasileira de Bancos, 2025. Disponível em: https://portal.febraban.org.br/pagina/3106/1117/pt-br/pesquisa. Acesso em: 05 dez. 2025.

FIELDING, R. T.; TAYLOR, R. N. **Architectural styles and the design of network-based software architectures.** University of California, Irvine, 2000.

FÓRUM BRASILEIRO DE SEGURANÇA PÚBLICA. **19º Anuário Brasileiro de Segurança Pública**. São Paulo: FBSP, 2025. Disponível em: https://publicacoes.forumseguranca.org.br/handle/123456789/279. Acesso em: 05 dez. 2025.

FRIEDMAN, J. H. **Stochastic gradient boosting.** Computational Statistics & Data Analysis, v. 38, n. 4, p. 367–378, 2002. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S0167947301000652. Acesso em: 19 dez. 2025.

GOMEDE, Everton. **Understanding Data Drift in Machine Learning.** The Modern Scientist, 2023. Disponível em: https://medium.com/the-modern-scientist/understanding-data-drift-in-machine-learning-51bf0022ee11. Acesso em: 06 dez. 2025.

INDRASIRI, Kasun; KURUPPU, Danesh. **gRPC: Up and Running.** 1. ed. Sebastopol: O'Reilly Media, 2020.

JURGOVSKY, Johannes et al. **Sequence classification for credit-card fraud detection.** Expert Systems with Applications, v. 100, p. 234-245, 2018. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S0957417418300435. Acesso em: 07 dez. 2025.

KLEPPMANN, Martin. **Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.** 1. ed. Sebastopol: O'Reilly Media, 2017.

KREPS, J.; NARKHEDE, N.; RAO, J. **Kafka: a distributed messaging system for log processing.** Proceedings of the NetDB, Association for Computing Machinery, v. 11, p. 1–7, 2011. Disponível em: https://www.semanticscholar.org/paper/Kafka-%3A-a-Distributed-Messaging-System-for-Log-Kreps/ea97f112c165e4da1062c30812a41afca4dab628. Acesso em: 05 jan. 2026.

KREUZBERGER D.; KÜHL N.; HIRSCHL S. **Machine Learning Operations (MLOps): Overview, Definition, and Architecture**. IEEE Access (Volume 11), 2023. Disponível em: https://ieeexplore.ieee.org/document/10081336. Acesso em: 07 dez. 2025.

RFC 6749. **The OAuth 2.0 Authorization Framework.** IETF, 2012. Disponível em: https://tools.ietf.org/html/rfc6749. Acesso em: 09 dez. 2025.

RFC 8259. **The JavaScript Object Notation (JSON) Data Interchange Format.** IETF, 2017. Disponível em: https://tools.ietf.org/html/rfc8259. Acesso em: 09 dez. 2025.

RFC 8705. **OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens.** IETF, 2020. Disponível em: https://tools.ietf.org/html/rfc8705. Acesso em: 09 dez. 2025.

THE APACHE SOFTWARE FOUNDATION. **Apache JMeter.** 2024. Disponível em: https://jmeter.apache.org/. Acesso em: 10 dez. 2025.

THE KUBERNETES AUTHORS. **Kubernetes: Production-Grade Container Orchestration.** Disponível em: https://kubernetes.io/. Acesso em: 09 dez. 2025.

LOPEZ-ROJAS, E. A.; ELMIR, A; AXELSSON, S. **PAYSIM: A FINANCIAL MOBILE MONEY SIMULATOR FOR FRAUD DETECTION**. Conferência: 28th European Modeling and Simulation Symposium 2016 (EMSS 2016). Larnaca Cyprus. Disponível em: https://www.researchgate.net/publication/313138956_PAYSIM_A_FINANCIAL_MOBILE_MONEY_SIMULATOR_FOR_FRAUD_DETECTION. Acesso em: 07 dez. 2025.

MERKEL, Dirk. **Docker: lightweight linux containers for consistent development and deployment.** Linux Journal, v. 2014, n. 239, p. 2, 2014.

METABASE. **Metabase Documentation.** 2025. Disponível em: https://www.metabase.com/docs/. Acesso em: 09 dez. 2025.

NARKHEDE, N.; SHAPIRA, G.; PALINO, T. **Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale.** 1. ed. Sebastopol: O'Reilly Media, 2017.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. **FIPS PUB 180-4: Secure Hash Standard (SHS).** Gaithersburg: NIST, 2015. Disponível em: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. Acesso em: 09 dez. 2025.

NEWMAN, Sam. **Building Microservices: Designing Fine-Grained Systems**. 1. ed. Sebastopol: O'Reilly Media, 2015.

NUMFOCUS, INC. **Pandas - Python Data Analysis Library.** 2025. Disponível em: https://pandas.pydata.org/. Acesso em: 10 dez. 2025.

NUMPY TEAM. **NumPy: The fundamental package for scientific computing with Python.** 2025. Disponível em: https://numpy.org/. Acesso em: 10 dez. 2025.

OAKS, Scott. **Java Performance: The Definitive Guide.** 2. ed. Sebastopol: O'Reilly Media, 2020.

OBE, R. O.; HSU, L. S. **PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database.** 3. ed. Sebastopol: O'Reilly Media, 2017.

PALEYES, A.; URMA, R. G.; LAWRENCE, N. **Challenges in deploying machine learning: a survey of case studies.** ACM Transactions on Knowledge Discovery from Data (TKDD), v. 16, n. 6, p. 1-29, 2022. Disponível em: https://www.researchgate.net/publication/360288117_Challenges_in_Deploying_Machine_Learning_a_Survey_of_Case_Studies. Acesso em: 07 dez. 2025.

SPRING. **Spring Cloud LoadBalancer.** 2025. Disponível em: https://docs.spring.io/spring-cloud-commons/reference/spring-cloud-commons/loadbalancer.html. Acesso em: 09 dez. 2025.

STEVENS, W. Richard; FALL, Kevin R. **TCP/IP Illustrated, Vol. 1: The Protocols.** 2. ed. Boston: Addison-Wesley Professional, 2011.

TANENBAUM, Andrew S.; BOS, Herbert. **Modern Operating Systems.** 4. ed. Upper Saddle River: Pearson, 2016.

TURBAN, Efraim et al. **Electronic Commerce 2018: A Managerial and Social Networks Perspective.** 9. ed. Cham: Springer, 2018.

VAN ROSSUM, G.; DRAKE, F. L. **Python 3 Reference Manual.** Scotts Valley: CreateSpace, 2009.

WALLS, Craig. **Spring Boot in Action.** 1. ed. Shelter Island: Manning Publications, 2016.

WIERINGA, Roel. **Design Science Methodology for Information Systems and Software Engineering.** Berlin, Heidelberg: Springer, 2014.

ZHANG, Y. et al. **Customer Transaction Fraud Detection Using Xgboost Model.** ICCEA. Guangzhou, China: IEEE, 2020. p. 554-558. Disponível em: https://ieeexplore.ieee.org/document/9103880. Acesso em 07 dez. 2025.