RΣCΣT
e-ISSN: 2763-8855

Revista Eletrônica de Ciências Exatas e Tecnológicas
Vol. 7 (2026)

Periódicos disponíveis no Portal de Periódicos UFRB
https://periodicos.ufrb.edu.br/recet/

Artigo Original
DOI: 10.66983/recet.v7i1.5297

CHAT-EDA: Uma proposta de Arquitetura Orientada a Eventos para *chatbots* Inteligentes

DOS SANTOS, Anderson Oliveira*,a,1 e VALLE, Tassio Ferreira†,a,2

[a] *Universidade Federal do Recôncavo da Bahia*

Submitted: 13 fev. 2025 Approved 13 jul. 2026 Published 29 jul. 2026

Resumo

Este trabalho apresenta CHAT-EDA, uma arquitetura de software orientada a eventos voltada para o desenvolvimento de *chatbots* inteligentes e sistemas distribuídos. A proposta tem como objetivo resolver os desafios de acoplamento entre módulos, garantindo a escalabilidade, flexibilidade, consistência e persistência de dados, além de assegurar a manutenção do sistema. A arquitetura é composta por três módulos principais: *frontend*, *backend* e um módulo de Inteligência Artificial (Artificial Intelligence, AI), interligados por um barramento de eventos. Cada módulo opera de forma independente e assíncrona, promovendo a modularidade, o desacoplamento e a integridade dos dados. O modelo foi implementado no projeto *Techtalk*, um *chatbot* desenvolvido para otimizar o registro e a análise de chamados técnicos em uma rede de assistência técnica. O sistema foi avaliado por meio de testes de carga com 1.000 usuários concorrentes e testes de resiliência, validando atributos de qualidade como desempenho, consistência e observabilidade. Os resultados demonstraram que a arquitetura é capaz de processar 1.000 requisições simultâneas com tempo médio de resposta de 83ms, taxa de erro inferior a $0,5\%$ sob carga e latência média de 45ms. O sistema manteve a integridade dos dados e a eficiência no processamento assíncrono, atendendo um volume médio de 10.000 interações diárias. A CHAT-EDA mostrou-se eficaz em atender às demandas de sistemas corporativos complexos, destacando sua robustez para futuras expansões. As decisões arquiteturais tomadas reforçam a viabilidade do modelo para aplicações de larga escala, assegurando integração eficiente com tecnologias de inteligência artificial e atendimento a requisitos de negócios diversos.

Palavras-chave: *Arquitetura Orientada a Eventos. Chatbots Inteligentes. Sistemas Distribuídos. Processamento Assíncrono.*

Abstract

This paper presents CHAT-EDA, an Event-Driven Architecture for developing intelligent chatbots in a distributed systems fashion. This proposal addresses challenges related to module coupling while ensuring scalability, flexibility, data consistency, and persistence, along with maintainability. The architecture comprises three main modules: frontend, backend, and an artificial intelligence module, interconnected by an event bus. Each module operates independently and asynchronously, promoting modularity, decoupling, and data integrity. The model was implemented in Techtalk, a chatbot project developed to optimize the registration and analysis of technical service calls in a network of technical support centers. The system was evaluated through load and resilience tests with 1,000 concurrent users, validating quality attributes such as performance, consistency, and observability. The results demonstrated the architecture's capability to handle high volumes of simultaneous messages while maintaining data integrity and efficiency in asynchronous processing. The system handled $1,000$ concurrent users with an average response time of 83ms, an error rate below 0.5% under load, average latency of 45ms, and an average volume of $10,000$ daily interactions. The CHAT-EDA proved effective in addressing the demands of complex corporate systems, highlighting its robustness for future expansions. These architectural decisions reinforce the feasibility of the model for large-scale applications, ensuring efficient integration with artificial intelligence technologies and meeting diverse business requirements.

Keywords: *Event-Driven Architecture. Intelligent chatbots. Distributed Systems. Asynchronous Processing*

Sumário

* Anderson Oliveira dos Santos
† Tassio Ferreira Valle

Esta classe de documento foi preparada no Overleaf e compilada com XeLaTeX. Nenhum erro foi encontrado durante o processo de compilação.

1. INTRODUÇÃO

Com os avanços em Inteligência Artificial (Artificial Intelligence, AI), campo que engloba técnicas de aprendizado de máquina, processamento de linguagem natural e sistemas cognitivos, aplicações baseadas em sistemas de *chatbots* têm se tornado ferramentas essenciais em diversos setores, especialmente naqueles que requerem interação constante e bidirecional entre humanos e sistemas automatizados (Nicolescu; Tudorache, 2022). Esses *chatbots* inteligentes desempenham um papel fundamental na automatização de tarefas repetitivas, no aprimoramento do atendimento ao cliente e na organização eficiente de informações. No suporte técnico, por exemplo, *chatbots* permitem que profissionais identifiquem e registrem com precisão sintomas, causas e soluções, otimizando o fluxo de trabalho e reduzindo erros. Essa interação dinâmica, que pode incluir texto, áudio e outros formatos de mensagens, também contribui para uma experiência de usuário mais personalizada e eficiente (Alaoui; Aouene; Cavalli-Sforza, 2023).

Apesar de seus benefícios, o desenvolvimento de *chatbots* inteligentes apresenta desafios significativos, especialmente relacionados à arquitetura de software. A flexibilidade, escalabilidade e modularidade do sistema são requisitos críticos para garantir que o processamento de diferentes tipos de mensagens ocorra de forma eficiente e que os módulos, como *backend* e AI, operem de maneira integrada, assíncrona e independente (Alaoui; Aouene; Cavalli-Sforza, 2023). Abordagens arquiteturais tradicionais, como o modelo monolítico, podem inicialmente atender às necessidades do sistema (Laigner; Zhou; Salles, 2021), mas frequentemente resultam em problemas de manutenção, dificuldades de escalabilidade e limitações no suporte a futuras evoluções (Lazzari; Farias, 2023). Por outro lado, a Arquitetura Orientada a Eventos, baseada no paradigma de programação orientada a eventos, tem se mostrado uma alternativa eficaz ao desacoplar componentes e permitir o processamento assíncrono de eventos, promovendo maior flexibilidade e distribuição da carga de trabalho (Schipor; Vatavu; Wu, 2019).

Entretanto, a implementação de uma Arquitetura Orientada a Eventos (Event Driven Architecture, EDA) apresenta desafios próprios (Lazzari; Farias, 2023). O gerenciamento do fluxo de dados entre os módulos, a garantia de consistência e a prevenção de sobrecarga da infraestrutura são problemas recorrentes que exigem soluções robustas (Júnior *et al.*, 2018). Em sistemas de *chatbots*, a troca constante de informações contextuais, sintomas e soluções aumenta ainda mais a complexidade do *design* arquitetural, demandando uma infraestrutura que possa suportar altas cargas em tempo real sem comprometer a experiência do usuário ou o desempenho geral do sistema (Juraska *et al.*, 2021).

Este trabalho tem como objetivo apresentar e validar a CHAT-EDA, uma Arquitetura de software orientada a eventos para o desenvolvimento de *chatbots* inteligentes que endereça os requisitos não funcionais de acoplamento, modularidade, escalabilidade, consistência e persistência de dados. A proposta visa garantir que o módulo de AI opere de forma independente, mas bem integrado com o *backend*, possibilitando o processamento de diferentes tipos de mensagens de forma eficiente, sem sobrecarregar a infraestrutura, mantendo a integridade dos dados e assegurando sua durabilidade. Além disso, busca-se assegurar que a arquitetura seja suficientemente flexível para suportar expansões e evoluções futuras do sistema, atendendo às demandas crescentes de aplicações modernas baseadas em *chatbots*. Para validar a eficácia da proposta, a arquitetura foi aplicada no *chatbot Techtalk*, onde foram conduzidos testes práticos para avaliar seu desempenho, escalabilidade e a capacidade de lidar com múltiplas mensagens simultâneas. Os resultados obtidos evidenciam a robustez e a eficiência da CHAT-EDA.

Este artigo está estruturado da seguinte forma: a Seção 2 apresenta o referencial teórico associado à EDA, discutindo conceitos-chave e desafios enfrentados em sistemas distribuídos. A Seção 3 descreve a proposta de arquitetura CHAT-EDA, detalhando sua organização em módulos desacoplados e o modelo de processamento assíncrono de mensagens. Já a Seção 4 explora o estudo de caso do projeto *Techtalk*, destacando sua aplicação prática, os componentes do sistema e os desafios resolvidos. Por fim, a Seção 5 analisa os resultados obtidos, avaliando o desempenho e a escalabilidade da arquitetura, enquanto a Seção 6 apresenta as considerações finais e possibilidades de expansão futura do modelo arquitetural.

2. REFERENCIAL TEÓRICO

O referencial teórico deste trabalho aborda conceitos fundamentais que sustentam a proposta da arquitetura CHAT-EDA. A exploração inicial dos avanços e desafios no desenvolvimento de sistemas de *chatbots* inteligentes fornece o contexto para entender as demandas de arquiteturas modernas. Em seguida, aspectos essenciais de sistemas distribuídos e as abordagens baseadas em eventos são apresentados, ressaltando sua relevância para soluções escaláveis e desacopladas. Adicionalmente, discutem-se os mecanismos para garantir a persistência e consistência de dados em arquiteturas orientadas a eventos, bem como os métodos de avaliação da arquitetura proposta por meio de testes de desempenho e resiliência.

2.1. Chatbots e sistemas inteligentes

Os *chatbots* são programas que interagem com humanos via linguagem natural, textual ou falada, utilizando técnicas de Processamento de Linguagem Natural (Natural Language Processing, NLP) e Inteligência Artificial (Artificial Intelligence, AI). Eles podem ser baseados em regras fixas ou em aprendizado profundo. Segundo Alaoui, Aouene, and Cavalli-Sforza (2023),

chatbots simulam interações humanas naturais e são amplamente utilizados em atendimento ao cliente e suporte técnico.

Os sistemas inteligentes, que aplicam técnicas de AI como aprendizado de máquina para resolver problemas de forma autônoma, percebem o ambiente e tomam decisões para maximizar a alcance de seus objetivos, como descrito por Russell and Norvig (2020). Dentro desse contexto, a evolução dos *chatbots* acompanhou essa trajetória, começando com sistemas baseados em regras, como PARRY (1972) e A.L.I.C.E. (*Artificial Linguistic Internet Computer Entity* - 1995), e avançando para modelos de IA mais robustos e flexíveis. O advento de redes neurais profundas e o uso de modelos como o GPT (*Generative Pre-trained Transformer*) elevaram a capacidade de entendimento e resposta dos *chatbots*, aproximando suas respostas da linguagem humana (Vaswani *et al.*, 2023).

A evolução dos *chatbots* modernos, impulsionada pela crescente complexidade e pela integração de redes neurais profundas, exige a implementação de arquiteturas de software modulares e escaláveis, capazes de integrar eficientemente o processamento de linguagem, rastreamento de diálogos e geração de respostas (Juraska *et al.*, 2021). Essa necessidade de modularidade e flexibilidade nas soluções de *chatbots* reflete a importância de arquiteturas que possam lidar com o aumento da demanda e com a evolução do sistema, permitindo ajustes e expansões sem comprometer o funcionamento. Nesse contexto, a arquitetura de software para sistemas distribuídos se torna crucial. Esses sistemas são compostos por componentes independentes que interagem por meio de protocolos de rede, formando um sistema coeso, mas flexível, o que é essencial para suportar as complexas necessidades de escalabilidade e resiliência dos *chatbots* (Tanenbaum; Steen, 2007).

2.2. Arquitetura de software para sistemas distribuídos

Sistemas distribuídos consistem em componentes independentes que interagem por meio de protocolos de rede, apresentando-se como um sistema único (Tanenbaum; Steen, 2007). Para garantir sua robustez, é necessário enfrentar desafios como eficiência de comunicação, consistência de dados e tolerância a falhas, influenciados pela arquitetura de software adotada (Tanenbaum; Steen, 2007).

Para um Sistema Distribuído, a Arquitetura de Software refere-se ao arranjo organizacional dos componentes de software, suas interações e as políticas usadas para gerenciar essas interações. De acordo com Pressman and Maxim (2021), a arquitetura de software desempenha um papel crucial na determinação da qualidade de um sistema, abrangendo diversos atributos. O **desempenho**, conforme definem Pressman and Maxim (2021), consiste na capacidade do sistema de realizar suas funções de forma eficiente, com tempos de resposta adequados, baixos índices de latência e outros requisitos não-funcionais. Em sistemas distribuídos, o desempenho é um atributo crítico, pois depende da comunicação entre múltiplos componentes e da eficiência na troca de dados (Lazzari; Farias, 2023). A arquitetura precisa ser projetada para que o sistema mantenha tempos de resposta consistentes, mesmo sob carga elevada.

Complementarmente, a **escalabilidade** refere-se à habilidade de um sistema de se adaptar ao aumento da carga de trabalho ou do número de usuários sem degradação significativa de desempenho. Pressman and Maxim (2021) destacam a importância da escalabilidade para acomodar o crescimento, permitindo que novos recursos sejam adicionados de forma transparente e que o sistema mantenha a performance mesmo em cenários de alta demanda, o que é significativo para sistemas distribuídos. A **modificabilidade**(ou flexibilidade do software), conforme indica Pressman and Maxim (2021), refere-se à facilidade com que um sistema pode ser alterado para atender a novas funcionalidades ou ajustes sem comprometer sua estabilidade. Em arquiteturas distribuídas, a modificabilidade é especialmente importante, pois facilita a manutenção e evolução do sistema, possibilitando a introdução de melhorias e correções de forma isolada, sem impacto negativo nos outros componentes (Newman, 2015). Por fim, a **confiabilidade**, como um fator da qualidade do software segundo Pressman and Maxim (2021), é a capacidade do sistema de executar suas funções corretamente e de forma contínua, mesmo diante de falhas. Em sistemas distribuídos, a arquitetura deve prever redundância e mecanismos de recuperação para assegurar que falhas em componentes individuais não afetem o funcionamento do sistema como um todo, garantindo assim a continuidade dos serviços (Rahmatulloh *et al.*, 2022).

A arquitetura de software para sistemas distribuídos se torna um pilar fundamental no desenvolvimento de *chatbots* modernos, uma vez que esses sistemas requerem alta escalabilidade e robustez para suportar a crescente demanda e complexidade das interações. A capacidade de garantir a comunicação eficiente, a consistência dos dados e a tolerância a falhas, característica dos sistemas distribuídos, é essencial para proporcionar uma experiência estável e resiliente em *chatbots* (Tanenbaum; Steen, 2007). Nesse cenário, a arquitetura orientada a eventos (EDA) surge como uma abordagem eficaz para lidar com a comunicação assíncrona e o processamento de grandes volumes de interações em tempo real, sendo uma solução ideal para a construção de sistemas de *chatbots* escaláveis e responsivos.

A EDA oferece uma forma flexível e desacoplada de interação entre componentes, onde produtores e consumidores de eventos interagem de forma independente, promovendo escalabilidade e eficiência no processamento. No caso dos *chatbots*, a utilização de EDA permite a criação de sistemas mais dinâmicos, capazes de reagir rapidamente às solicitações dos usuários, sem sobrecarregar o sistema central (Rahmatulloh *et al.*, 2022). Dessa forma, é possível garantir que o *chatbot* seja capaz de processar múltiplas requisições simultâneas e responder de forma eficiente, mesmo sob alta carga.

2.3. Arquitetura Orientada a Eventos

Na arquitetura orientada a eventos, componentes produzem e consomem eventos de forma assíncrona, promovendo desacoplamento (Rahmatulloh *et al.*, 2022). Os três elementos principais são: os **produtores de eventos**, que identificam mudanças de estado e disparam eventos; os **eventos**, que representam a mudança de estado e contêm informações de carga (conhecidas como *payload*, que corresponde aos dados efetivos transportados pelo evento); e os **consumidores de eventos**, que recebem e reagem a eventos de forma independente (Lazzari; Farias, 2023).

Além desses componentes centrais, concorrência e paralelismo desempenham papéis essenciais no design de sistemas orientados a eventos, especialmente em ambientes distribuídos. A concorrência refere-se à capacidade do sistema de lidar com múltiplas tarefas ao mesmo tempo (Tanenbaum; Steen, 2007), sendo particularmente útil em filas de mensagens onde diferentes componentes aguardam para processar mensagens, permitindo sua organização e distribuição entre consumidores, evitando gargalos de processamento e problemas de espera ativa. O paralelismo, por sua vez, envolve a execução simultânea de

múltiplas tarefas em processadores ou núcleos distintos (Tanenbaum; Steen, 2007), implementado em arquiteturas orientadas a eventos ao permitir que vários consumidores processem mensagens em diferentes *threads* ou processos simultaneamente, maximizando a utilização de recursos computacionais e otimizando o desempenho global do sistema.

No contexto de *chatbots*, dois conceitos são fundamentais para a interação e comunicação dentro de um sistema orientado a eventos: (i) **Mensagens**: Representam uma comunicação direta entre um produtor e um consumidor específico. Elas carregam dados e instruções para serem interpretados e executados pelo consumidor, como comandos ou solicitações diretas. Em um *chatbot*, por exemplo, uma mensagem enviada pelo usuário pode ser uma consulta que demanda resposta imediata. As mensagens geralmente envolvem dados relevantes ao domínio da aplicação, estruturados em objetos que contêm informações de contexto, remetente e o conteúdo da interação. Esse modelo de mensagens promove respostas rápidas e personalizadas ao usuário, garantindo uma experiência interativa fluida (Lazzari; Farias, 2023). Por outro lado, (ii) **Eventos**: Descrevem uma alteração ou atualização no estado do sistema, como o início ou término de uma sessão de conversa, ou a notificação de uma ação que foi concluída. Diferentemente das mensagens, eventos não exigem uma resposta direta e podem ser consumidos por diversos componentes interessados. Em um *chatbot*, um evento pode ser o encerramento de uma conversa que, embora não necessite de resposta, gera notificações para fins de monitoramento ou análise. Assim, enquanto a "mensagem" é um dado ativo de comunicação com o usuário, o "evento" representa um registro desse dado, que serve como um sinal para outros componentes do sistema (Lazzari; Farias, 2023). Ferramentas como Apache Kafka e RabbitMQ garantem a publicação e distribuição de eventos, facilitando a escalabilidade e o processamento assíncrono (Lazzari; Farias, 2023). Sistemas baseados em EDA podem escalar horizontalmente, replicando componentes conforme a demanda (Kieran; Yan, 2010).

Embora promova agilidade e flexibilidade, a EDA apresenta desafios, como rastreamento de eventos e garantia de consistência. Ferramentas de rastreamento distribuído ajudam a monitorar e depurar esses sistemas, enquanto o armazenamento de eventos garante sua reprocessabilidade (Laigner; Zhou; Salles, 2021).

A EDA se mostra crucial para o desenvolvimento de sistemas escaláveis e ágeis, proporcionando uma estrutura flexível e eficiente para a comunicação assíncrona entre componentes. A utilização dessa arquitetura é particularmente relevante para o trabalho de *chatbots*, onde a capacidade de escalar horizontalmente e lidar com múltiplas interações simultâneas é essencial. O desacoplamento promovido pela EDA também é fundamental para garantir que o sistema de *chatbot* seja mais resiliente e capaz de processar eventos de maneira eficiente, sem sobrecarregar os componentes centrais. Com isso, a transição para a persistência e consistência de dados se torna vital para garantir a integridade das interações e operações do sistema, como veremos a seguir.

2.4. Persistência e consistência dos dados em sistemas distribuídos baseados em eventos

Para garantir consistência em sistemas orientados a eventos, é necessário armazenar estados e eventos. A persistência de eventos pode ocorrer de duas formas principais: a persistência por evento, onde cada evento é armazenado com seu *payload* (dados efetivos) e metadados, permitindo a reprodução completa do histórico de alterações (Stopford, 2018), e a persistência por estado, onde o sistema armazena apenas o estado resultante das operações, reduzindo a complexidade de armazenamento mas limitando a capacidade de auditoria de todas as mudanças (Stopford, 2018).

A consistência eventual é uma característica desses sistemas, permitindo que os componentes alcancem um estado consistente ao final do processamento dos eventos (Stopford, 2018). Mecanismos de idempotência garantem que eventos repetidos não impactem o estado final. Padrões de arquitetura como CQRS e *Event Sourcing* são comumente usados para gerenciar a persistência e a consistência (Stopford, 2018).

A persistência e a consistência dos dados são desafios críticos em sistemas distribuídos baseados em eventos, especialmente quando se trata de garantir que o estado do sistema esteja sempre atualizado e correto. No contexto de *chatbots*, essa abordagem é vital para preservar a integridade das interações do usuário, garantindo que o sistema continue a funcionar de forma consistente, mesmo diante de falhas ou inconsistências temporárias. A persistência de eventos, em particular, fornece um meio de garantir que as interações sejam armazenadas e possam ser reprocessadas, caso necessário. Esse conceito se conecta diretamente aos testes de software e arquitetura, onde a confiabilidade e a consistência dos dados devem ser avaliadas para garantir o sucesso do sistema.

2.5. Testes de software e arquitetura em sistemas distribuídos

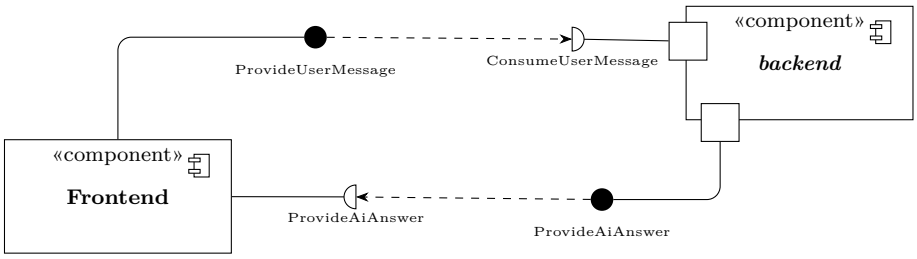
Testes de desempenho, carga e resiliência avaliam a escalabilidade e a estabilidade dos sistemas distribuídos. Ferramentas como Apache JMeter simulam vários usuários simultâneos, analisando tempos de resposta e uso de recursos (Hendayun; Ginanjar; Ihsan, 2023). Para esta pesquisa, foi escolhido o Apache JMeter (v5.6.3) como ferramenta primary de teste de desempenho devido à sua arquitetura robusta baseada em *threads*, que permite simular com precisão múltiplos protocolos simultâneos (HTTP, WebSocket, etc.) em um único cenário de teste. Embora alternativas como *Gatling* e *Locust* ofereçam recursos modernos, o JMeter foi selecionado pela sua capacidade de reproduzir fielmente o comportamento de usuários reais em ambientes corporativos distribuídos, especialmente na simulação de comunicação bidirecional via *WebSocket* necessária para a avaliação completa da arquitetura CHAT-EDA. Dois tipos principais de testes foram conduzidos: **testes de carga**, que avaliam a capacidade do sistema sob carga normal em cenários operacionais típicos (Hendayun; Ginanjar; Ihsan, 2023), e **testes de resiliência**, que testam a resposta do sistema sob carga extrema e falhas de componentes, identificando sua resistência a condições adversas (Hendayun; Ginanjar; Ihsan, 2023).

3. CHAT-EDA: UMA PROPOSTA DE ARQUITETURA ORIENTADA A EVENTOS PARA *CHATBOTS* IMPLANTADOS COMO SISTEMAS INTELIGENTES DISTRIBUÍDOS

Esta Seção descreve a metodologia adotada para o desenvolvimento da arquitetura CHAT-EDA, Arquitetura Orientada a Eventos para *chatbots* implantados como Sistemas Inteligentes Distribuídos.

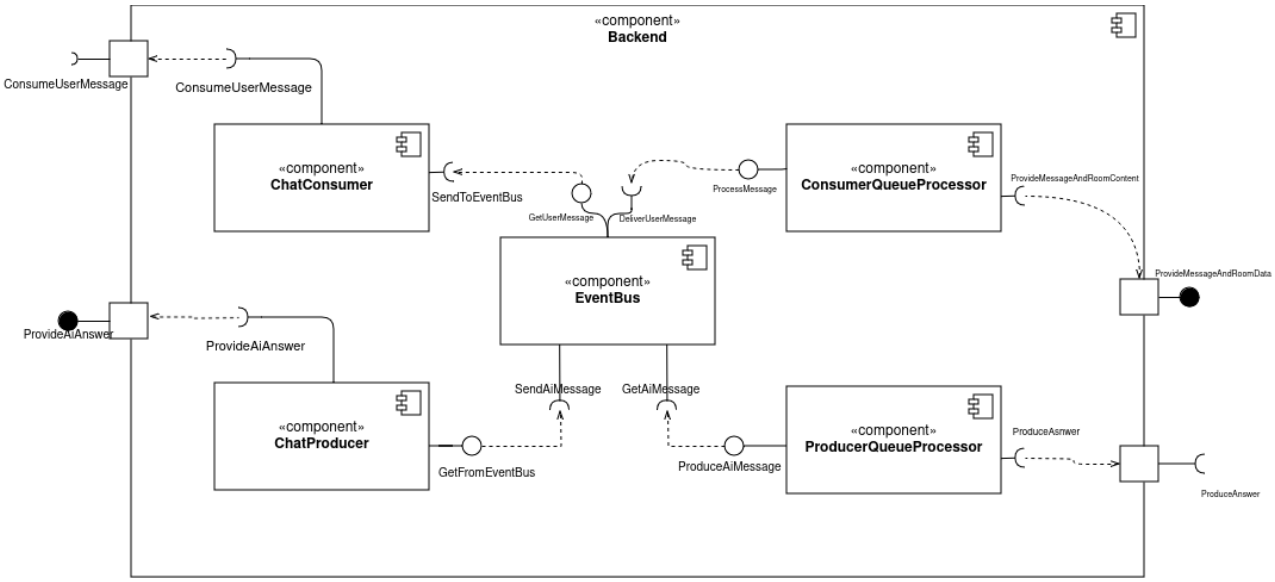
A primeira etapa na construção da arquitetura foi a definição dos requisitos fundamentais para o sistema de *chatbots*, incluindo Escalabilidade, Modularidade, Desacoplamento, Resiliência e Eficiência. Esses requisitos formaram a base para as decisões arquiteturais subsequentes, como a escolha de uma abordagem orientada a eventos e a adoção de uma arquitetura

Figura 1. Integração entre os componentes de *frontend* e *backend*



Fonte: Autores

Figura 2. Organização interna do *backend*, destacando o barramento de eventos *event bus* e o processamento de mensagens.



Fonte: Autores

distribuída. A arquitetura proposta organiza o sistema em três principais módulos: *frontend*, *backend* e Módulo de AI, cada módulo desempenha uma função específica no fluxo de mensagens do *chatbot*, desde a interação inicial com o usuário até a geração e entrega de respostas.

A Figura 1 ilustra o papel do *frontend*, destacando como as mensagens dos usuários são recebidas e encaminhadas para processamento. Essa visão destaca as interações principais com o *backend*, evidenciando o fluxo de mensagens que trafegam entre os módulos.

Na Figura 2, é apresentada a organização interna do *backend*, que atua como o núcleo de comunicação e processamento. O diagrama detalha o uso do barramento de eventos *event bus* para orquestrar a troca de informações entre consumidores e produtores de mensagens, garantindo um fluxo de trabalho desacoplado.

Por fim, a Figura 3 foca no Módulo de IA, evidenciando sua integração com o *backend* para receber as mensagens processadas e retornar respostas inteligentes. Esse módulo demonstra o uso de técnicas avançadas para analisar o conteúdo das mensagens e gerar respostas apropriadas ao contexto.

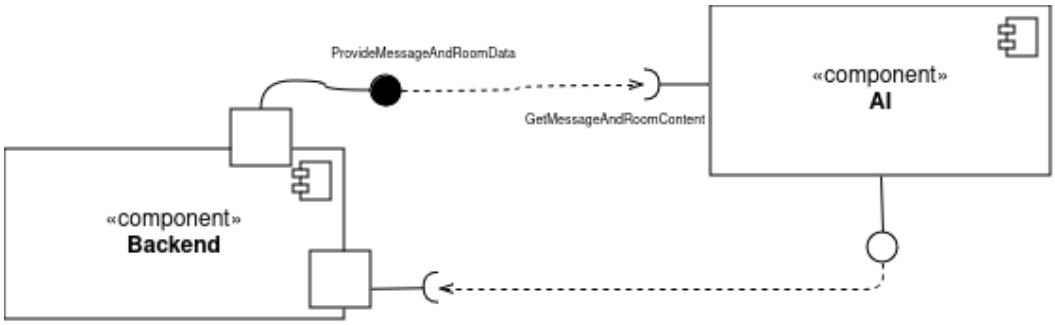
A CHAT-EDA organiza-se em três módulos principais, descritos nas Figuras anteriores. O ***Frontend***, representado na Figura 1, atua como ponto de interação com o usuário, capturando as mensagens enviadas e as encaminhando ao *Backend* através de protocolos HTTP e *WebSocket*, garantindo processamento eficiente. O ***Backend***, destacado na Figura 2, funciona como núcleo de gerenciamento, conectando o *Frontend* ao módulo de AI e intermediando o fluxo de processamento e entrega das interações. Por fim, o **Módulo de Inteligência Artificial** (IA), ilustrado na Figura 3, representa o componente inteligente do sistema, responsável por processar mensagens, interpretar contexto e gerar respostas utilizando técnicas de processamento de linguagem natural e rastreamento de estado de diálogo. Esses três módulos integram-se através do barramento de eventos para funcionar de forma desacoplada, permitindo que cada componente seja desenvolvido, testado e escalado independentemente, mantendo coesão arquitetural.

As subseções a seguir detalham as principais características da arquitetura CHAT-EDA: a independência e o desacoplamento dos componentes, o modelo de processamento assíncrono utilizado e os mecanismos para garantir a consistência e a persistência dos dados. Dentro do *Backend*, destacam-se componentes específicos como o *ChatConsumer* (que recebe mensagens do *Frontend* e organiza em tópicos), o barramento de eventos (que coordena o fluxo assíncrono), o *ConsumerQueueProcessor* (que enriquece mensagens com dados contextuais), o *ProducerQueueProcessor* (que gerencia respostas do módulo de AI) e o *ChatProducer* (que converte e entrega respostas ao *Frontend*).

3.1. Arquitetura desacoplada e independência dos componentes

A Arquitetura Orientada a Eventos busca reduzir o acoplamento entre componentes, permitindo que cada módulo funcione de forma independente. Em sistemas de *chatbot*, a separação entre o módulo de IA e o backend possibilita que ambos sejam

Figura 3. Integração do módulo de AI com o *backend* para análise e geração de respostas.



Fonte: Autores

desenvolvidos e escalados de maneira autônoma, o que facilita o gerenciamento e a manutenção do sistema ao longo do tempo. A estrutura modular da arquitetura é essencial para que os componentes, como IA e *backend*, possam operar de forma isolada e independente.

Na CHAT-EDA, a comunicação entre módulos ocorre por meio de eventos, o que elimina a necessidade de interdependências rígidas entre eles. Esse modelo oferece benefícios significativos para sistemas de *chatbot*, onde diferentes componentes — como processamento de linguagem natural (NLP), análise de contexto e resposta ao usuário — precisam operar de forma autônoma e reagir a eventos gerados pelo usuário de forma dinâmica. Cada módulo ou microsserviço é responsável por uma funcionalidade específica, como o processamento de linguagem, o armazenamento de dados ou a análise preditiva, e pode ser modificado ou escalado conforme necessário, sem interferir no funcionamento dos demais.

Além disso, a CHAT-EDA utiliza comunicação assíncrona, característica fundamental para sistemas que processam grandes volumes de dados em tempo real — definindo-se, neste contexto, tempo real como o processamento com deadlines inferiores a 150 milissegundos para garantir a interatividade esperada pelo usuário. O uso de filas e barramentos de eventos viabiliza o fluxo contínuo de dados entre os componentes, reduzindo a latência média de troca de mensagens para aproximadamente 66 milissegundos (via *WebSocket*) com tempo médio de resposta geral de 83 milissegundos, conforme validado nos testes de carga com 1.000 usuários simultâneos. Essa redução substancial de latência otimiza o desempenho do sistema, minimizando o impacto de possíveis problemas de *backpressure* em cenários de alta concorrência. Com a comunicação baseada em eventos, o sistema evita dependências de chamadas síncronas, o que minimiza gargalos de desempenho e aumenta a resiliência da aplicação. Esse modelo também permite que cada módulo processe as informações conforme sua capacidade, promovendo maior flexibilidade e escalabilidade para lidar com variações de carga e evitar sobrecargas.

3.2. Modelo para processamento assíncrono de mensagens

Na arquitetura CHAT-EDA, o uso de filas de mensagens é essencial para organizar e processar de forma eficiente grandes volumes de dados provenientes de interações. As filas permitem o gerenciamento assíncrono de mensagens e eventos, distribuindo o processamento conforme a disponibilidade dos consumidores e evitando a sobrecarga do sistema. Esse modelo visa garantir que a aplicação possa lidar com diferentes tipos de interações sem comprometer a estabilidade ou o desempenho do sistema, atendendo a necessidade de eficiência no processamento.

Quando um evento de disparo de mensagem é gerado por um produtor (como o *frontend* do *chatbot* ou o módulo de IA), ele sinaliza que uma nova mensagem foi criada e precisa ser processada. Mensagens que requerem uma resposta direta são enfileiradas para consumidores específicos, enquanto eventos que representam mudanças de estado no sistema são disponibilizados para múltiplos consumidores, permitindo uma distribuição mais ampla. Esse fluxo básico é exemplificado na Figura 4.

A fila organiza esses dados seguindo regras específicas, como ordem de chegada (FIFO) ou por nível de prioridade. Dessa forma, eventos que apenas notificam uma alteração ficam acessíveis a diversos consumidores simultaneamente, suportando assinaturas múltiplas.

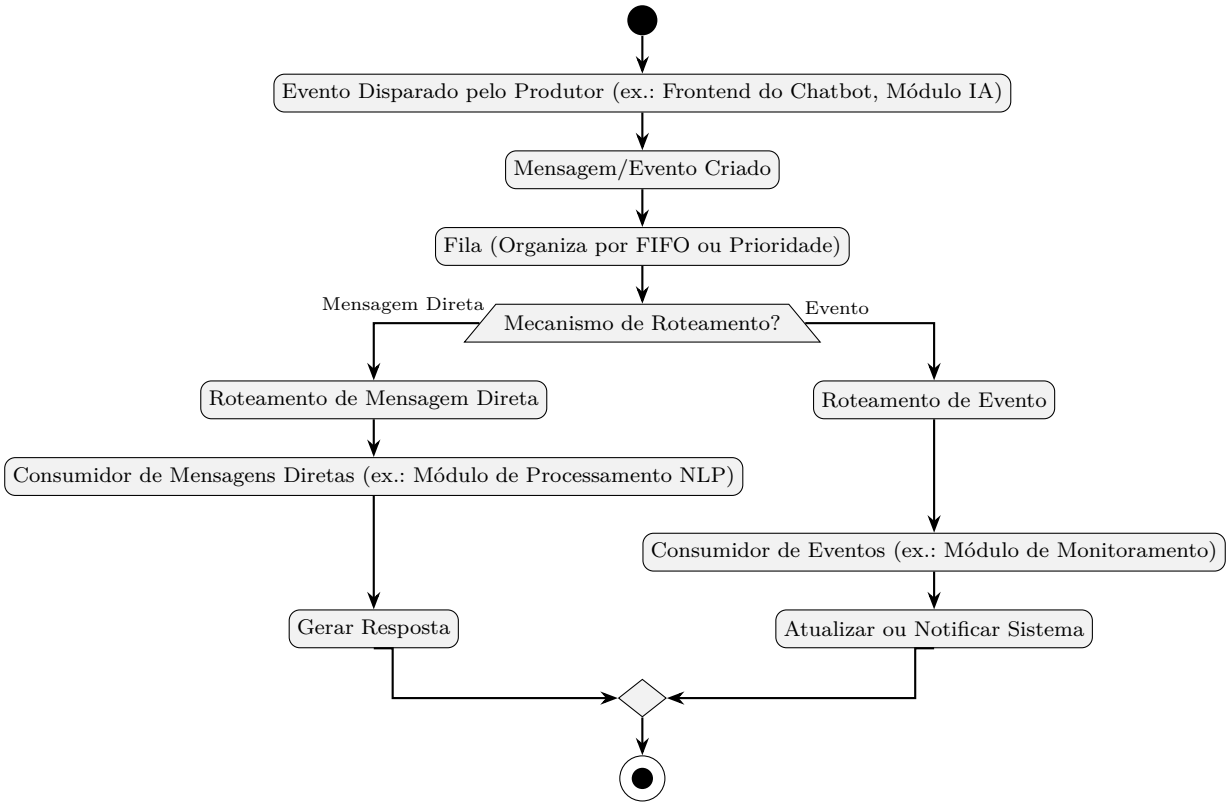
Dependendo do tipo de interação, o sistema aplica mecanismos de roteamento para direcionar as mensagens e eventos ao módulo apropriado. Por exemplo, uma mensagem de texto enviada pelo usuário é roteada ao módulo de processamento de linguagem natural, enquanto um evento de encerramento de conversa é direcionado ao módulo de monitoramento para registro ou análise.

Assim que um consumidor está disponível, ele retira a mensagem da fila e a processa. No caso de mensagens, o processamento resulta em uma resposta ou ação direta. Já para eventos, o processamento pode desencadear atualizações internas ou notificações para outros módulos, sem necessariamente gerar uma resposta imediata.

Tecnologias como Apache Kafka podem ser utilizadas neste contexto, pois atuam como *brokers* (intermediários) de mensagens e barramentos de eventos, permitindo que os dados sejam distribuídos a vários consumidores de forma paralela e eficiente. Kafka e outros *brokers* de eventos otimizam o processamento assíncrono, gerenciando o fluxo e a distribuição das mensagens, o que é crucial para o desempenho em sistemas de grande escala. Ao longo do fluxo de processamento, mecanismos de *logs*, captura de erros e persistência de dados são aplicados. Os *logs* permitem o monitoramento contínuo das atividades, enquanto as capturas de erro identificam e tratam falhas sem comprometer a integridade do sistema. A persistência de dados garante que as mensagens e eventos críticos sejam armazenados de forma durável, possibilitando sua recuperação em caso de falhas. Um exemplo de como o esquema de processamento pode ser adaptado do exemplo básico é mostrado na Figura 5:

Dessa forma, o modelo de processamento assíncrono proposto atende ao objetivo de garantir que o sistema seja capaz

Figura 4. Exemplo básico do fluxo de processamento de mensagens e eventos.



Fonte: Autores

de lidar com diferentes tipos de interações sem sobrecarregar o processamento. A arquitetura baseada em filas e eventos distribui o fluxo de mensagens conforme a disponibilidade dos consumidores, otimiza o uso de recursos e evita gargalos que poderiam comprometer a experiência do usuário e a performance da aplicação. Além disso, ao utilizar *brokers* de eventos, o sistema consegue gerenciar a concorrência e o paralelismo, permitindo o processamento eficiente de mensagens em tempo real. Para que essa comunicação assíncrona mantenha a integridade dos dados em um ambiente distribuído, é necessário implementar mecanismos de persistência e consistência adequados, que serão abordados na próxima seção. Esses mecanismos asseguram que, mesmo com a independência dos componentes e o fluxo assíncrono, os dados permaneçam seguros e acessíveis para todos os módulos do sistema.

3.3. Consistência e persistência de dados

No modelo proposto de Arquitetura Orientada a Eventos (EDA), a persistência de eventos é fundamental para manter a integridade do sistema distribuído, garantindo que os eventos e os estados dos serviços sejam armazenados de forma durável. Já a consistência eventual é um aspecto chave, permitindo que os serviços operem de forma independente e assíncrona, processando eventos no seu próprio ritmo, enquanto o sistema garante que, ao longo do tempo, os dados se tornem consistentes entre os diferentes serviços.

O *event bus* é um componente central na arquitetura proposta. Ele atua como um intermediário que organiza e distribui os eventos entre os serviços. Sempre que um evento é gerado por um componente, ele é publicado no *event bus* e disponibilizado para todos os consumidores interessados, permitindo assim, permitindo que outros componentes interajam de maneira assíncrona e independente, inclusive com bancos de dados independentes, se necessário.

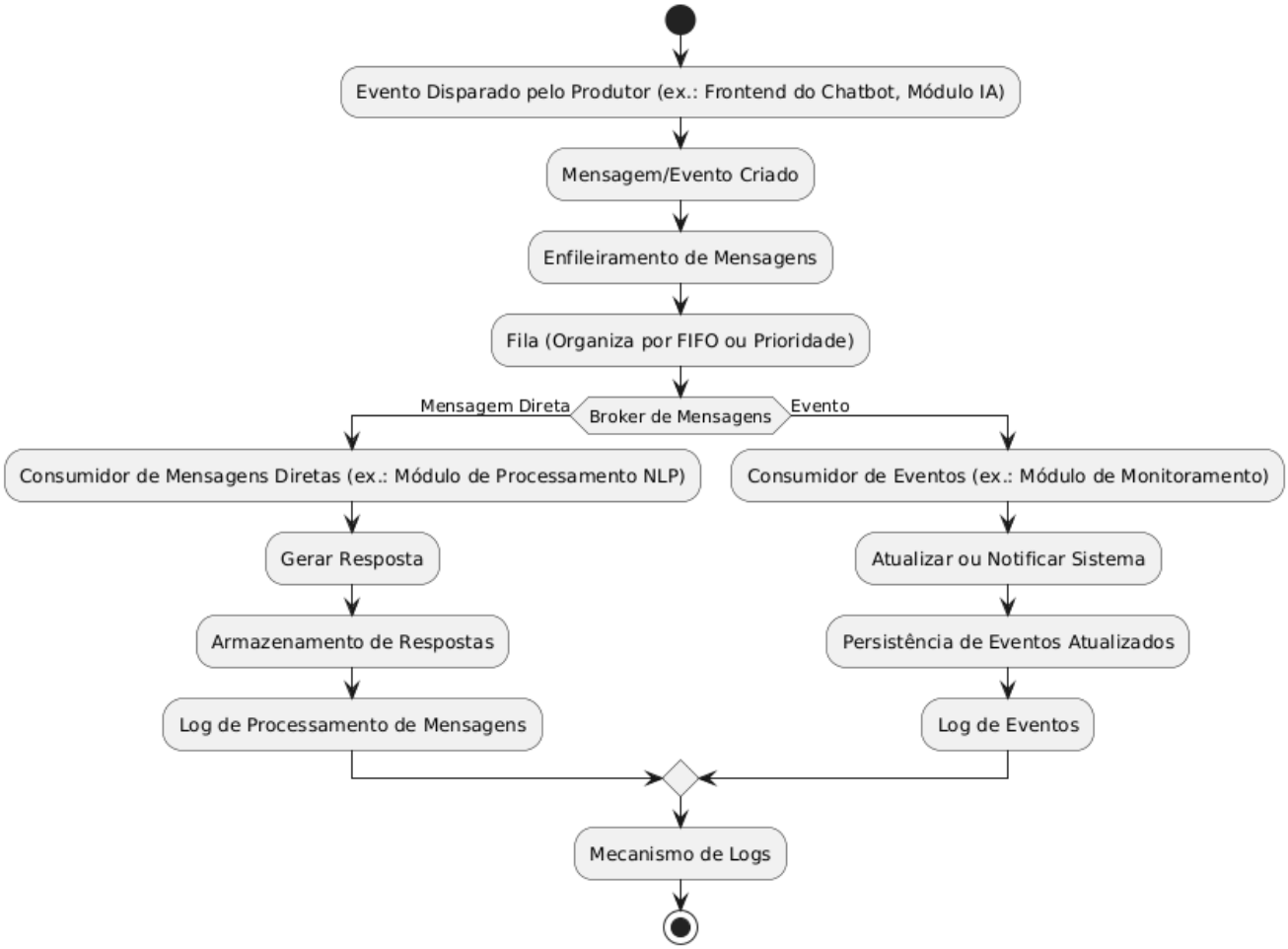
Os eventos são categorizados em tópicos, facilitando o gerenciamento de dados e o direcionamento dos eventos para os serviços apropriados. Por exemplo, eventos relacionados a "mensagens de usuário" podem ser publicados em um tópico específico, enquanto eventos relacionados a "monitoramento do sistema" são categorizados em outro, este fluxo é exemplificado na Figura 6.

O *event bus* não só organiza o fluxo de eventos, mas também gerencia a persistência inicial por meio de *logs* duráveis, garantindo que os eventos sejam armazenados de forma confiável e possam ser reprocessados em caso de falhas.

Para assegurar a integridade e durabilidade dos dados, o modelo proposto utiliza dois tipos principais de persistência. A **persistência de evento** envolve o armazenamento de cada evento gerado em um *log* de eventos, garantindo que as mudanças de estado possam ser reproduzidas e auditadas a qualquer momento. Esse *log* permite que o sistema mantenha um histórico completo de todas as operações, proporcionando a capacidade de reconstruir o estado de qualquer serviço com base na sequência de eventos processados. Complementarmente, a **persistência de estado** funciona em paralelo, onde o sistema mantém uma persistência do estado final de cada entidade (por exemplo, o estado da mensagem enviada pelo usuário), criando uma abordagem dual que maximiza tanto a auditoria quanto a performance operacional.

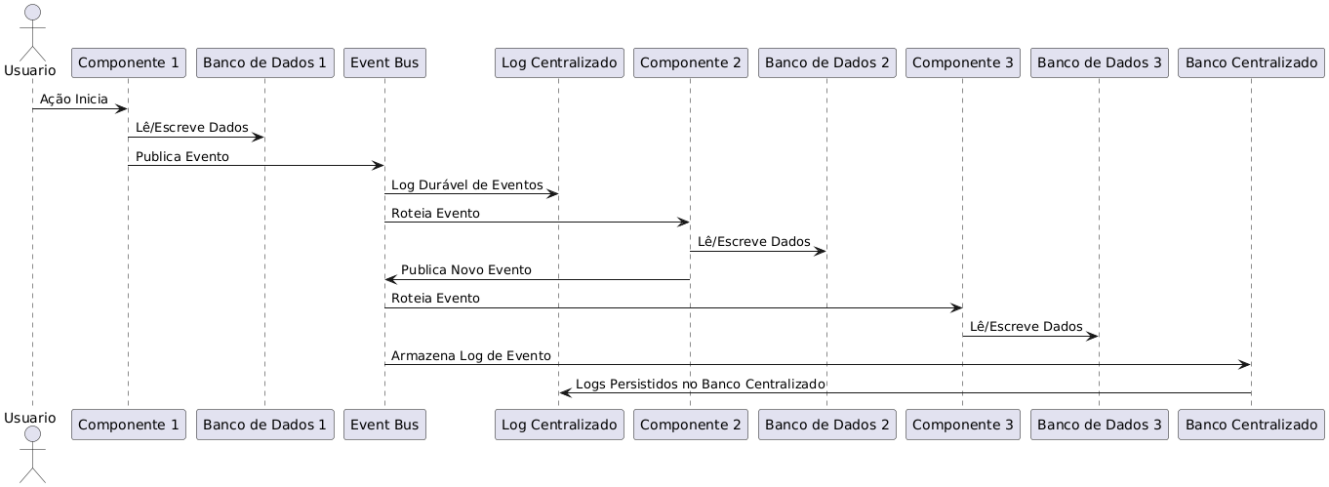
A consistência eventual é uma propriedade fundamental da CHAT-EDA, permitindo que os componentes alcancem uma visão coerente do sistema ao longo do tempo, sem exigir sincronização imediata. Como estes componentes podem processar eventos em momentos diferentes, é possível que haja uma visão temporariamente inconsistente do sistema. No entanto, ao final

Figura 5. Exemplo de esquema de processamento adaptado com *brokers* de mensagens e *logs*.



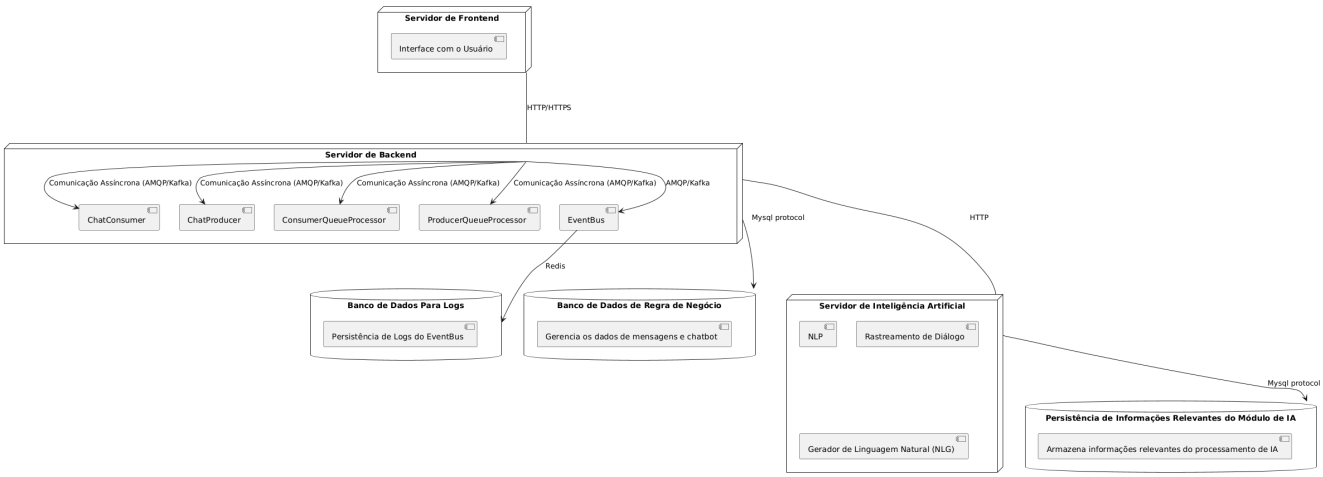
Fonte: Autores

Figura 6. Exemplo de fluxo dos eventos e de persistência no *event bus*.



Fonte: Autores

Figura 7. Representação esquemática da implementação da Arquitetura Orientada a Eventos.



Fonte: Autores

do processamento de todos os eventos, todos os serviços convergem para o mesmo estado. Para garantir que o processamento de eventos seja confiável e não resulte em duplicações, o modelo adota o princípio da idempotência, conceito que assegura que, mesmo que um evento seja processado várias vezes, ele não terá efeitos colaterais adicionais. Assim, se um evento for reprocessado devido a falhas de comunicação ou problemas no sistema, o componente poderá lidar com ele de maneira segura e consistente.

Para ilustrar uma possível abordagem de implementação da arquitetura CHAT-EDA apresentada nesta Seção, a Figura 7 apresenta uma representação gráfica da Arquitetura Orientada a Eventos proposta, destacando os componentes principais e suas interações em um contexto de implementação prática. Este diagrama ilustra como os conceitos explorados ao longo deste capítulo podem ser traduzidos em um sistema funcional, utilizando tecnologias reconhecidas para cada módulo da arquitetura.

3.4. Validação da arquitetura

A validação do modelo foi realizada por meio de testes de desempenho, escalabilidade e resiliência. O primeiro conjunto de testes foi projetado para avaliar o desempenho e a escalabilidade da arquitetura proposta, com foco em verificar o comportamento do sistema sob carga crescente de trabalho. O teste de carga envolveu a simulação de múltiplos usuários realizando operações simultâneas, como a criação de chamados, envio de mensagens e encerramento de conexões. O cenário de teste contemplou 1.000 usuários concorrentes com um período de *ramp-up* de 90 segundos (isto é, um período de aquecimento onde os usuários foram gradualmente adicionados ao sistema ao longo de 90 segundos), com o objetivo de verificar como o sistema lida com a alta concorrência sem comprometer os tempos de resposta ou a degradação do desempenho geral.

A infraestrutura de teste foi configurada para replicar cenários reais de produção, utilizando máquinas virtuais. O Apache JMeter foi empregado para gerar a carga simulada, monitorar o desempenho e registrar métricas essenciais. Os parâmetros de configuração foram ajustados para refletir cenários de alta demanda, com o intuito de testar o sistema sob condições extremas de uso.

A consistência dos dados em sistemas orientados a eventos é um aspecto crítico, especialmente ao se trabalhar com processamento assíncrono e filas de mensagens. Para validar a integridade dos dados e a eficácia da persistência no sistema, foram realizados testes que simularam falhas e verificaram a capacidade do sistema de manter a consistência e a recuperação de dados. Os testes de consistência de dados incluíram dois cenários principais: primeiramente, foi realizado o teste de **parada e reinício do componente de fila**, onde o componente responsável pela fila foi interrompido enquanto processava mensagens ativas, observando-se o comportamento do sistema após a reinicialização; posteriormente, foi executado o teste de **falha de componente consumidor**, onde um componente consumidor foi forçado a falhar durante o processamento de eventos, e o sistema foi avaliado quanto à sua capacidade de reprocessar os eventos pendentes sem duplicação de dados, garantindo a idempotência dos eventos.

A observabilidade é essencial para garantir que o sistema seja monitorado em tempo real e para permitir a identificação e resolução rápida de problemas. Durante os testes, a infraestrutura de monitoramento foi configurada para capturar *logs* detalhados e métricas de desempenho.

Esses testes foram projetados para verificar a resiliência do sistema a falhas e garantir que a persistência de eventos, aliada aos *logs* duráveis, assegure a continuidade do processamento sem perda de dados. Além disso, os testes de idempotência foram realizados com o intuito de verificar se os eventos reprocessados não causaram inconsistências ou duplicações, validando a robustez do sistema para lidar com falhas de comunicação.

Na Seção seguinte, será apresentado um estudo de caso prático que detalha a aplicação deste modelo arquitetural em um cenário real, permitindo avaliar sua eficácia e adequação para diferentes contextos de desenvolvimento.

4. ESTUDO DE CASO: PROJETO TECHTALK

O *Techtalk* foi desenvolvido em uma parceria entre o Núcleo de Engenharia de *software* e Sistemas (NESS) da Universidade Federal do Recôncavo da Bahia (UFRB) e uma grande empresa brasileira do setor de tecnologia, com o objetivo de atender às demandas de sua extensa rede de assistências técnicas distribuídas por todo o território nacional.

Com milhares de chamados técnicos sendo registrados mensalmente, tornou-se essencial implementar um sistema que otimizasse o fluxo de informações e reduzisse os erros no registro e análise de dados.

O *Techtalk* visa não apenas otimizar o fluxo de informações e reduzir erros no registro e análise de dados, mas também oferecer análises estratégicas para a empresa. A partir da identificação de falhas recorrentes, é possível propor melhorias em processos ou no design dos produtos, contribuindo para sua evolução e a redução de defeitos. Além disso, os dados permitem a otimização de estoques, antecipando a demanda por peças específicas com base nos históricos de reparos, o que resulta em um gerenciamento mais eficiente. O sistema de *chatbot* também facilita a detecção de dificuldades recorrentes enfrentadas pelos técnicos, possibilitando a oferta de treinamentos direcionados, alinhados às necessidades reais do time. Outro benefício importante é o aprimoramento do atendimento, pois, ao reduzir o tempo necessário para a resolução de problemas, melhora-se a experiência do cliente, aumentando sua satisfação e fidelidade.

O *Techtalk* foi escolhido como estudo de caso para avaliar a arquitetura CHAT-EDA devido à sua complexidade técnica, requisitos exigentes e impacto direto no fluxo de operações em larga escala. O sistema combina várias tecnologias modernas, como *frameworks* de *frontend* e *backend*, integração com APIs de inteligência artificial e mecanismos de comunicação em tempo real (*websockets*), além de lidar com requisitos de escalabilidade, alta disponibilidade e observabilidade. O projeto enfrenta desafios reais encontrados em sistemas corporativos, incluindo o gerenciamento de comunicação em alta concorrência e em múltiplos formatos, processamento em tempo real com persistência de dados estruturados, observabilidade para depuração e melhoria contínua, e integração com inteligência artificial para fornecer respostas contextuais e personalizadas. Esses desafios tornam o *Techtalk* uma oportunidade valiosa para validar as estratégias de arquitetura discutidas neste trabalho, especialmente no que diz respeito à manutenção dos atributos de qualidade, como desempenho, escalabilidade e confiabilidade.

4.1. Arquitetura e tecnologias utilizadas

A implementação do *Techtalk* foi estruturada com base na arquitetura CHAT-EDA, utilizando uma abordagem modular para promover a separação de responsabilidades e facilitar a manutenção. Essa estrutura modular assegura que cada parte do sistema funcione de maneira independente, mas bem integrada, atendendo aos requisitos de escalabilidade, flexibilidade e desempenho.

A Figura 8 apresenta uma visão geral da implementação, destacando como os módulos estão conectados e os fluxos de dados entre eles.

4.2. Componentes do sistema

Os componentes desta implementação são descritos a seguir: O ***Frontend***, desenvolvido em *React.js* (v18.3.1), é responsável por capturar as mensagens dos usuários e transmiti-las ao *Backend* utilizando protocolos HTTP e *WebSocket*, exibindo também as respostas geradas pelo sistema de forma interativa e responsiva. O ***Backend***, construído em *Django* (v4.2.5), atua como o núcleo do sistema, intermediando a comunicação entre o Frontend e o módulo de AI. A **Persistência de Dados** é gerenciada por um Banco de Dados Centralizado que utiliza *MySQL* (v8.0) para armazenar o estado das mensagens, histórico de interações e eventos críticos, enquanto Redis implementa o *log* de eventos e suporta a fila assíncrona gerenciada pelo *Django* Q. Finalmente, o Módulo de Inteligência Artificial (IA) integra a API do *ChatGPT* e o *LangGraph* para processar mensagens e gerar respostas contextuais, consumindo mensagens da fila de eventos, processando os dados recebidos e publicando respostas no barramento. Este módulo possui uma camada própria de persistência em *MySQL*, onde armazena detalhes técnicos relacionados aos equipamentos e interações.

4.3. Resolução de desafios

O desenvolvimento do *Techtalk* abordou desafios típicos de sistemas distribuídos corporativos, integrando princípios da Arquitetura Orientada a Eventos (EDA) para atingir os objetivos propostos. O gerenciamento de comunicação em alta concorrência e em múltiplos formatos (como texto e áudio) representou um dos desafios mais críticos. A utilização de filas assíncronas, implementadas no barramento de eventos, permitiu organizar e distribuir mensagens para os consumidores de forma eficiente, evitando sobrecarga no sistema. A decisão de desacoplar os módulos por meio de eventos também garantiu que o *frontend*, o *backend* e o módulo de inteligência artificial operassem de forma independente. Cada componente pode processar mensagens e eventos no seu próprio ritmo, promovendo escalabilidade horizontal, com o *ConsumidorChat* (correspondente ao *ChatConsumer*) inserindo mensagens de usuários na fila de eventos para organizar o tráfego de entrada, e o *ProcessadorFilaProdutora* (correspondente ao *ProducerQueueProcessor*) gerenciando respostas do módulo de inteligência artificial.

Para atender ao objetivo de garantir a consistência e a persistência dos dados em uma arquitetura distribuída, foram implementados dois mecanismos complementares de persistência. A persistência de eventos, com *logs* mantidos pelo barramento de eventos, possibilitou rastreamento e auditoria com histórico completo para reconstrução do estado de qualquer serviço em caso de falhas. Em paralelo, a persistência de estado utilizou o banco de dados *MySQL* centralizado no *backend* para armazenar informações contextuais e status de mensagens. A combinação dessas abordagens assegurou que o sistema mantivesse sua integridade mesmo em cenários de falhas ou atrasos, alinhando-se ao modelo de consistência eventual típico de arquiteturas orientadas a eventos.

A observabilidade foi priorizada para mitigar a complexidade do monitoramento em uma arquitetura distribuída. *Logs* detalhados, armazenados no *Redis* e no sistema de *logs* centralizado, forneceram visibilidade sobre o fluxo de eventos e mensagens em tempo real. Ferramentas como o *Silk* ajudaram na análise de desempenho, enquanto os logs de erro garantiram que falhas fossem identificadas e corrigidas rapidamente. O módulo de inteligência artificial, baseado na API do *ChatGPT* e no *LangGraph*, foi integrado ao *backend* por meio do barramento de eventos, assegurando a independência do módulo e permitindo que ele processasse mensagens de forma assíncrona e escalável.

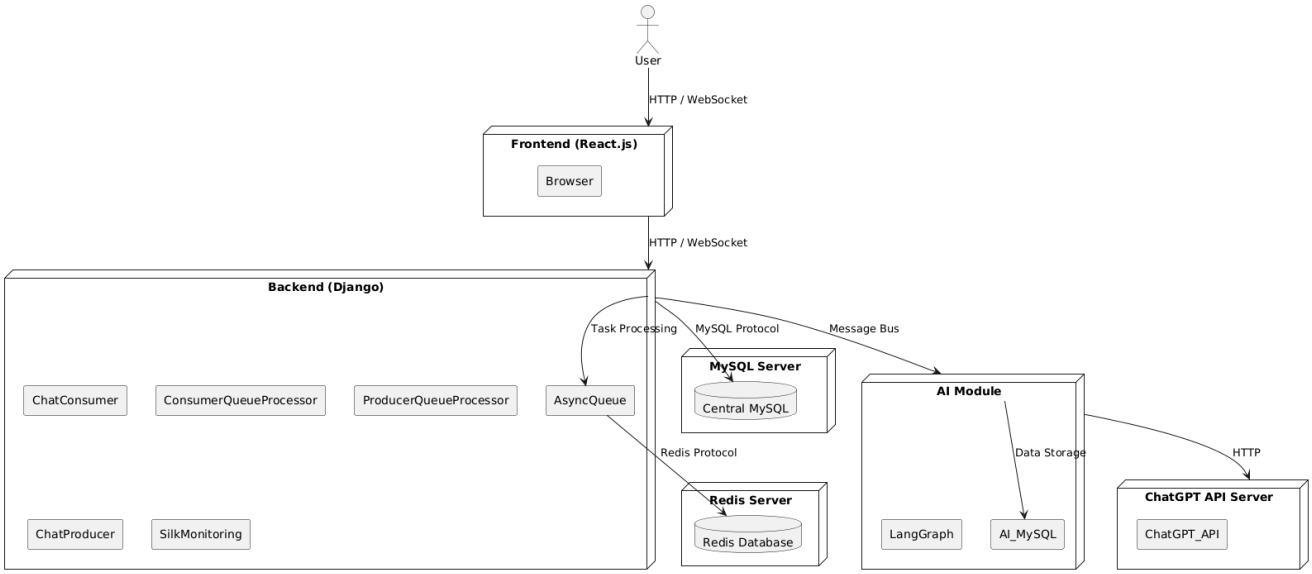
5. RESULTADOS E AVALIAÇÃO DA ARQUITETURA

5.1. Desempenho e escalabilidade

Os testes de desempenho realizados demonstraram que o sistema foi capaz de lidar eficientemente com um grande número de usuários simultâneos. Mesmo com 1.000 usuários concorrentes e em condições de alta carga, o sistema manteve tempos de resposta dentro dos limites aceitáveis. A Tabela 1 resume os principais resultados dos testes de carga, incluindo o tempo médio e máximo de resposta, a taxa de erros e o *throughput*.

A análise dos dados mostrou que a arquitetura foi eficiente em processar chamadas simultâneas, com a troca de mensagens via *websocket* mantendo uma latência baixa e o sistema utilizando a fila assíncrona para garantir a continuidade do proces-

Figura 8. Diagrama de implementação do *Techtalk*.



Fonte: Autores

Table 1: Resultados dos testes de carga

Operação	Erro (%)	Tempo Médio (ms)	Desvio Padrão (σ)	Tempo Máximo (ms)	*throughput* (req/s)
Criar Sala	0,0	205,5	142,3	1.237,0	11,07
Obter Ticket	0,1	169,1	128,6	1.223,0	11,05
Abertura de Conexão *websocket*	0,1	59,5	45,8	568,0	11,08
Troca de Mensagens via *websocket*	0,6	66,2	58,1	771,0	5,76
Encerramento de Conexão *websocket*	0,6	2,3	4,2	83,0	5,77

Fonte: Autores

samento sem gargalos no *backend*. O uso do barramento de eventos se provou eficaz, não comprometendo o desempenho, mesmo com alta concorrência.

A arquitetura provou-se altamente escalável, utilizando o conceito de barramento para descentralizar o processamento e isolar o *backend* do afogamento de conexões abertas. Em uma comparação analítica com abordagens monolíticas tradicionais (onde o tráfego de rede e o processamento ocorrem no mesmo processo síncrono), a manutenção de 1.000 conexões *websocket* ativas exigiria uma alocação de *threads* na razão de $1:1$, elevando exponencialmente o consumo de memória RAM do servidor principal e gerando gargalos diretos no banco de dados (Laigner; Zhou; Salles, 2021). Na CHAT-EDA, o barramento de eventos atua como um amortecedor (*buffer*), permitindo que o servidor web apenas mantenha a conexão ativa, enquanto o trabalho pesado é distribuído de forma enfileirada, mantendo o *throughput* estável mesmo durante picos de interação.

Com base nos resultados, o sistema demonstrou alta capacidade de escalabilidade e capacidade de lidar com cargas elevadas, sem comprometer o desempenho.

5.2. Consistência de dados e persistência

Os testes de consistência de dados validaram a resiliência do sistema, garantindo que, mesmo em cenários de falhas, os dados não fossem perdidos. A estratégia de persistência de eventos foi comprovada como eficaz. Durante o teste de falhas de componentes, o sistema manteve a integridade dos dados e assegurou que as mensagens fossem processadas corretamente após a recuperação do sistema, sem duplicações.

A abordagem de idempotência foi validada, evitando que falhas de comunicação gerassem inconsistências. Esses testes confirmaram que o sistema é robusto e confiável, garantindo que a consistência dos dados fosse mantida, mesmo em situações de falhas ou reinicializações.

5.3. Observabilidade e monitoramento

O monitoramento contínuo foi uma peça-chave na avaliação da arquitetura. Utilizando ferramentas como o *Silk*, foi possível acompanhar em tempo real as métricas de desempenho da API e identificar possíveis gargalos. O *log* detalhado capturado durante os testes forneceu informações valiosas sobre o fluxo de mensagens, garantindo visibilidade do sistema e permitindo ajustes rápidos. Contudo, é importante notar que a implementação extensiva de *logging* em sistemas distribuídos apresenta um custo computacional considerável: cada operação de escrita de *log* incorre em uma latência I/O de aproximadamente $2 - 5$ms, e o armazenamento centralizado desses registros pode consumir vários gigabytes de espaço em disco, dependendo do volume de transações. Essa sobrecarga deve ser cuidadosamente balanceada com os benefícios de rastreabilidade e diagnosticabilidade em ambientes corporativos, onde a observabilidade é essencial para garantir conformidade regulatória e facilitar a depuração de falhas distribuídas.

O *log* capturado, apresentado na Figura 9, fornece uma visão detalhada sobre o comportamento do sistema durante o teste de observabilidade. Ele destaca etapas cruciais no fluxo de execução, incluindo a atualização de contextos no *backend*, a persistência de mensagens no banco de dados e o enfileiramento de mensagens

Figura 9. Exemplo de *log* capturado durante o teste de observabilidade.

Fonte: Autores

para processamento.

Inicialmente, o sistema atualiza o contexto das salas de chat no banco de dados, refletindo as interações do usuário de forma consistente no estado do *backend*. Isso é evidente no registro das mudanças de atributos, como o campo `confirmed`, que armazena a confirmação ou não de uma pergunta feita ao usuário. Essas atualizações garantem a integridade dos dados e a continuidade do fluxo de interação.

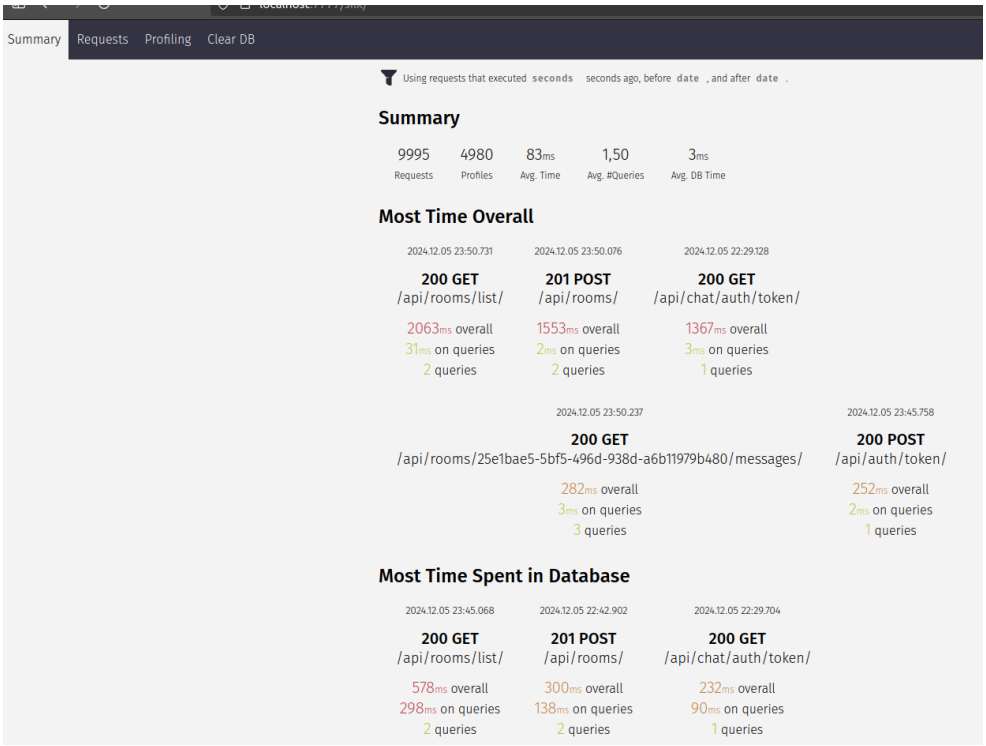
Na sequência, o *log* demonstra a persistência das mensagens no banco de dados, utilizando comandos como `INSERT INTO messages`. Esse processo é essencial para armazenar o histórico de conversas, possibilitando a rastreabilidade das interações entre o *chatbot* e o usuário. Um exemplo notável inclui a mensagem "`Barebone: X695\nConfirma?`", que foi registrada com opções de resposta específicas: ["`Sim`", "`Não`"].

Além disso, as mensagens enfileiradas são direcionadas ao barramento de eventos para processamento pelo módulo de inteligência artificial. O *log* captura esse fluxo em detalhes, registrando desde o enfileiramento até a recepção da mensagem pelo módulo de IA, evidenciando a integração eficiente entre os componentes do sistema. Essa Arquitetura Orientada a Eventos garantiu um processamento ágil das interações e respostas aos usuários.

Por fim, cada evento registrado é acompanhado de informações complementares, como *timestamps*, identificadores únicos de salas e detalhes das configurações utilizadas. Esses dados foram essenciais para diagnosticar possíveis problemas, validar o fluxo correto das mensagens e otimizar a performance geral do sistema.

Além disso, a ferramenta *Silk* desempenhou um papel crucial no monitoramento das métricas de desempenho da API em tempo real. Durante os testes, foram capturados dados importantes, como tempos de resposta de *endpoints*, o número de requisições processadas e a quantidade de consultas ao banco de dados realizadas para atender a cada requisição. Essas informações per-

Figura 10. Captura de tela do painel do Silk para monitoramento da API.



Fonte: Autores

mitiram identificar gargalos de desempenho e otimizar os componentes do sistema.

A Figura 10 apresenta um exemplo do painel de monitoramento do *Silk*, onde é possível observar detalhes sobre os *endpoints* mais utilizados e aqueles que consumiram mais tempo de processamento. Por exemplo, o *endpoint* `/api/rooms/list/`, associado a requisições do tipo `GET`, apresentou um tempo de resposta médio elevado devido a múltiplas consultas ao banco de dados, como indicado pela métrica `Most Time Spent in Database`. Da mesma forma, o *endpoint* `/api/chat/auth/token/`, responsável pela autenticação, demonstrou picos de latência em determinadas cargas.

O *Silk* forneceu uma visão consolidada do número total de requisições (9,995 requisições) e do tempo médio de execução (83 ms por requisição), com detalhes adicionais, como tempos médios de execução no *backend* (3 ms) e tempos de processamento no banco de dados. Essa análise detalhada possibilita a priorização de otimizações nos *endpoints* mais críticos, melhorando a experiência geral de uso do sistema. Com o auxílio dessas métricas, é possível aplicar estratégias para reduzir a latência dos *endpoints* mais utilizados e balancear a carga em situações de alta demanda, garantindo uma API mais eficiente e responsiva.

5.4. Discussão

Os resultados dos testes realizados demonstram a eficácia prática da Arquitetura Orientada a Eventos proposta em um cenário corporativo real. Quanto ao desempenho e escalabilidade, o sistema conseguiu lidar com cargas de 1.000 usuários simultâneos, mantendo tempos de resposta médios de 83ms e taxas de erro inferiores a $0,5\%$, valores que se alinham com os requisitos típicos de sistemas de suporte técnico. No que tange à consistência e persistência de dados, a arquitetura garantiu a integridade dos dados através de uma abordagem dual de persistência (eventos e estado) com princípio de idempotência, comprovado mesmo em cenários de falhas de componentes. Também, a observabilidade foi essencial, com a captura de *logs* detalhados (9.995 requisições capturadas durante os testes) e métricas em tempo real permitindo a identificação rápida de gargalos de desempenho e facilitando a otimização contínua do sistema.

Esses resultados confirmam que a arquitetura orientada a eventos proposta é capaz de atender aos requisitos de modularidade, escalabilidade e flexibilidade, sendo robusta o suficiente para lidar com alta concorrência e integração com inteligência artificial. Outrossim, a arquitetura permite futuras expansões e melhorias contínuas no *Techtalk*. No entanto, é importante notar que enquanto estes testes validam a efetividade arquitetural, uma comparação com uma implementação equivalente baseada em arquitetura monolítica tradicional forneceria perspectivas mais profundas sobre as vantagens comparativas do modelo orientado a eventos.

6. CONSIDERAÇÕES FINAIS

Este trabalho propôs a CHAT-EDA, uma Arquitetura de Software Orientada a Eventos (EDA) para o desenvolvimento de *chatbots* inteligentes, com o objetivo de resolver problemas de acoplamento entre módulos, garantindo a manutenção, escalabilidade e flexibilidade do sistema como um todo. A arquitetura foi desenvolvida para permitir que os módulos de Inteligência Artificial (AI) operem de forma independente, mas de maneira integrada com os componentes do *backend*, possibilitando o processamento de diferentes tipos de mensagens de maneira eficiente, sem sobrecarregar a infraestrutura e mantendo a flexibilidade para futuras expansões do sistema.

A arquitetura proposta foi validada por meio de uma série de testes que avaliaram seu desempenho, escalabilidade, consistência de dados e observabilidade. A modularidade da arquitetura foi comprovada pela independência entre os componentes do sistema, como o *backend* e o módulo de AI, que funcionam de

forma desacoplada, mas bem integrados. Isso garante a flexibilidade necessária para a evolução contínua do sistema e facilita a manutenção.

No que se refere à escalabilidade, os testes de carga com 1.000 usuários simultâneos demonstraram que o sistema é capaz de processar grandes volumes de mensagens de maneira eficiente, sem comprometer a latência ou o desempenho. A utilização de filas assíncronas e do barramento de eventos validou-se como uma estratégia eficaz para distribuir o processamento, evitando gargalos no *backend* e mantendo a latência sob controle.

A consistência e persistência dos dados, aspectos cruciais em arquiteturas distribuídas, foram testadas por meio de simulações de falhas de componentes, e os resultados confirmaram a resiliência do sistema. A integridade dos dados foi preservada graças ao uso de *logs* duráveis e ao princípio da idempotência, que garantiu que o reprocessamento de eventos não resultasse em dados inconsistentes ou duplicados.

A observabilidade foi uma prioridade na arquitetura proposta, permitindo monitorar o sistema em tempo real e identificar possíveis gargalos ou falhas no processamento. A captura de *logs* detalhados e a análise de métricas de desempenho com ferramentas como *Silk* foram essenciais para a identificação e resolução de problemas de latência, melhorando a performance do sistema de forma contínua.

Entretanto, é importante destacar que os resultados dos testes podem ser influenciados pelas especificações técnicas da infraestrutura utilizada, como as máquinas virtuais configuradas para simular o ambiente de produção. A escalabilidade observada pode variar em cenários de maior carga ou com infraestrutura diferente da utilizada nos testes.

Embora os testes realizados tenham validado a eficácia da arquitetura proposta, algumas limitações foram identificadas, como a necessidade de otimização contínua para cenários de extrema carga e a melhoria das ferramentas de monitoramento para rastreamento distribuído. Além disso, em sistemas de grande escala, o desempenho pode ser impactado pela capacidade de processamento e pela configuração dos componentes distribuídos.

Trabalhos futuros podem se concentrar na otimização da escalabilidade, com foco em melhorar o balanceamento de carga e a gestão de filas de mensagens para lidar com volumes ainda maiores de dados, assim como explorar o uso de ferramentas de rastreamento distribuído, como *OpenTelemetry*, para aprimorar a observabilidade em sistemas de larga escala. Também seria valioso investigar a implementação desta arquitetura com o uso de modelos de inteligência artificial mais avançados, como redes neurais profundas ou aprendizado por reforço, para melhorar a personalização e adaptação do *chatbot*, tornando-o mais eficiente e capaz de lidar com interações mais complexas.

A CHAT-EDA se mostrou uma solução eficaz para o desenvolvimento de sistemas distribuídos de *chatbots* inteligentes. Ela garantiu a comunicação eficiente entre os componentes, assegurando a consistência e persistência dos dados, além de proporcionar a escalabilidade e flexibilidade necessárias para lidar com grandes volumes de dados e interações simultâneas. A modularidade e a flexibilidade da arquitetura permitem futuras expansões, tornando-a uma solução robusta para o *Techtalk* e sistemas semelhantes. Com os testes realizados, ficou evidente que a arquitetura é capaz de lidar com os desafios reais encontrados em sistemas corporativos, como alta concorrência, integridade de dados e integração com módulos de AI, fornecendo uma base sólida para futuras melhorias e expansões no sistema.

REFERÊNCIAS

ALAOUI, H. E.; AOUENE, Z. E.; CAVALLI-SFORZA, V. **Building Intelligent Chatbots: Tools, Technologies, and Approaches**. *In:* 2023 3rd International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET). [*S. l.: s. n.*], 2023. p. 1–12. Cit. on p. 37.

HENDAYUN, M.; GINANJAR, A.; IHSAN, Y. **Analysis of Application Performance Testing Using Load Testing and Stress Testing Methods in API Service**. JURNAL SISFOTEK GLOBAL, v. 30, n. 3, 2023. Cit. on p. 39.

JÚNIOR, E. S. *et al.* **Provendo Segurança e Privacidade em Coordenação Distribuída e Extensível**. *In:* ANAIS do XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. Porto Alegre, RS, Brasil: SBC, 2018. p. 267–280. Available from: https://sol.sbc.org.br/index.php/sbrc/article/view/2421. Cit. on p. 37.

JURASKA, J. *et al.* **Athena 2.0: Contextualized Dialogue Management for an Alexa Prize SocialBot**. *In:* ADEL, H.; SHI, S. (eds.). **Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations**. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021. p. 124–133. Available from: https://aclanthology.org/2021.emnlp-demo.15. Cit. on pp. 37, 38.

KIERAN, D.; YAN, W. **A Framework for an Event Driven Video Surveillance System**. *In:* 2010 7th IEEE International Conference on Advanced Video and Signal Based Surveillance. [*S. l.: s. n.*], 2010. p. 97–102. Cit. on p. 39.

LAIGNER, R.; ZHOU, Y.; SALLES, M. A. V. **A Distributed Database System for Event-Based Microservices**. *In:* PROCEEDINGS of the 15th ACM International Conference on Distributed and Event-Based Systems. New York, NY, USA: Association for Computing Machinery, 2021. (DEBS '21), p. 25–30. ISBN 9781450385558. DOI: 10.1145/3465480.3466919. Cit. on pp. 37, 39, 46.

LAZZARI, L.; FARIAS, K. **Uncovering the Hidden Potential of Event-Driven Architecture: A Research Agenda**. [*S. l.: s. n.*], 2023. Cited on pages 3, 5, 6 and 7. eprint: 2308.05270. Available from: https://arxiv.org/abs/2308.05270. Cit. on pp. 37–39.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. [*S. l.*]: O'Reilly Media, 2015. ISBN 9781491950333. Available from: https://books.google.com.br/books?id=jjl4BgAAQBAJ. Cit. on p. 38.

NICOLESCU, L.; TUDORACHE, M. T. **Human-Computer Interaction in Customer Service: The Experience with AI Chatbots—A Systematic Literature Review**. Electronics, v. 11, n. 10, 2022. ISSN 2079-9292. Available from: https://www.mdpi.com/2079-9292/11/10/1579. Cit. on p. 37.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software**. 9th. [*S. l.*]: McGraw Hill Brasil, 2021. ISBN 9786558040118. Available from: https://books.google.com.br/books?id=FSE3EAAAQBAJ. Cit. on p. 38.

RAHMATULLOH, A. *et al.* **Event-Driven Architecture to Improve Performance and Scalability in Microservices-Based Systems**. *In:* 2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS). [*S. l.: s. n.*], 2022. p. 01–06. Cit. on p. 38.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence**. 4th. Upper Saddle River, NJ: Pearson, 2020. Cit. on p. 38.

SCHIPOR, O.-A.; VATAVU, R.-D.; WU, W. **Sapiens: Towards Software Architecture to Support Peripheral Interaction in Smart Environments**. Proc. ACM Hum.-Comput. Interact. Association for Computing Machinery, New York, NY, USA, v. 3, EICS, June 2019. DOI: 10.1145/3331153. Cit. on p. 37.

STOPFORD, B. **Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka**. [*S. l.*]: O'Reilly Media, 2018. ISBN 9781492038245. Cit. on p. 39.

TANENBAUM, A.; STEEN, M. V. **Sistemas Distribuídos: Princípios e Paradigmas**. [*S. l.*]: Pearson Universidades, 2007. ISBN 9788576051428. Available from: https://books.google.com.br/books?id=r2SGPgAACAAJ. Cit. on pp. 38, 39.

VASWANI, A. *et al.* **Attention Is All You Need**. [*S. l.: s. n.*], 2023. Cited on page 4. arXiv: 1706.03762 [cs.CL]. Available from: https://arxiv.org/abs/1706.03762. Cit. on p. 38.