Revista Eletrônica de Iniciação Científica em Computação, 2026, 24:1
ISSN: 3085-8461 • doi: 10.5753/reic.2026.7249

Artigo de Pesquisa/Research Paper

Migração de uma Aplicação Monolítica para Microsserviços: uma Avaliação de Desempenho em Ambientes Distribuídos

Migration of a Monolithic Application to Microservices: a Performance Evaluation in Distributed Environments

David Luiz Becker de Souza [Universidade Tecnológica Federal do Paraná | *davidsouza@alunos.utfpr.edu.br*]
Thiago Henrique Xavier Medeiros [Universidade Tecnológica Federal do Paraná |*thiagomedeiros@alunos.utfpr.edu.br*]
Willian Becker de Souza [Universidade Tecnológica Federal do Paraná |*willian.1998@alunos.utfpr.edu.br*]
Ana Cristina Barreiras Kochem Vendramin [Universidade Tecnológica Federal do Paraná |*criskochem@utfpr.edu.br*]

Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, Av. Sete de Setembro, 3165, Curitiba, PR, 80230-901, Brasil.

Resumo. Este artigo apresenta o processo de reengenharia arquitetural de um sistema de vendas de ingressos, originalmente desenvolvido sob uma arquitetura monolítica, para uma arquitetura de microsserviços. O sistema existente, construído com Next.js, foi reestruturado de forma proativa para garantir escalabilidade e alto desempenho diante de um crescimento futuro no volume de acessos e transações simultâneas. A nova arquitetura distribui funcionalidades como gerenciamento de usuários, catálogo de eventos e processamento de pagamentos em serviços independentes, desenvolvidos em Go e C# .NET. A comunicação entre os serviços é orquestrada por um API *Gateway* com Hive Gateway, enquanto a comunicação assíncrona para fluxos críticos, como o processamento de pagamentos, é realizada via RabbitMQ. O objetivo central é comparar o desempenho das duas arquiteturas, utilizando métricas como tempo de resposta e número de requisições atendidas por segundo, a fim de validar os benefícios da abordagem de microsserviços como estratégia para o crescimento sustentável do sistema. Os resultados indicam que a nova arquitetura distribuída obtém melhor tempo médio de resposta e maior *throughput* em três dos cinco cenários avaliados.

Abstract. This article presents the architectural reengineering of a ticket sales system, originally developed under a monolithic architecture, into a microservices-based architecture. The existing system, built with Next.js, was proactively restructured to ensure scalability and high performance in anticipation of future growth in the volume of simultaneous accesses and transactions. The new architecture distributes functionalities such as user management, event catalog, and payment processing into independent services developed in Go and C# .NET. Communication between services is orchestrated by an API Gateway using Hive Gateway, while asynchronous communication for critical flows, such as payment processing, is performed via RabbitMQ. The main objective is to compare the performance of the two architectures using metrics such as response time and number of requests per second (throughput) to validate the benefits of the microservices approach as a strategy for the sustainable system growth. The results indicate that the new distributed architecture achieves better average response time and higher throughput in three of the five evaluated scenarios.

Palavras-chave: Sistema Monolítico, Arquitetura de Microsserviços, Reengenharia Arquitetural, Avaliação de Desempenho

Keywords: Monolithic System, Microservices Architecture, Architectural Reengineering, Performance Evaluation

Recebido/Received: 18 December 2025 • **Aceito/Accepted:** 20 March 2026 • **Publicado/Published:** 27 March 2026

1 Introdução

O crescimento da digitalização nas últimas décadas foi o principal fator para impulsionar o desenvolvimento de *softwares* e sistemas para resolver diversos problemas da sociedade. Existem *softwares* para praticamente todas as finalidades, tornando-os parte importante da vida das pessoas. Segundo [Pressman and Maxim, 2016], à medida que os *softwares* se tornaram mais críticos para a humanidade, maior se tornou a exigência por qualidade nos produtos de *software*.

Bass *et al*. [2021] destacam que todo sistema de *software* é desenvolvido para satisfazer os objetivos de uma organização, e que a arquitetura atua como a ponte entre esses objetivos e os resultados alcançados pelo sistema. Segundo os autores, a arquitetura de *software* é o conjunto de estruturas necessárias para compreender o sistema, incluindo seus elementos, as relações entre eles e as propriedades associadas. Nesse sentido, a arquitetura é fundamental para que o *software* atinja seus objetivos como produto, sendo um dos fatores cruciais para determinar sua qualidade, pois influencia diretamente atributos como confiabilidade, desempenho e escalabilidade.

Existem muitos tipos de arquiteturas de *software*, e a escolha da mais adequada para um projeto deve considerar cuidadosamente seus requisitos e objetivos. Tradicionalmente, os sistemas de *software* eram desenvolvidos de forma monolítica, ou seja, como uma única aplicação em que todos os componentes (interface, lógica de negócio e acesso a dados) eram fortemente acoplados e executados em um mesmo processo [Sommerville, 2011]. Embora essa abordagem simplifique o desenvolvimento inicial e a implantação, ela tende a dificultar a manutenção, a escalabilidade e a atualização do sistema à medida que sua complexidade aumenta. Sob cargas crescentes, como as observadas em plataformas de vendas

online, arquiteturas monolíticas frequentemente apresentam gargalos de desempenho, pontos únicos de falha e dificuldades operacionais em processos de atualização e implantação [Blinowski *et al*., 2022]. Esses fatores tornam-se críticos em sistemas de venda de ingressos, que enfrentam picos extremos de tráfego e exigem respostas rápidas e consistentes para garantir boa experiência ao usuário e evitar perdas financeiras.

Como alternativa, a arquitetura de microsserviços propõe decompor a aplicação em módulos menores, independentes e implantáveis separadamente. Essas características favorecem especialmente a manutenção e a escalabilidade, pois cada serviço é projetado para desempenhar uma função específica de maneira eficiente, mantendo um foco bem definido [Newman, 2015]. Além disso, quando bem projetada, essa arquitetura pode trazer melhorias significativas em métricas de desempenho, especialmente em ambientes sujeitos a cargas variáveis e tráfego intenso [Barczak and Barczak, 2021]. Contudo, a transição para microsserviços também introduz desafios inerentes a sistemas distribuídos, como o aumento da complexidade operacional, maior dependência de comunicação inter-serviços e a necessidade de mecanismos avançados de observabilidade [Newman, 2015]. Segundo Majors *et al*. [2022], observabilidade em sistemas de *software* é a capacidade de compreender e explicar qualquer estado em que um sistema possa entrar a partir dos dados que ele produz, sem a necessidade de prever previamente as perguntas ou modificar o código para investigá-las.

Este trabalho, então, investiga a migração de um sistema real de venda de ingressos de uma arquitetura monolítica para uma arquitetura baseada em microsserviços, com ênfase na avaliação comparativa de desempenho entre ambas as versões. O estudo envolve a definição das fronteiras dos serviços, reformulação de fluxos críticos, introdução de comunicação assíncrona por meio de *middleware* orientado a mensagens e utilização de um API (*Application Programming Interface*) *Gateway* para centralizar o acesso externo. A partir dessas modificações, foram realizados testes sistemáticos de desempenho, incluindo análises de tempo de resposta, vazão e comportamento sob carga, com o objetivo de caracterizar os impactos reais da transição arquitetural em um sistema.

A escolha de um sistema de vendas de ingressos como objeto de estudo traz uma relevância prática e vital para este trabalho. Diferentemente de um projeto teórico, criado apenas como prova de conceito, a migração aborda problemas reais de um sistema em produção, cujas limitações de escalabilidade impactam diretamente a experiência do usuário e o negócio.

O sistema real de vendas de ingressos é chamado Ticket King. Esse sistema provê a venda de ingressos, o gerenciamento completo dos eventos, o cadastro de lotes e preços e a validação de entradas. Além disso, oferece aos produtores ferramentas de administração e relatórios, enquanto os usuários podem navegar, selecionar eventos e realizar compras de forma simples e segura.

A relevância do tema é reforçada pelo amplo interesse da literatura em compreender as diferentes arquiteturas de *software*. Ao contribuir com dados empíricos provenientes de um sistema de venda de ingressos em operação, este estudo amplia a compreensão dos impactos de microsserviços sobre o desempenho e fornece evidências práticas para equipes e organizações que consideram realizar migrações similares. A análise dos resultados obtidos busca apoiar decisões arquiteturais fundamentadas, destacando quando e sob quais condições a adoção de microsserviços pode trazer ganhos significativos em ambientes de alta demanda.

2 Trabalhos Relacionados

Esta seção baseia-se em uma revisão nas bases IEEE Xplore, ACM Digital Library, Scopus e Google Scholar, considerando publicações a partir de 2014, período de consolidação da arquitetura de microsserviços. Foram incluídos estudos sobre migração de sistemas monolíticos para microsserviços, avaliação de desempenho e arquiteturas aplicadas a sistemas reais, excluindo trabalhos puramente conceituais sem validação experimental ou foco em desempenho e escalabilidade.

A arquitetura em microsserviços e a transição de arquiteturas monolíticas para microsserviços são temas bem documentados na literatura, popularizado por pioneiros como Fowler and Lewis [2014]. Empresas de tecnologia como Netflix [Miguel, 2021] e Uber [Gluck, 2020] publicaram extensos relatos sobre suas jornadas, detalhando os desafios e ganhos obtidos. Na academia, estudos como o de Mohottige *et al*. [2024] analisam empiricamente os *trade-offs* de desempenho entre as duas arquiteturas.

Mohottige *et al*. [2024] realizam uma revisão abrangente das técnicas de identificação de microsserviços, classificando-as em abordagens estáticas, dinâmicas e híbridas. As abordagens estáticas analisam artefatos como código-fonte, esquema de banco de dados e histórico de versões; as dinâmicas observam o comportamento do sistema em execução, considerando fluxos reais de tarefas e interações entre componentes; e as híbridas combinam ambas para fornecer uma visão mais completa dos limites dos serviços.

Fowler and Lewis [2014] discorrem sobre as motivações para o uso da arquitetura em microsserviços para *softwares*. Os autores comentam que a arquitetura monolítica estava causando frustração nos desenvolvedores de sistemas, especialmente porque as aplicações começaram a ser cada vez mais implantadas em nuvem e uma mudança em uma pequena parte da aplicação monolítica exigia o *rebuild* e *deploy* da aplicação inteira. É concluído que a arquitetura em microsserviços é uma abordagem importante e que deve estar em pauta na definição de arquiteturas de aplicações da indústria.

No campo industrial, Miguel [2021] descreve a plataforma Cosmos da Netflix, que combina microsserviços, fluxos assíncronos e funções *serverless* para lidar com o processamento de mídia em larga escala. A solução trouxe melhorias em observabilidade, escalabilidade e velocidade de implantação, embora os autores ressaltem que tal nível de complexidade nem sempre é apropriado para sistemas menores.

De forma semelhante, Gluck [2020] apresenta a Arquitetura de Microsserviços Orientada a Domínio, desenvolvida pela Uber para reduzir a complexidade de um ecossistema que chegou a operar com mais de 2200 microsserviços. A proposta reorganiza serviços em domínios, camadas e portais, oferecendo diretrizes para manter a flexibilidade dos microsserviços e reduzir sua fragmentação excessiva.

Estratégias para migração de sistemas monolíticos para microsserviços têm sido discutidas na literatura. Seedat *et al*. [2024] apresentam um método sistemático, baseado

em *Domain-Driven Design*, para a decomposição de sistemas monolíticos em microsserviços, validado por um estudo de caso conduzido em um contexto financeiro. Faustino *et al*. [2024] analisam o processo de migração de um sistema monolítico para uma arquitetura de microsserviços, usando uma arquitetura modular monolítica como passo intermediário, avaliando tanto o esforço de migração quanto o impacto no desempenho. Stradolini [2020] apresenta um estudo de caso de migração de um sistema monolítico de pagamentos para microsserviços, descrevendo o processo, as decisões envolvidas e uma proposta de migração aplicada e avaliada por especialistas, bem como os desafios e benefícios observados.

Diferentemente dos trabalhos analisados, este estudo concentra-se na migração e avaliação de um sistema real de venda de ingressos, o Ticket King, atualmente em operação em ambiente de produção e que apresenta limitações práticas de escalabilidade. Enquanto estudos anteriores propõem estratégias gerais, exploram abordagens guiadas por *Domain-Driven Design* ou analisam processos de migração em cenários controlados, este trabalho descreve a implementação completa da migração, incluindo a definição das fronteiras dos serviços, a reformulação de fluxos críticos, a introdução de comunicação assíncrona e a adoção de um API *Gateway*. Além disso, embora Faustino *et al*. [2024] e Stradolini [2020] discutam a transição de arquiteturas monolíticas para microsserviços, não apresentam uma comparação sistemática sob carga controlada considerando múltiplos níveis de paralelismo. O presente estudo diferencia-se ao comparar as arquiteturas monolítica e baseada em microsserviços de um mesmo sistema, em cinco cenários distintos e sob diferentes níveis de concorrência, analisando métricas como tempo de resposta, vazão e comportamento sob variação de carga. Esta abordagem busca investigar os impactos da migração arquitetural em um ambiente de alta demanda, oferecendo subsídios para decisões técnicas em contextos semelhantes.

3 Materiais e Métodos

Esta seção apresenta o processo de construção do novo sistema, desde o levantamento de requisitos, modelagem e arquitetura, bem como as ferramentas, tecnologias e o modo como foram aplicadas na implementação da nova arquitetura baseada em microsserviços.

3.1 Levantamento de Requisitos

Inicialmente, foi realizada uma análise detalhada do sistema monolítico existente, identificando suas funcionalidades, fluxos de trabalho e pontos críticos que precisam ser considerados na migração para a arquitetura de microsserviços. A Tabela 1 apresenta os requisitos funcionais e não funcionais.

3.2 Escopo do Sistema

A decomposição do sistema monolítico foi conduzida por meio da análise e definição das fronteiras de cada serviço (*Bounded Context*), resultando na seguinte divisão dos microsserviços que compõem a nova aplicação:

- **Microsserviço Autenticação**: responsável pelo gerenciamento de *login*, autenticação e autorização de usuários, além de gerenciar o cadastro, perfil e dados relacionados a eles, incluindo permissões e papéis;
- **Microsserviço Carrinhos**: gerencia a lógica do carrinho de compras, incluindo adição, remoção e cálculos;
- **Microsserviço Cupons**: gerencia a lógica de cupons e de grupos de cupons, incluindo a busca e validação, bem como a definição de tipos e modalidades de desconto;
- **Microsserviço Eventos**: gerencia os eventos, incluindo informações gerais, colaboradores, e configuração de lotes (tipos de ingresso);
- **Microsserviço Ingressos**: responsável pela emissão e validação de ingressos, além de lidar com *status* de uso e reembolsos;
- **Microsserviço Listas**: gerencia listas de convidados e distribuidores, incluindo controle de acessos e relatórios;
- **Microsserviço Lotes**: responsável pelo gerenciamento dos lotes de ingressos, controlando e garantindo a atualização do estoque;
- **Microsserviço Notificações**: envio de notificações, como atualizações de *status* de pedidos, mensagens promocionais e lembretes para os usuários, via e-mail ou outros canais;
- **Microsserviço Organizações**: gerencia as organizações, incluindo informações administrativas, colaboradores e configurações específicas;
- **Microsserviço Pagamentos**: responsável pelo processamento de pagamentos, incluindo integração com *gateways* de pagamento para pagamento *online*;
- **Microsserviço Reservas**: gerencia reservas de espaços, incluindo a criação, atualização e disponibilidade.

3.3 Modelagem

Esta seção apresenta a modelagem arquitetural do sistema, contemplando tanto a estrutura monolítica original quanto a proposta baseada em microsserviços. A descrição da arquitetura monolítica estabelece a linha de base para a análise comparativa, justificando as decisões de decomposição adotadas na abordagem distribuída.

3.3.1 Arquitetura Monolítica

A arquitetura monolítica, ilustrada na Figura 1, é composta por um *frontend* implementado em *React* (via *Next.js*) e um *backend* centralizado em *Node.js*, ambos integrados a um único banco de dados *MySQL* hospedado na Vercel. O sistema também utiliza um agendador de tarefas baseado em *Nest* + *BullMQ*, apoiado pelo armazenamento chave–valor Redis.

Embora simples, essa arquitetura centraliza a lógica de negócio em um único serviço, elevando o acoplamento, limitando a escalabilidade e dificultando sua evolução.

3.3.2 Arquitetura de Microsserviços

Para desenvolver os microsserviços de forma organizada e que tragam maior manutenibilidade foi utilizado o conceito de arquitetura limpa [Martin, 2019]. O conceito principal desta arquitetura consiste em organizar o código de modo que ele seja independente de *frameworks*, da interface do usuário, do banco de dados, de bibliotecas e APIs externas. Dessa forma, torna-se possível modificar esses componentes sem a necessidade de alterar as regras de negócio e reescrever os microsserviços.

Na arquitetura migrada para microsserviços, ilustrada na Figura 2, cada domínio passa a operar como um serviço independente, com responsabilidades bem definidas e banco de

Tabela 1. Requisitos Funcionais e Não Funcionais

Requisitos Funcionais	
RF01	O sistema deve permitir o cadastro de usuários.
RF02	O sistema deve permitir ao usuário fazer login utilizando suas credenciais.
RF03	O sistema deve permitir ao usuário a recuperação de senha.
RF04	O sistema deve permitir ao usuário pesquisar por eventos.
RF05	O sistema deve exibir uma lista com os eventos futuros.
RF06	O sistema deve apresentar os eventos em formato visual estruturado.
RF07	O sistema deve exibir os eventos com mais acessos nas últimas 24 horas.
RF08	O sistema deve permitir a filtragem de eventos por categoria.
RF09	O sistema deve permitir ao usuário visualizar detalhes dos eventos (hora, local, valor do ingresso).
RF10	O sistema deve permitir ao usuário selecionar o tipo de ingresso (VIP, pista, camarote), quando aplicável.
RF11	O sistema deve permitir ao usuário adicionar ingressos ao carrinho de compras.
RF12	O sistema deve permitir ao usuário remover ingressos do carrinho.
RF13	O sistema deve permitir ao usuário atualizar a quantidade de ingressos no carrinho.
RF14	O sistema deve permitir ao usuário visualizar os itens adicionados ao carrinho.
RF15	O sistema deve calcular automaticamente o valor total da compra, considerando taxas e descontos aplicáveis.
RF16	O sistema deve permitir ao usuário aplicar um cupom de desconto durante o processo de compra.
RF17	O sistema deve validar a elegibilidade e validade do cupom informado.
RF18	O sistema deve criar uma reserva temporária dos ingressos selecionados durante o processo de compra.
RF19	O sistema deve bloquear temporariamente os ingressos reservados até a conclusão do pagamento.
RF20	O sistema deve liberar automaticamente a reserva caso o pagamento não seja confirmado dentro de dez minutos.
RF21	O sistema deve impedir a reserva ou compra de ingressos indisponíveis.
RF22	O sistema deve permitir ao usuário realizar a compra de ingressos.
RF23	O sistema deve permitir o pagamento com PIX e cartão de crédito.
RF24	O sistema deve enviar a confirmação de compra ao usuário.
RF25	O sistema deve permitir que o usuário visualize seus ingressos comprados.
RF26	O sistema deve enviar ao usuário, por meio eletrônico, o ingresso com QR Code.
RF27	O sistema deve permitir a validação do ingresso por meio da leitura do QR Code.
RF28	O sistema deve registrar o status de uso do ingresso após a validação.
RF29	O sistema deve controlar a quantidade disponível de ingressos por lote.
RF30	O sistema deve impedir a venda de ingressos quando o lote estiver esgotado.
RF31	O sistema deve permitir o cadastro e gerenciamento de organizações responsáveis por eventos.
RF32	O sistema deve permitir a associação de colaboradores a organizações com diferentes papéis e permissões.
RF33	O sistema deve permitir o cadastro e gerenciamento de listas de convidados para eventos.
RF34	O sistema deve enviar notificações ao usuário sobre alterações no status do pedido ou da reserva.
Requisitos Não Funcionais	
RNF01	O sistema deve oferecer suporte à interface em língua portuguesa.
RNF02	O sistema deve ser compatível com os navegadores Chrome, Firefox e Edge.
RNF03	O sistema deve possuir design responsivo, garantindo funcionamento adequado em navegadores de dispositivos móveis.
RNF04	O sistema deve utilizar métodos de criptografia para garantir a segurança das transações.
RNF05	O sistema deve garantir disponibilidade de no mínimo 99% nas vendas de ingressos.
RNF06	O sistema deve garantir tolerância a falhas, evitando indisponibilidade total diante da falha de um componente.
RNF07	O sistema deve permitir a implantação independente de seus componentes, sem interromper o funcionamento global da aplicação.
RNF08	O sistema deve permitir a modificação ou evolução de um serviço com impacto mínimo nos demais componentes.
RNF09	O sistema deve registrar *logs* e métricas para monitoramento, rastreamento de falhas e auditoria.

dados próprio. O *frontend* deixa de acessar diretamente o *backend* e passa a se comunicar exclusivamente por meio de um API *Gateway*, responsável por rotear e orquestrar requisições.

A comunicação entre os microsserviços ocorre de duas formas complementares:

- **Síncrona**: realizada por meio de um API *Gateway*, que centraliza e encaminha requisições do *frontend* para os serviços internos, reduzindo a complexidade e o acoplamento;
- **Assíncrona**: realizada por meio de um sistema de mensageria, permitindo integração orientada a eventos, maior escalabilidade e resiliência.

A aplicação *frontend* atua como principal interface para produtores e clientes, servindo como ponto de entrada para todas as interações com o *backend*. No projeto, manteve-se o mesmo *frontend* do sistema monolítico, sendo necessária apenas sua integração com as rotas do novo *backend* distribuído, já que o foco deste trabalho está na transição arquitetural do *backend*. Desenvolvido em Next.js e TypeScript, o *frontend* combina o ecossistema React com recursos de SSR (*Server-Side Rendering*) e SSG (*Static Site Generation*), garantindo desempenho e boa indexação, além de maior segurança e robustez por meio de tipagem estática.

Na arquitetura de microsserviços, o *frontend* se comunica exclusivamente com o API *Gateway*, centralizando todas as requisições ao *backend*. Para lidar com operações assíncronas críticas, como o processamento de pagamentos, utiliza-se

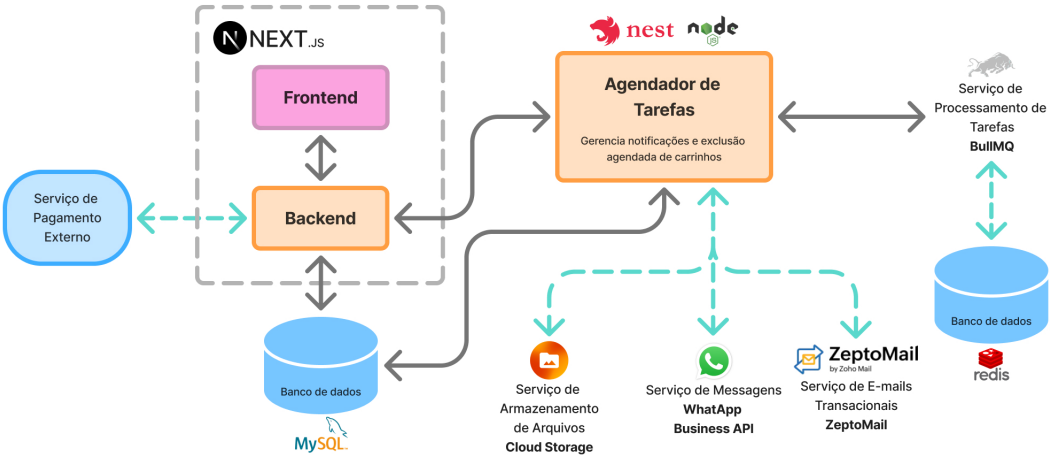

Figura 1. Arquitetura do Sistema Monolítico

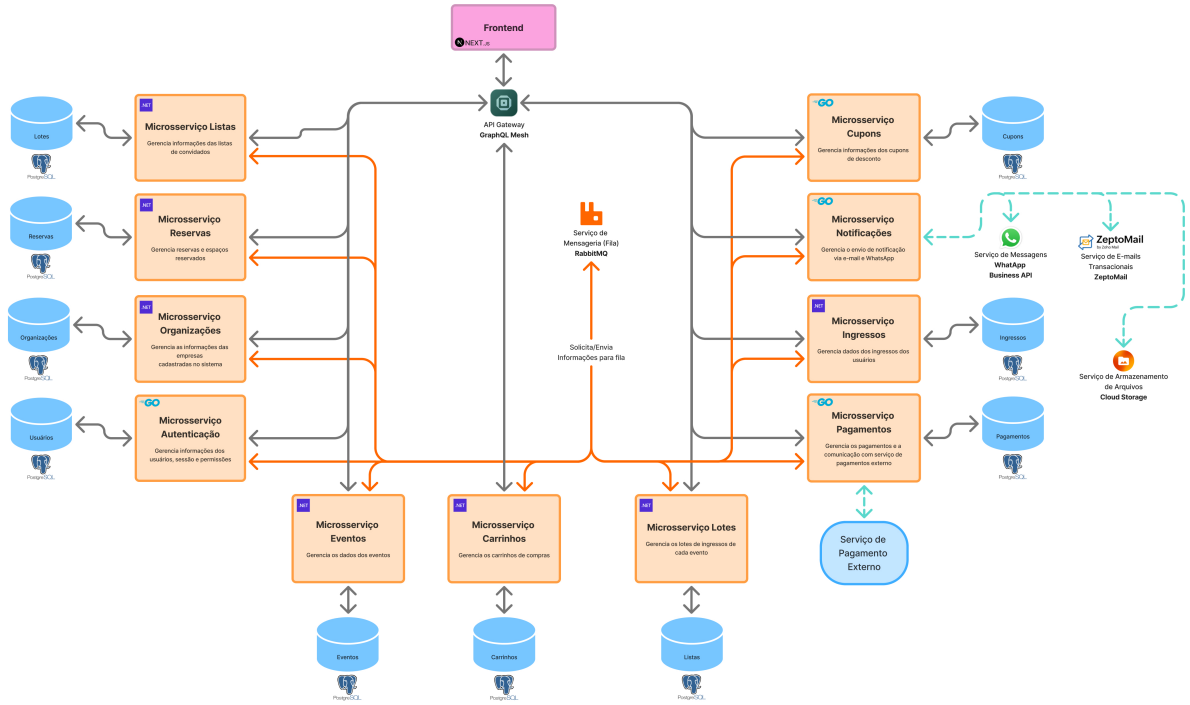

Figura 2. Arquitetura do Sistema em Microsserviços. As setas contínuas em preto representam comunicação síncrona no padrão requisição-resposta mediada pelo API Gateway. As setas contínuas em laranja indicam comunicação assíncrona baseada em mensageria por meio do RabbitMQ. As linhas tracejadas representam integrações com serviços externos.

SSE (*Server-Sent Events*), permitindo ao servidor enviar eventos em tempo real ao cliente, sem a necessidade de *polling*.

O API *Gateway* funciona como porta de entrada única para os microsserviços, simplificando e organizando a comunicação. Neste trabalho adotou-se o *Hive Gateway*, baseado no conceito de *GraphQL Federation*, permitindo unificar múltiplas APIs em uma única interface de consulta [GraphQL, 2025]. Assim, o *frontend* faz uma única requisição, e o *gateway* encaminha internamente as chamadas para os microsserviços relevantes.

Essa camada foi implementada com GraphQL Mesh, que gera um esquema GraphQL unificado a partir dos documentos OpenAPI dos microsserviços. Cada serviço é registrado como um subgrafo, possibilitando ao *gateway* compor e integrar suas APIs em uma estrutura centralizada, reduzindo a complexidade e a sobrecarga de comunicação.

Foram utilizadas as linguagens *Go* e C (*.NET Core*) na implementação dos microsserviços. A linguagem Go foi empregada nos serviços que exigem maior concorrência, como autenticação, pagamentos, cupons e notificações. A lingua-

gem C foi empregada em domínios orientados a dados e lógica estruturada, como nos serviços de eventos, lotes, ingressos, organizações, listas, carrinhos e reservas. Essa escolha heterogênea evidencia uma vantagem central dos microsserviços: cada componente pode ser desenvolvido com a tecnologia mais adequada às suas necessidades.

Nos serviços ligados ao núcleo do domínio de negócio, adotou-se C devido à maturidade do *.NET Core*, ao suporte a orientação a objetos e ao uso de ferramentas como o EF (*Entity Framework*) *Core*, que facilitam o trabalho com modelos fortemente estruturados e dados relacionais. Já para serviços que demandam alta concorrência e eficiência, optou-se por Go, cujo desempenho, baixo consumo de recursos e suporte nativo à concorrência a tornam ideal para operações intensivas.

O API *Gateway* comunica-se com os microsserviços via REST (*Representational State Transfer*). Esse estilo arquitetural, proposto em [Fielding, 2000], define sistemas baseados em recursos identificados por URLs (*Uniform Resource Locator*) únicas. Todos os microsserviços expõem APIs REST documentadas por contratos OpenAPI, que especificam formalmente os *endpoints*, métodos e formatos de dados. A interação é *stateless*, exigindo que cada requisição contenha todas as informações necessárias, o que contribui para simplicidade, interoperabilidade e escalabilidade.

Para a comunicação assíncrona entre os microsserviços, conforme pode ser visto na Figura 3, adotou-se o *RabbitMQ*, um *middleware* de mensagens amplamente utilizado em sistemas distribuídos [RabbitMQ, 2025]. Ele foi empregado principalmente em fluxos críticos, como processamento de pagamentos, atualização de estoque, controle de transações e envio de notificações. Cada serviço publica e consome mensagens em filas específicas, garantindo baixo acoplamento e maior resiliência. As principais filas criadas incluem:

- **cart.created**: criação de carrinhos, consumida pelo serviço de lotes para atualizar estoque;
- **cart.scheduled.deletion**: fila com *delay* de 10 minutos, usada para remover carrinhos não pagos;
- **cart.deleted**: remoção de carrinhos, consumida pelo serviço de lotes para reverter estoque;
- **ticket.cancelled**: cancelamento de ingressos, utilizada pelo serviço de lotes;
- **ticket.created**: criação de ingressos, usada pelo serviço de notificações;
- **payment.approved**: pagamento aprovado, consumida pelo serviço de ingressos para gerar o ingresso;
- **payment.rejected**: pagamento rejeitado.

Esse mecanismo permite que o fluxo de compra seja totalmente assíncrono: o serviço de pagamentos publica o *status* da transação, o serviço de ingressos cria o ingresso e gera o *QR Code*, e o serviço de notificações envia essas informações ao usuário. Quando dados adicionais são necessários, o API *Gateway* atua como orquestrador, reunindo as informações de diferentes microsserviços.

O *PostgreSQL* [PostgreSQL, 2025] foi adotado como banco de dados dos microsserviços, em substituição ao *MySQL* da arquitetura monolítica, devido ao suporte mais avançado a concorrência, tipos de dados e consultas complexas. Cada microsserviço possui seu próprio banco, reforçando a independência dos contextos.

Nos serviços escritos em C# (*.NET*), a integração foi feita com o EF *Core*, que mapeia entidades para tabelas e gerencia todas as operações de persistência, incluindo migrações versionadas. Já nos microsserviços em Go utilizou-se o *SQLC*, responsável por gerar código Go a partir de consultas SQL, além da ferramenta *Goose* para migração e versionamento do esquema. Em ambos os casos, o acesso ao banco é implementado na camada de *Infrastructure*, seguindo o padrão de *Repository*.

3.4 Implementação do Sistema Ticket King na Arquitetura de Microsserviços

O Ticket King é uma plataforma unificada para gestão de eventos e venda de ingressos. O fluxo principal consiste na seleção do evento, escolha de tipos e quantidades de ingressos, conferência do carrinho, pagamento e envio dos ingressos ao usuário. Além disso, o sistema disponibiliza ferramentas para organizadores, incluindo gestão de eventos, listas de convidados, reservas e demais configurações administrativas.

A página inicial (Figura 4) reúne um menu com barra de busca, opções de login e anúncios, além de banners rotativos de eventos em destaque. Abaixo, são exibidas listas de eventos em alta nas últimas 24 horas e de próximos eventos, organizadas para oferecer uma navegação objetiva, evitando excesso de opções não categorizadas.

As páginas de evento (Figura 5 e Figura 6) apresentam o banner, informações completas de local e horário, dados do organizador, categorias do evento e um link para rota de acesso ao local. A área principal contém a listagem de lotes, com tipos de ingressos, preços e seleção de quantidades. Antes da finalização, o usuário pode inserir um cupom de desconto.

A etapa de pagamento (Figura 7) exibe, à direita, o resumo da compra com valores parciais e total. À esquerda, encontra-se o formulário com dados do comprador e do participante que utilizará os ingressos.

O painel de administração (Figura 8) é voltado aos organizadores e apresenta a lista de eventos cadastrados, com informações como nome, *status* (ativo, oculto ou encerrado), data, local, total de ingressos, disponibilidade e ações, além da opção de criação de novos eventos.

Na visão geral do evento (Figura 9), são exibidos dados básicos (nome, *status*, data e local), indicadores de vendas e gráficos de distribuição por público e validação de ingressos. O resumo financeiro inclui vendas totais, valores aguardando pagamento, total a receber, total recebido, ticket médio bruto e líquido, além do consolidado específico para vendas online.

Os preços dos ingressos para os eventos frequentemente não são fixos e evoluem com o tempo, por isso o sistema de venda de ingresso precisa ter uma funcionalidade para suprir esta necessidade, permitindo a criação de lotes para um evento. A criação de lotes permite definir título, quantidade, preço, período de vendas, gênero, forma de disponibilidade (público, por *link* ou manual), limites de compra, descrição e campos solicitados ao usuário, garantindo a disponibilização automática conforme os parâmetros definidos.

Por fim, o sistema permite a gestão de cupons de desconto (Figura 10), utilizados para oferecer condições especiais e para análise de campanhas de *marketing*. O cadastro inclui código identificador, valor do desconto e vinculação a grupos ou colaboradores, permitindo mensurar a origem das vendas.

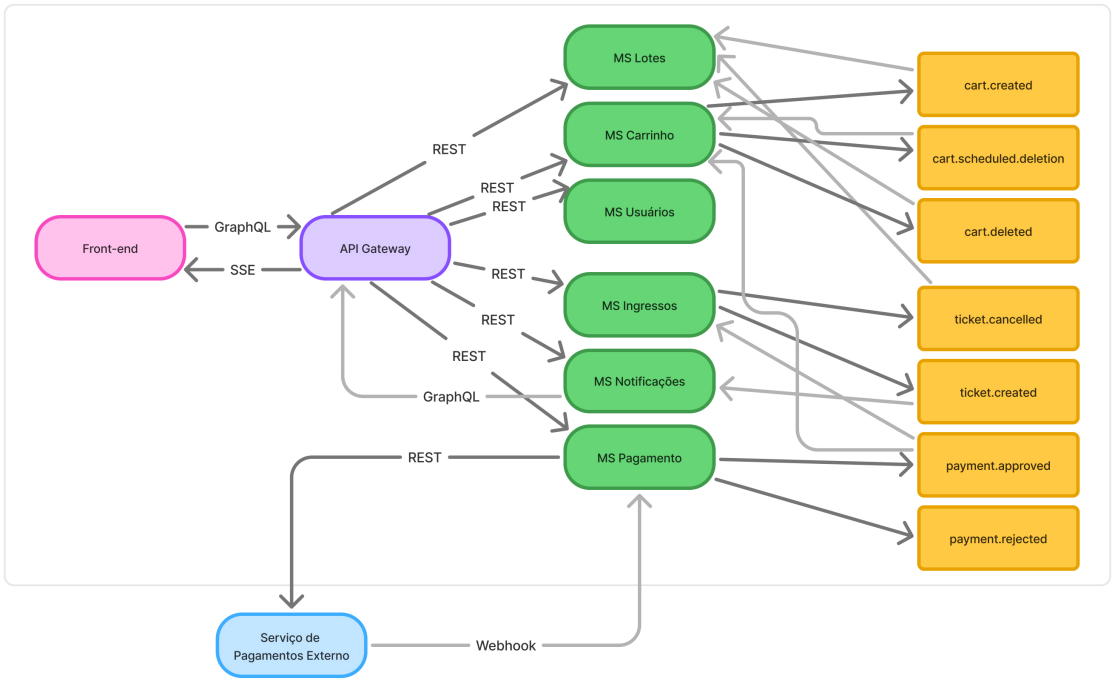

Figura 3. Interação dos Microsserviços

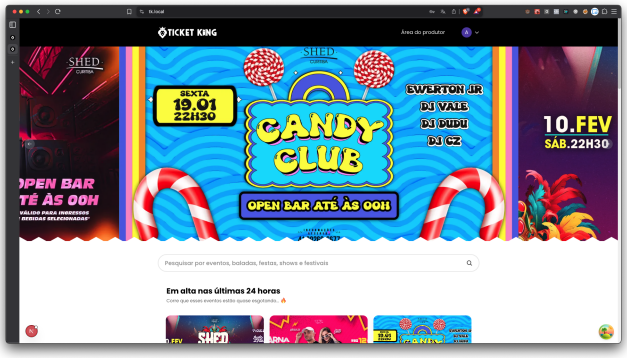

Figura 4. Página Inicial do Sistema

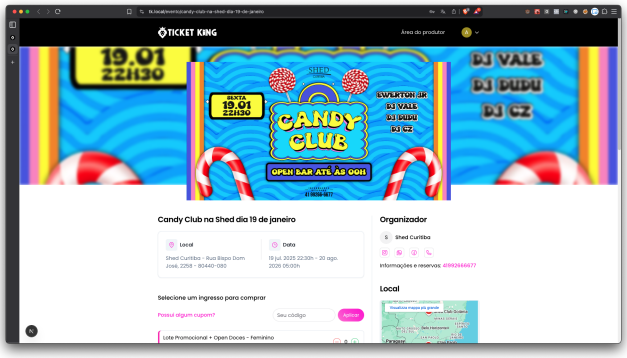

Figura 5. Página de Evento

3.5 Descrição da Avaliação

Os experimentos foram planejados para reproduzir situações reais de operação do sistema, considerando o número de usuários simultâneos, os tipos de requisições realizadas e o volume de tráfego gerado. No contexto de uma loja virtual de venda de ingressos, são comuns picos súbitos de acesso durante o lançamento de eventos de alta demanda, concentrando grande volume de requisições em rotas específicas da aplicação. Esse cenário fundamentou o delineamento dos testes de carga conduzidos neste estudo.

Para a execução dos testes de carga foi utilizado o *Apache JMeter*, responsável pela geração de requisições concorrentes e coleta das métricas de desempenho. A análise estatística dos dados obtidos foi realizada em *Python*, com o auxílio das bibliotecas *Pandas* para processamento dos dados e *Matplotlib* para visualização gráfica dos resultados.

Foram definidos cinco cenários correspondentes às principais operações do sistema:

- Cenário 1: listagem de eventos ativos no sistema;
- Cenário 2: consulta detalhada de um evento específico;
- Cenário 3: criação de carrinho para realizar uma compra;
- Cenário 4: busca de carrinho com dados do usuário e dos eventos adicionados;
- Cenário 5: consulta de registro de pagamento para posterior disponibilização dos ingressos.

Cada cenário foi submetido a uma carga de 500 usuários simultâneos, com *ramp-up* de 60 s e duração total de 300 s. Esses parâmetros foram definidos com base nos picos de vendas observados no sistema em produção.

Os testes foram executados com 1 e 10 *workers*, permitindo avaliar a escalabilidade horizontal das arquiteturas. Um *worker* é uma unidade de trabalho que processa as requisições dos usuários, sendo, neste caso, instâncias de execução do

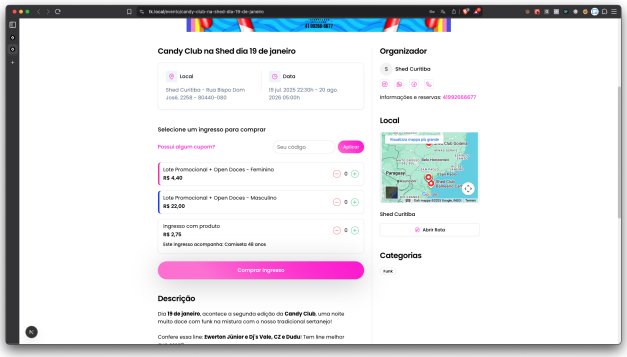

Figura 6. Página de Evento - Listagem de Lotes

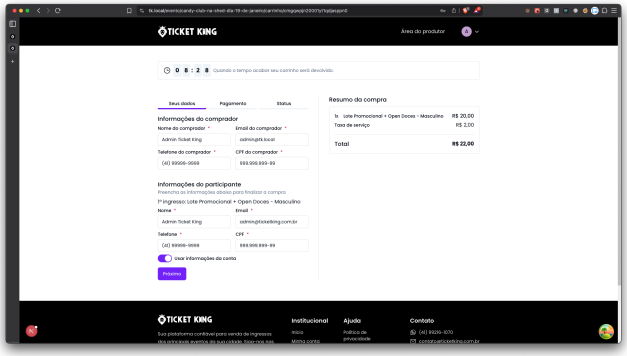

Figura 7. Carrinho - Pagamento

sistema que trazem paralelismo no processamento de requisições.

Foram utilizadas as seguintes métricas de desempenho:

- Tempo de Resposta (ms): intervalo de tempo entre o envio da requisição e o recebimento da resposta pelo usuário;
- Tempo de Resposta - Percentil 90, 95 e 99 (ms): valores abaixo dos quais 90%, 95% e 99% das requisições foram concluídas, respectivamente;
- *Throughput* (requisições por segundo): taxa de processamento de requisições por segundo;
- Taxa de Erro (%): percentual de requisições que falharam durante os testes.

O ambiente de execução foi controlado, com ambas as arquiteturas hospedadas na mesma máquina, garantindo condições equivalentes para a comparação: sistema operacional macOS Sequoia 15.6, processador Intel Core i7 (9ª geração), 16 GB de RAM, banco de dados *MySQL* 8.0 para o monólito e PostgreSQL 15 para os microsserviços.

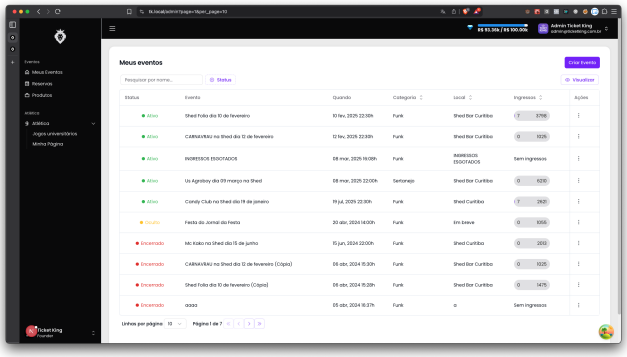

Figura 8. Painel de Administração

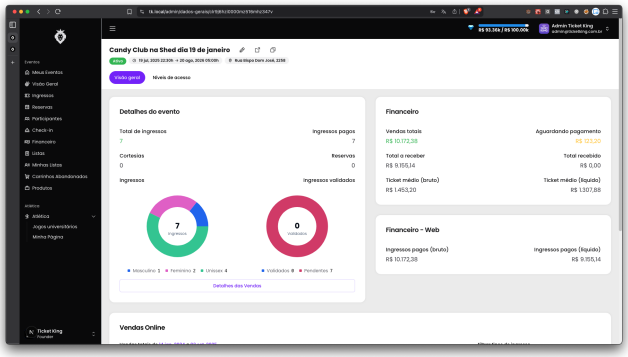

Figura 9. Administração - Visão Geral do Evento

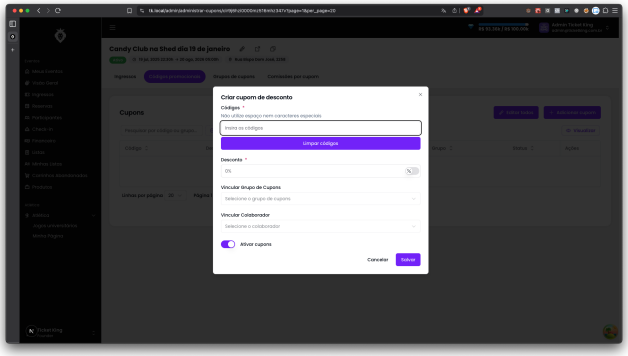

Figura 10. Painel de Criação de Cupom

4 Avaliação de Desempenho

Esta seção apresenta os resultados obtidos a partir dos experimentos descritos na Seção 3.5. São analisados os indicadores de desempenho das arquiteturas monolítica e em microsserviços sob condições equivalentes de carga, com o objetivo de comparar seu comportamento quanto a desempenho e escalabilidade, bem como identificar os possíveis benefícios e limitações decorrentes da migração para uma arquitetura baseada em microsserviços.

4.1 Resultados com 1 *worker*

Essa seção detalha os resultados obtidos nos cinco cenários considerando um único *worker*.

4.1.1 *Cenário 1*

A Tabela 2 apresenta os resultados obtidos para o cenário de listagem de eventos com um *worker*. Embora o tempo médio indique leve vantagem do monólito, essa métrica é influenciada por valores extremos. A análise da mediana revela melhor desempenho dos microsserviços, aproximadamente 18% inferior ao tempo do monólito, refletindo uma experiência mais rápida para a maioria dos usuários. Comportamento semelhante ocorre nos percentis 90 e 95. Já o percentil 99 evidencia picos de latência na arquitetura distribuída, explicando o aumento da média.

A Figura 11 ilustra o comportamento temporal das latências, mostrando estabilidade em ambas as arquiteturas. Em relação ao *throughput*, a Figura 12 demonstra maior pico do monólito (aproximadamente 230 requisições (req) por segundo (s)) frente aos microsserviços (cerca de 200 req/s).

Tabela 2. Resultados do cenário 1 com 1 *worker*.

Métrica	Monolítico	Microsserviços
Requisições processadas	136.057	119.681
Tempo médio de resposta (ms)	838,82	953,54
Tempo mediano (ms)	723,00	595,00
Percentil 90 (ms)	790,00	748,00
Percentil 95 (ms)	822,00	816,95
Percentil 99 (ms)	8.982,91	23.493,94
Throughput (transações/s)	226,43	199,23
Taxa de erro (%)	0,00%	0,00%

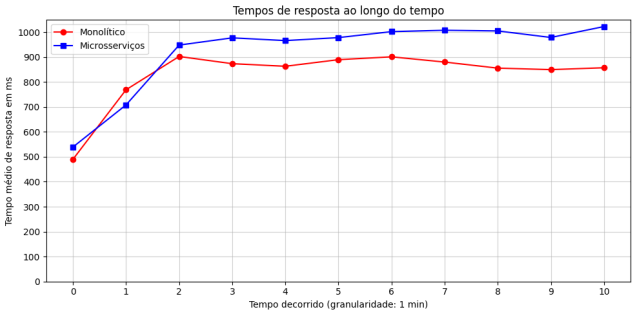

Figura 11. Tempo de resposta ao longo do tempo no Cenário 1

4.1.2 Cenário 2

Os resultados da Tabela 3 demonstram desempenho superior dos microsserviços em relação ao *throughput*, aproximadamente 55% maior que o monólito. O sistema monolítico apresentou saturação com tempos próximos de 5 s, apresentando uma latência elevada e consistente, com média de 4.532 ms e mediana de 5.067 ms. Já os microsserviços exibiram padrão bimodal, com baixas latências para a maioria das requisições e picos pontuais refletidos nos percentis mais elevados. Para 90% das requisições, o desempenho foi excelente, com mediana de 240 ms e percentil 90 de 331 ms.

Tabela 3. Resultados do cenário 2 com 1 *worker*.

Métrica	Monolítico	Microsserviços
Requisições processadas	30.052	46.600
Tempo médio de resposta (ms)	4.532,74	2.914,42
Tempo mediano (ms)	5.067,00	240,00
Percentil 90 (ms)	5.284,00	331,00
Percentil 95 (ms)	5.408,90	54.788,70
Percentil 99 (ms)	5.797,99	56.039,99
Throughput (transações/s)	98,83	153,72
Taxa de erro (%)	0,00%	0,00%

A Figura 13 mostra o acúmulo gradual de latências mais elevadas, enquanto a Figura 14 evidencia maior estabilidade dos microsserviços ao longo do teste.

4.1.3 Cenário 3

Conforme apresentado na Tabela 4, a arquitetura em microsserviços apresentou desempenho superior neste cenário, alcançando *throughput* mais que duas vezes maior em comparação ao monólito. Enquanto o monolítico mostrou comportamento próximo à saturação, com latências frequentemente acima de 5 s, os microsserviços obtiveram tempos medianos significativamente inferiores, refletindo maior agilidade para a maior parte das requisições.

Entretanto, observaram-se picos isolados de latência na arquitetura distribuída, impactando os percentis 95 e 99 e

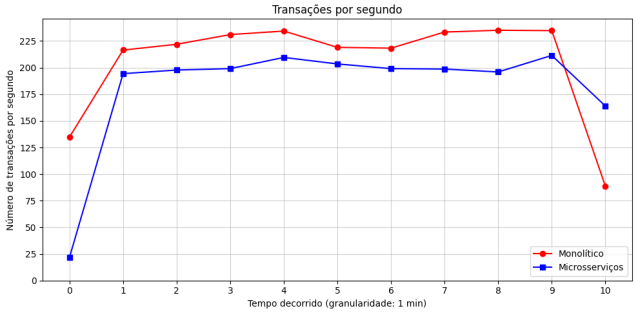

Figura 12. Transações por segundo no Cenário 1

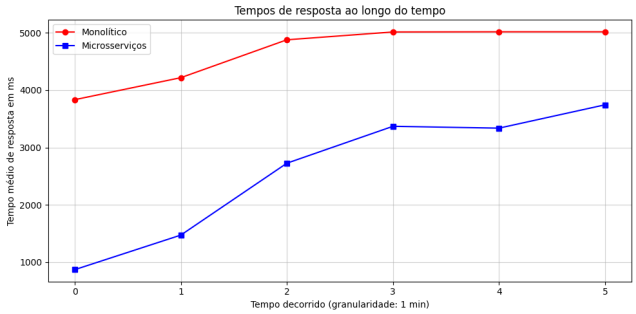

Figura 13. Tempo de resposta ao longo do tempo no Cenário 2

elevando a média de resposta. Esses valores indicam sobrecarga pontual relacionada à comunicação inter-serviços e ao processamento concorrente. Ainda assim, a maioria das requisições apresentou desempenho superior ao do monólito, conforme ilustrado na Figura 15. O *throughput* ao longo do teste, apresentado na Figura 16, reforça a maior capacidade de processamento sustentado dos microsserviços.

Tabela 4. Resultados do cenário 3 com 1 *worker*.

Métrica	Monolítico	Microsserviços
Requisições processadas	28.307	60.418
Tempo médio de resposta (ms)	4.821,84	2.248,20
Tempo mediano (ms)	5.314,00	245,00
Percentil 90 (ms)	5.469,00	382,00
Percentil 95 (ms)	5.518,00	51.829,95
Percentil 99 (ms)	5.630,00	55.581,00
Throughput (transações/s)	92,63	199,36
Taxa de erro (%)	0,00%	0,00%

4.1.4 Cenário 4

Os resultados apresentados na Tabela 5 indicam maior robustez da arquitetura baseada em microsserviços neste cenário, que apresentou maior *throughput* e ausência de falhas durante a execução dos testes. Em contraste, o sistema monolítico registrou uma taxa de erro de 6,25%, além de tempos médios e medianos de resposta superiores a 5 s, caracterizando comportamento de saturação sob a carga aplicada.

A variação temporal das latências, apresentada na Figura 17, evidencia o aumento progressivo dos tempos de resposta em ambas as arquiteturas, embora de forma mais acentuada no monólito. Por sua vez, a Figura 18 demonstra a manutenção de maior volume de transações por segundo pelos microsserviços ao longo de todo o teste, sem que o monólito conseguisse superá-los em nenhum momento.

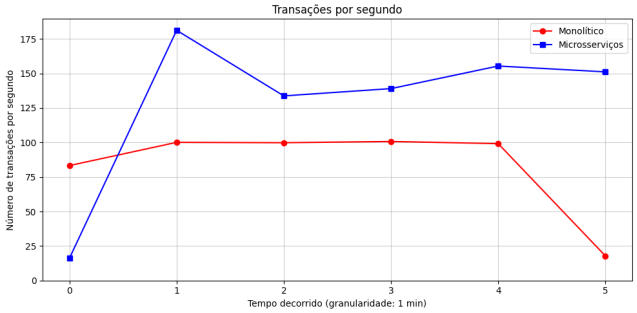

Figura 14. Transações por segundo no Cenário 2

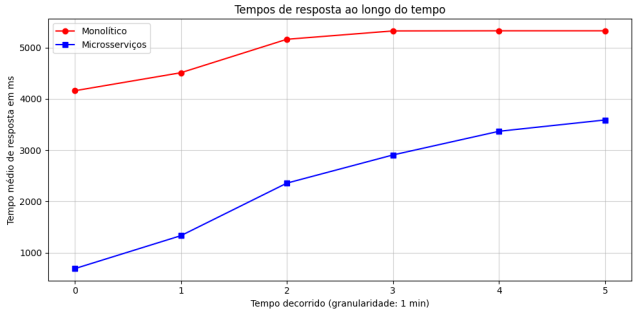

Figura 15. Tempo de resposta ao longo do tempo no Cenário 3

Tabela 5. Resultados do cenário 4 com 1 *worker*.

Métrica	Monolítico	Microsserviços
Requisições processadas	28.906	69.385
Tempo médio de resposta (ms)	4.718,56	1.956,33
Tempo mediano (ms)	5.250,00	241,00
Percentil 90 (ms)	5.471,00	316,00
Percentil 95 (ms)	5.542,00	31.355,90
Percentil 99 (ms)	5.846,00	54.694,98
Throughput (transações/s)	94,83	229,26
Taxa de erro (%)	6,25%	0,00%

4.1.5 Cenário 5

De acordo com os dados apresentados na Tabela 6, o sistema monolítico apresentou maior *throughput* e menores tempos médios de resposta neste cenário. No entanto, a arquitetura em microsserviços manteve vantagem na mediana e no percentil 90, reiterando melhor desempenho para a maior parte das requisições, apesar da ocorrência de valores extremos refletidos nos percentis 95 e 99.

O comportamento dos tempos de resposta é apresentado na Figura 19, que indica maior estabilidade do monólito, com latências próximas a 1.600 ms ao longo da execução. Já os microsserviços apresentaram tempos próximos de 2.000 ms após o primeiro minuto de teste. A análise do *throughput*, apresentada na Figura 20, confirma o maior pico sustentado pela arquitetura monolítica neste cenário específico.

Tabela 6. Resultados do cenário 5 com 1 *worker*.

Métrica	Monolítico	Microsserviços
Requisições processadas	94.045	62.200
Tempo médio de resposta (ms)	1.440,40	2.181,35
Tempo mediano (ms)	1.609,00	236,00
Percentil 90 (ms)	1.690,00	328,00
Percentil 95 (ms)	1.720,00	32.988,15
Percentil 99 (ms)	1.768,00	54.819,99
Throughput (transações/s)	311,84	205,57
Taxa de erro (%)	0,00%	0,00%

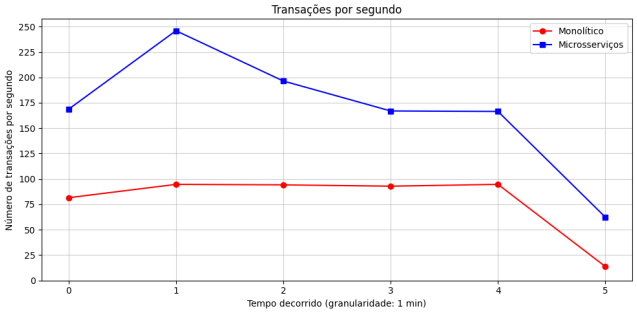

Figura 16. Transações por segundo no Cenário 3

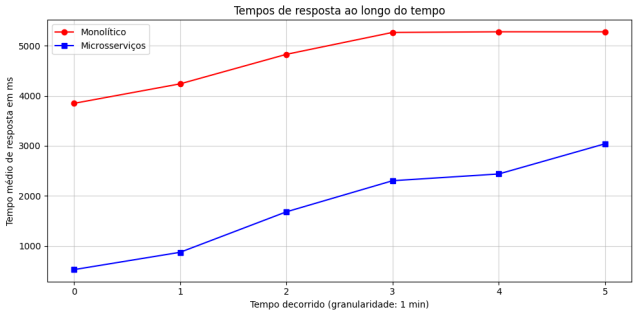

Figura 17. Tempo de resposta ao longo do tempo – Cenário 4

4.2 Resultados com dez *Workers*

Nas execuções com dez *workers* a arquitetura em microsserviços teve uma clara vantagem de desempenho em todos os cenários, evidenciando uma dificuldade do sistema monolítico em trabalhar com instâncias paralelas para melhorar o desempenho. Como houve vantagem da arquitetura em microsserviços em todos os cenários, é apresentado somente o resultado para o cenário 4 de testes com dez *workers*.

A Tabela 7 mostra o comportamento observado com múltiplas instâncias. Os microsserviços apresentaram redução expressiva nos tempos de resposta e aumento do *throughput*, baixando o tempo de resposta médio para 538,12 ms e aumentando o *throughput* para 836,20 transações por segundo, mostrando uma melhor capacidade de escalabilidade de desempenho utilizando mais *workers* para processamento. Em contrapartida, o monólito sofreu degradação de desempenho e elevação da taxa de erro.

4.3 Síntese dos Resultados

De forma geral, os experimentos indicam que a arquitetura em microsserviços apresentou melhor desempenho típico (mediana e percentil 90) e superior capacidade de escalabilidade com o aumento do número de *workers*. Apesar da ocorrência de picos de latência em percentis elevados, seu comportamento sob alta concorrência foi mais estável e eficiente.

A análise dos percentis 95 e 99 indica maior variabilidade nos tempos de resposta dos microsserviços em comparação ao monólito em determinados cenários. Esse comportamento pode estar relacionado a características inerentes a sistemas distribuídos, como o *overhead* de comunicação inter-serviços, custos de serialização e a coordenação entre componentes. Sob alta concorrência, esses fatores amplificam oscilações de latência, especialmente em requisições com múltiplas interações, ainda que o desempenho típico permaneça superior.

A arquitetura monolítica demonstrou desempenho com-

Tabela 7. Resultados do cenário 4 com dez *workers*.

Métrica	Monolítico	Microsserviços
Requisições processadas	24.863 (-13,99%)	**251.156 (+261,97%)**
Tempo médio de resposta (ms)	5.495,64 (+16,47%)	**538,12 (-72,49%)**
Tempo mediano (ms)	**1.172,00 (-77,67%)**	538,00 (+123,24%)
Percentil 90 (ms)	15.555,00 (+184,32%)	710,00 (+124,69%)
Percentil 95 (ms)	15.561,00 (+180,78%)	**779,95 (-97,51%)**
Percentil 99 (ms)	48.303,87 (+726,27%)	**962,97 (-98,24%)**
Throughput (transações/s)	81,40 (-14,16%)	**836,20 (+264,74%)**
Taxa de erro (%)	23,20% (+271,2%)	0,00% (0%)

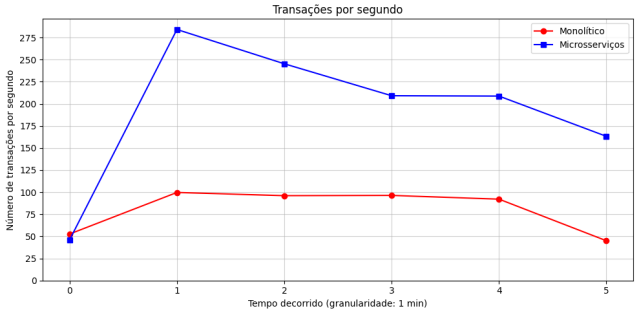

Figura 18. Transações por segundo no Cenário 4

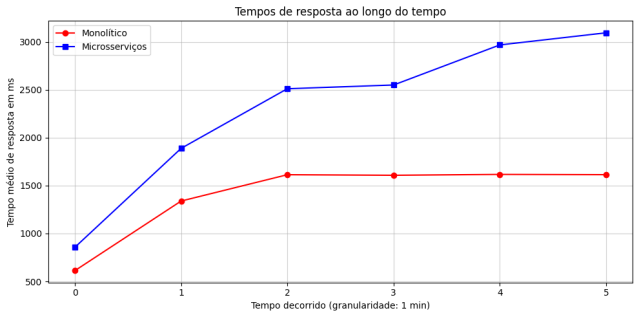

Figura 19. Tempo de resposta ao longo do tempo no Cenário 5

petitivo em cenários de baixa complexidade, porém apresentou limitações sob cargas elevadas e execução paralela intensiva, refletindo menor flexibilidade de escalabilidade.

As arquiteturas avaliadas foram implementadas com diferentes linguagens, bancos de dados e ecossistemas tecnológicos, de forma intencional, a fim de refletir cenários realistas de adoção, nos quais estilos arquiteturais distintos frequentemente estão associados tecnologias específicas. Embora essas diferenças possam influenciar parcialmente os resultados, o ambiente experimental e as condições de carga foram mantidos controlados para preservar a comparabilidade dos experimentos. Assim, privilegia-se o realismo prático da avaliação, ainda que nem todas as variáveis tecnológicas sejam completamente isoladas.

Dessa forma, os resultados indicam que a migração para microsserviços tende a ser mais adequada para aplicações que demandam grande volume de requisições simultâneas, elasticidade de recursos e maior resiliência operacional.

Vale destacar que os resultados obtidos corroboram a literatura apresentada na Seção 2. Estudos como Fowler and Lewis [2014] e Barczak and Barczak [2021] apontam a arquitetura de microsserviços como mais adequada para cenários que demandam escalabilidade horizontal e alta concorrência, especialmente em contraste com as limitações de escalabilidade e implantação associadas às arquiteturas monolíticas.

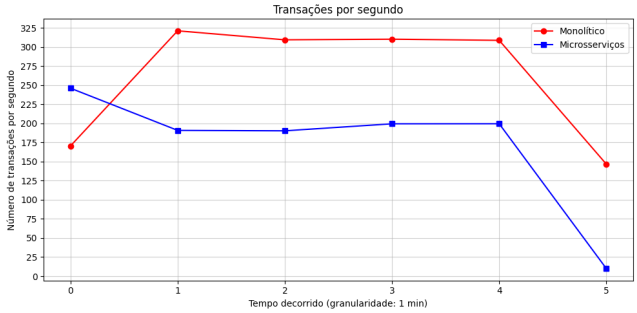

Figura 20. Transações por segundo no Cenário 5

Em particular, o aumento expressivo de vazão com múltiplos *workers* confirma empiricamente a capacidade de distribuição de carga defendida por esses autores. Por outro lado, os picos observados nos percentis mais elevados reforçam os desafios inerentes a sistemas distribuídos descritos por Newman [2015], especialmente no que se refere à complexidade operacional e à latência decorrente da comunicação inter-serviços. Esses achados indicam que os benefícios da arquitetura em microsserviços não são absolutos, mas dependem do contexto de aplicação, da carga de trabalho e das estratégias de implementação adotadas. Em comparação com trabalhos como Faustino *et al.* [2024] e Stradolini [2020], que discutem processos de migração e impactos arquiteturais, este estudo contribui ao apresentar uma avaliação empírica sob múltiplos cenários de carga, ampliando a evidência prática acerca dos *trade-offs* entre as duas abordagens.

5 Conclusão

Este artigo apresentou a migração de um sistema monolítico real de venda de ingressos para uma arquitetura de microsserviços, analisando benefícios, desafios e implicações práticas. A metodologia contemplou levantamento de requisitos, definição das fronteiras dos serviços, implementação e testes de carga para avaliar o desempenho das arquiteturas.

Os resultados indicaram que a adequada decomposição dos serviços, a comunicação assíncrona com RabbitMQ e a orquestração via *API Gateway* foram determinantes para o desempenho observado. Embora o monólito tenha apresentado melhor desempenho em cenários específicos, os microsserviços superaram-no na maioria dos casos, especialmente em escalabilidade horizontal sob alta concorrência.

Conclui-se que microsserviços são mais adequados para sistemas com alta demanda e necessidade de escalabilidade, apesar da maior complexidade operacional, enquanto a arquitetura monolítica permanece indicada para aplicações de menor porte, nas quais simplicidade e menor custo de gerenciamento são prioritários.

Do ponto de vista científico, este trabalho contribui ao fornecer evidências empíricas da migração de um sistema real, avaliando múltiplos cenários de carga por meio de métricas objetivas e enriquecendo a discussão sobre os *trade-offs* entre arquiteturas monolíticas e microsserviços.

Sob a perspectiva profissional, os resultados apoiam a decisão pela adoção de microsserviços, cujos benefícios se mostram mais expressivos em cenários de alta concorrência e necessidade de escalabilidade horizontal, embora impliquem maior complexidade operacional e variabilidade de latência.

Para refletir cenários realistas de adoção, as arquiteturas foram implementadas com diferentes tecnologias, privilegiando o realismo prático da avaliação. Essa escolha implica que parte das diferenças observadas pode estar associada ao ecossistema tecnológico, e não exclusivamente ao estilo arquitetural. Além disso, a análise concentrou-se em um único domínio e foi conduzida em ambiente controlado, o que delimita a generalização dos achados.

Como trabalhos futuros, propõe-se: (i) realizar múltiplas execuções independentes dos experimentos com análise estatística inferencial, a fim de fortalecer a robustez dos resultados; (ii) executar experimentos em ambientes distribuídos reais, avaliando a latência de rede e as falhas de comunicação; (iii) investigar as causas dos picos em percentis elevados com ferramentas de observabilidade e técnicas de *profiling*; (iv) implementar *pipelines* de CI/CD (*Continuous Integration/Continuous Delivery*) para automatizar compilação, testes e implantação dos serviços; (v) investigar mecanismos de tolerância a falhas, como *circuit breakers* e políticas de *retry*; (vi) analisar os custos operacionais das arquiteturas; e (vii) explorar técnicas automatizadas para identificação de fronteiras de microsserviços.

Declarações complementares

Contribuições dos autores

David de Souza, Thiago Medeiros e Willian de Souza foram responsáveis pela conceitualização, investigação, metodologia, *software* e visualização. Ana Cristina Vendramin foi responsável pela supervisão e contribuiu com a edição do manuscrito. Todos os autores leram e aprovaram a versão final do artigo.

Conflitos de interesse

Os autores declaram que não têm nenhum conflito de interesses.

Disponibilidade de dados e materiais

O código do sistema estará disponível mediante solicitação.

Referências

Barczak, A. and Barczak, M. (2021). Performance comparison of monolith and microservices-based applications. In *Proceedings of the 25th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2021)*. International Institute of Informatics and Systemics.

Bass, L., Clements, P., and Kazman, R. (2021). *Software Architecture in Practice*. Addison-Wesley Professional, Boston, 4 edition. The Definitive, Practical, Proven Guide to Architecting Modern Software – Fully Updated with New Content on Mobility, the Cloud, Energy Management, DevOps, Quantum Computing, and More.

Blinowski, G., Ojdowska, A., and Przybylek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:1–1. DOI: 10.1109/ACCESS.2022.3152803.

Faustino, D., Gonçalves, N., Portela, M., and Silva, A. R. (2024). Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation. *Performance Evaluation*, 164:102411. DOI: 10.1016/j.peva.2024.102411.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, CA.

Fowler, M. and Lewis, J. (2014). Microservices: a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html`.

Gluck, A. (2020). Introducing domain-oriented microservice architecture. `https://www.uber.com/en-BR/blog/microservice-architecture`.

GraphQL (2025). Graphql federation. `https://graphql.org/learn/federation`.

Majors, C., Fong-Jones, L., and Miranda, G. (2022). *Observability Engineering: Achieving Production Excellence*. O'Reilly Media, Sebastopol, CA.

Martin, R. C. (2019). *Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software*. Alta Books, Rio de Janeiro. Tradução de: Clean Architecture: A Craftsman's Guide to Software Structure and Design.

Miguel, F. S. (2021). The netflix cosmos platform. `https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad`.

Mohottige, T., Polyvyanyy, A., Buyya, R., Fidge, C., and Barros, A. (2024). Microservices-based software systems reengineering: State-of-the-art and future directions. *arXiv*. DOI: 10.48550/arXiv.2407.13915.

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Sebastopol, CA. Copyright © 2015 Sam Newman. All rights reserved.

PostgreSQL (2025). Postgresql. `https://www.postgresql.org`.

Pressman, R. S. and Maxim, B. R. (2016). *Engenharia de Software: uma abordagem profissional*. AMGH, Porto Alegre, 8 edition.

RabbitMQ (2025). Rabbitmq. `https://www.rabbitmq.com`.

Seedat, M., Abbas, Q., Ahmad, N., Feroz, I., Qureshi, A., and Amelio, A. (2024). Transition strategies from monolithic to microservices architectures: A domain-driven approach and case study. *VAWKUM Transactions on Computer Sciences*, 12:94–110. DOI: 10.21015/vtcs.v12i1.1808.

Sommerville, I. (2011). *Engenharia de Software*. Pearson Prentice Hall, São Paulo, 9 edition. Tradução de Ivan Bosnic e Kalinka G. de O. Gonçalves. Revisão técnica de Kechi Hirama. Título original: *Software Engineering*. ISBN 978-85-7936-108-1.

Stradolini, C. J. (2020). Migração de sistemas monolíticos para microsserviços: Estudo de caso de migração de um módulo de pagamentos de e-commerce. Monografia (bacharelado em ciência da computação), Universidade Federal do Rio Grande do Sul, Porto Alegre.