Uma Análise Comparativa entre Arquitetura Monolítica Modular e Microsserviços em Aplicações Web

Gabriel Raulino
Universidade Federal do Ceará
Quixadá, Brasil
gabrielviana@alu.ufc.br

Carla Bezerra
Universidade Federal do Ceará
Quixadá, Brasil
carlailane@ufc.br

Emanuel Coutinho
Universidade Federal do Ceará
Quixadá, Brasil
emanuel.coutinho@ufc.br

Resumo

Arquitetura de software define a estrutura de um sistema, determinando como componentes interagem entre si. Toda arquitetura possui vantagens e desvantagens. Aplicações monolíticas são difíceis de melhorar e implementar funcionalidades, gerando um forte acoplamento entre as partes do sistema. Considerando as limitações e dificuldades dos monólitos, os microsserviços surgem como uma alternativa. Este trabalho investiga a arquitetura monolítica modular como uma alternativa viável e analisa o desempenho e complexidade estrutural de aplicações web implementadas com arquitetura monolítica modular e microsserviços. A arquitetura monolítica modular se apresentou eficiente e de baixo custo operacional para aplicações com domínios relativamente coesos e fluxos com dependências internas, e arquitetura de microsserviços trouxe maior independência, escalabilidade potencial e flexibilidade tecnológica, com custo adicional em termos de recursos e complexidade.

Palavras-chave

Arquitetura de software, Aplicações monolítica, Microsserviços

1 Introdução

Pressman and Maxim [17] definem que a arquitetura de software estabelece a estrutura fundamental de um sistema, determinando componentes, responsabilidades e interações. Uma arquitetura bem estabelecida permite atender requisitos como escalabilidade, manutenção e flexibilidade, sendo fundamental para sistemas que precisam evoluir rapidamente. Bass et al. [1] argumentam a importância da arquitetura de software, destacando três razões principais: a arquitetura fornece uma abstração do sistema, facilitando a comunicação entre os envolvidos no projeto; destaca desde o início decisões que terão impactos futuros no sistema; a arquitetura representa um modelo suficientemente pequeno e compreensível da estruturação do sistema e de como seus componentes se relacionam. Nesse cenário, a demanda crescente por sistemas de software levou a diversas abordagens para defini-los e mantê-los, incluindo o gerenciamento e a evolução de suas arquiteturas de software [2].

Muitas aplicações monolíticas ainda estão em uso e sendo mantidas ativamente hoje em dia [9]. Aplicações construídas em uma arquitetura monolítica são difíceis de melhorar e implementar funcionalidades, pois os componentes são desenvolvidos de forma entrelaçada em uma única unidade, gerando um forte acoplamento entre as partes do sistema [3]. Fowler and Lewis [12] apontam que a frustração com aplicações monolíticas, pois dificilmente é possível manter sua estrutura modularizada, tornando complicado realizar alterações que não afetem outras partes do sistema. Diante disso, observa-se que essa abordagem pode apresentar limitações em cenários de evolução.

Considerando as limitações e dificuldades dos monólitos diante da crescente demanda por sistemas escaláveis, flexíveis e distribuídos, os microsserviços surgem como uma solução. Os microsserviços ganharam destaque em grandes empresas como Netflix e Amazon, que buscaram resolver desafios de acoplamento, escalabilidade e autonomia de equipes enfrentados por arquiteturas monolíticas [12].

No entanto, essa popularização levou muitas organizações a adotarem microsserviços prematuramente, sem considerar o nível de maturidade técnica e a complexidade envolvida, ocasionando que várias dessas empresas não obtivessem os benefícios esperados com a migração para os microsserviços, enfrentando dificuldades em relação aos altos custos e complexidade dessa arquitetura [19]. Fowler and Lewis [12] alertam que, embora os microsserviços tragam benefícios reais, eles também impõem desafios substanciais, como a necessidade de automação robusta de *deploy*, testes distribuídos e gerenciamento de falhas. Quando essas exigências não são bem atendidas, o sistema pode se tornar ainda mais difícil de manter do que um monólito tradicional. Em grandes monólitos, qualquer pequena mudança pode afetar o sistema inteiro e exigir reimplantação completa [20]. Já microsserviços permitem um *deploy* independente e isolamento entre serviços, favorecendo manutenção e evolução separadas.

Diante dessas dificuldades, diversas organizações passaram a buscar abordagens intermediárias que combinem a simplicidade operacional do monólito com os benefícios estruturais da modularização. Nesse contexto, a arquitetura monolítica modular emerge como uma solução arquitetural que mantém a aplicação como um único serviço implantável, mas internamente estruturada em módulos coesos e de baixa dependência entre si. Segundo Su and Li [19], essa arquitetura se destaca como uma alternativa mais equilibrada, especialmente para projetos em fases iniciais, evitando os custos iniciais elevados dos microsserviços e, ao mesmo tempo, mitigando os problemas estruturais dos monólitos tradicionais. Fowler [11] argumenta que iniciar com um monólito bem modularizado é, na maioria dos casos, a melhor abordagem para novos sistemas, pois permite ganhos de produtividade e simplicidade, além de oferecer uma base sólida para futuras transições arquiteturais. A escolha do projeto arquitetônico adequado é essencial para garantir a sustentabilidade e a escalabilidade a longo prazo no desenvolvimento de software, que evolui rapidamente [18].

Nesse cenário, este trabalho propõe investigar a arquitetura monolítica modular como alternativa viável e estratégica para aplicações web em fase inicial. O objetivo é analisar comparativamente o desempenho e a complexidade estrutural de aplicações web implementadas com arquitetura monolítica modular e arquitetura de microsserviços. Essa abordagem sugere uma organização interna

mais coesa e desacoplada dentro de um único processo de execução, permitindo que o sistema evolua de maneira sustentável e que, se necessário, seja posteriormente refatorado em microsserviços. A relevância deste estudo reside no fato de que há escassez de pesquisas empíricas que analisam comparativamente o impacto da modularização em relação aos microsserviços, considerando tanto métricas de desempenho quanto de complexidade do código.

2 Trabalhos Relacionados

Alguns artigos na literatura trataram de refatorações arquiteturais propondo catálogos de refatorações arquiteturais [6][16]. Outros trouxeram experiências, desempenho e lições aprendidas sobre a decomposição em microsserviços [7][14].

Freitas et al. [13] propuseram uma metodologia para evoluir automaticamente uma aplicação Java monolítica para uma baseada em microsserviços, recebendo código Java e refatorando as classes originais para tornar cada microsserviço independente. A avaliação demonstrou que a ferramenta consegue refatorar com sucesso 80% das aplicações testadas.

Faustino et al. [10] investigaram os impactos da migração de uma aplicação monolítica para microsserviços, visando avaliar como diferentes estágios da migração afetam o desempenho do sistema e o esforço de manutenção. Embora os microsserviços tenham oferecido maior independência e elasticidade, o custo de orquestração e desafios de rede compensam parte desses benefícios, sobretudo em sistemas de pequeno a médio porte.

Peixoto and Disperati [15] discutiram sobre condições mais adequadas para a adoção de cada abordagem, tendo como base dois estudos de caso reais de migração. Os resultados revelaram que a escolha entre uma arquitetura monolítica ou microsserviços depende fortemente do contexto do projeto. A arquitetura monolítica mostra-se mais vantajosa em cenários de menor complexidade, com orçamentos limitados e equipes reduzidas, enquanto os microsserviços se destacam em projetos de grande porte e com necessidade de escalabilidade dinâmica.

Velepucha and Flores [20] compararam arquiteturas de software monolítica, monolítica modular e microsserviços, discutindo desafios e abordagens na decomposição de monólitos e o papel da arquitetura monolítica modular como uma etapa intermediária estratégica para a evolução de sistemas. Os resultados indicaram que a arquitetura monolítica, inicialmente simples e eficiente, torna-se um gargalo em manutenibilidade, escalabilidade e de lançamento com o crescimento da aplicação. Já microsserviços oferecem escalabilidade horizontal, alta tolerância a falhas e agilidade, mas introduzem complexidade de orquestração, desafios de consistência de dados, maior superfície de ataque e sobrecarga operacional.

Diferentemente dos trabalhos analisados, esta pesquisa combina métricas quantitativas de desempenho e complexidade estrutural em um ambiente em nuvem, comparando especificamente a arquitetura monolítica modular e sua migração para microsserviços.

3 Materiais e Métodos

3.1 Domínio da aplicação

A aplicação desenvolvida neste trabalho tem como propósito representar, de forma simplificada e controlada, o domínio de um sistema de comércio eletrônico (*e-commerce*), servindo como base para a análise comparativa entre as arquiteturas apresentadas. O objetivo não é reproduzir um sistema comercial completo, mas sim simular os principais comportamentos de um ambiente real que envolve interações entre módulos independentes, transações de negócios e operações de leitura e escrita.

O sistema contempla um fluxo típico de compra online, incluindo usuários, produtos, carrinhos e pedidos. O funcionamento geral consiste em permitir que um usuário cadastrado navegue pelo catálogo de produtos, adicione itens a um carrinho virtual e realize o *checkout* (conversão do carrinho em pedido). Cada uma dessas ações é tratada como uma unidade funcional independente, mas interligada, o que possibilita avaliar o nível de acoplamento e a coesão entre os componentes sob diferentes abordagens arquiteturais.

3.2 Desenvolvimento da aplicação modular

Para implementar as regras de negócio, cada componente foi implementado de forma isolada, respeitando os limites estabelecidos na modelagem arquitetural, com o objetivo de garantir independência funcional, alta coesão e baixo acoplamento entre os módulos. Essa abordagem permite avaliar a arquitetura modular não apenas como uma solução isolada, mas também como uma etapa intermediária e estruturada para a evolução em direção à arquitetura de microsserviços.

A Figura 1 apresenta uma visão geral dessa arquitetura, evidenciando a divisão da aplicação em contextos funcionais correspondentes aos principais domínios do sistema de comércio eletrônico, como autenticação, usuários, produtos, carrinho e pedidos. Cada um desses módulos é representado como uma unidade lógica independente, reforçando a separação de responsabilidades adotada no projeto. Todas as unidades funcionais seguem uma estrutura interna padronizada, composta por camadas de *controller*, *service* e *repository*. Essa padronização garante uniformidade arquitetural e facilita tanto a manutenção quanto a análise estrutural do código, uma vez que cada módulo encapsula suas próprias entidades, regras de negócio e mecanismos de persistência.

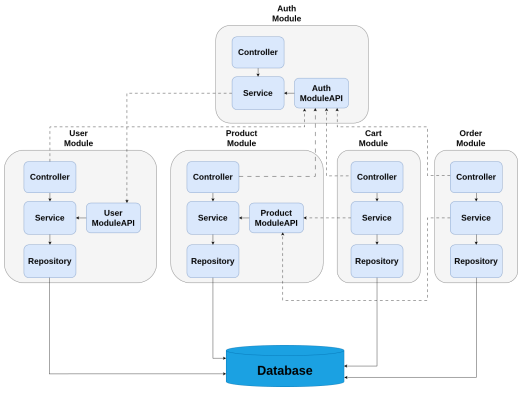

Figura 1: Visão geral da arquitetura monolítica da aplicação

A comunicação ocorre por meio de interfaces públicas, denominadas *ModuleAPI*, também destacadas na Figura 1. Essas interfaces representam os únicos pontos de acesso permitidos entre os domínios distintos, impedindo o acoplamento direto entre implementações internas. As dependências entre esses são, portanto, explícitas

e controladas, sendo representadas no diagrama por meio de conexões direcionais, o que contribui para a clareza da arquitetura e para a redução de dependências cruzadas.

Além da comunicação síncrona realizada por meio de interfaces públicas (*ModuleAPI*), a aplicação adota um modelo de comunicação orientado a eventos, baseado no padrão *Publish/Subscribe*, como um mecanismo complementar de interação entre módulos. Essa abordagem tem como objetivo reduzir ainda mais o acoplamento entre os componentes, permitindo que efeitos colaterais das operações de negócio sejam propagados sem a necessidade de dependências diretas entre as unidades envolvidas.

Outro aspecto evidenciado na Figura 1 é o uso de um banco de dados único, compartilhado por todos os módulos da aplicação. Apesar do compartilhamento da base de dados, cada componente acessa as informações exclusivamente por meio de seus próprios repositórios, mantendo a responsabilidade sobre seus dados e evitando acessos diretos a estruturas pertencentes a outros contextos. Essa decisão preserva o caráter monolítico da aplicação, ao mesmo tempo em que reforça sua organização modular.

3.3 Refatoração para microsserviços

A partir da aplicação monolítica modular apresentada, a evolução arquitetural para uma abordagem baseada em microsserviços foi realizada, a fim de avaliar os impactos da distribuição dos módulos em serviços independentes. Essa transição foi conduzida de forma incremental, aproveitando a estrutura, os limites de domínio e a lógica de negócio previamente estabelecidos na arquitetura modular. Os componentes definidos na aplicação monolítica modular foram convertidos diretamente em serviços independentes, mantendo a responsabilidade funcional de cada contexto. Dessa forma, módulos *product*, *user*, *cart*, *order* e *auth* passaram a ser executados como aplicações autônomas, cada uma com seu próprio ciclo de vida, configuração e persistência de dados. Essa estratégia reduziu o esforço de refatoração, uma vez que os limites de domínio já estavam claramente definidos na arquitetura anterior. A Figura 2 apresenta o resultado da migração dos módulos em serviços isolados.

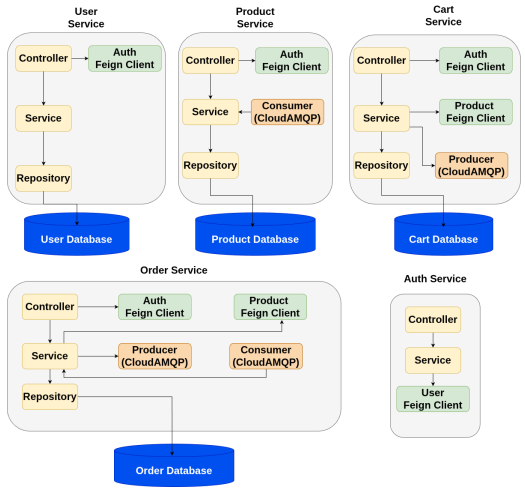

Figura 2: Visão geral da arquitetura de microsserviços

Com a arquitetura estabelecida, os fluxos principais da aplicação foram reavaliados sob a perspectiva de um ambiente distribuído. O fluxo de *checkout*, em particular, passou a envolver comunicação síncrona via rede entre serviços independentes, além da utilização de mensageria para a propagação de eventos.

Outro aspecto relevante da arquitetura de microsserviços, evidenciado na Figura 2, diz respeito à persistência de dados. Diferentemente da aplicação monolítica modular, em que os módulos compartilham um único banco de dados, cada microsserviço possui seu próprio banco de dados, seguindo o padrão *database per service*. A modelagem anteriormente adotada na arquitetura modular, com forte coesão dentro dos domínios e dependências restritas, facilitou significativamente essa transição, reduzindo conflitos de dados e minimizando a necessidade de reestruturação da camada de persistência.

3.4 Execução dos testes de desempenho

Com ambas as versões da aplicação devidamente desenvolvidas, foram executados os testes sistemáticos para comparação de desempenho, avaliando os impactos práticos de cada arquitetura. A análise de desempenho foi conduzida com base nas métricas de tempo de resposta e consumo de recursos (CPU e memória).

As aplicações foram hospedadas na AWS, onde todas as atividades de coleta de dados e a execução dos testes de carga foram realizadas. Utilizaram-se as ferramentas nativas da plataforma *AWS CloudWatch* para monitoramento em tempo real dos recursos e do comportamento da aplicação sob diferentes níveis de acesso. Os testes foram realizados utilizando a solução *Apache JMeter*, uma ferramenta de código aberto que permite simular múltiplos usuários acessando simultaneamente uma aplicação via requisições HTTP. A coleta de informações sobre as requisições foi obtida na própria ferramenta por meio dos ouvintes (*listeners*), que executam junto ao processo de requisições, garantindo informações sobre o tempo de resposta, *status* da requisição, médias e taxas de erro.

3.4.1 Fluxo de login. Concentra a responsabilidade de receber as informações do usuário e retornar um *token* de acesso. Essa operação é a mais custosa em ambas as arquiteturas, pois necessita de uma comunicação síncrona entre os componentes, acesso ao banco de dados e processamento para a geração do *token*. A carga utilizada para esse teste foi de 2000 acessos em um período de 120 segundos, acessando a funcionalidade com o banco contendo 1000 usuários cadastrados. Os valores permaneceram semelhantes para ambas as arquiteturas.

3.4.2 Fluxo de listagem de produtos. Responsável por trazer todos os produtos disponíveis de forma paginada, simulando o que seria um acesso ao catálogo do site. Essa funcionalidade requer pouco processamento, pois consiste apenas em uma consulta ao banco de dados sem nenhum critério de busca. A carga neste teste foi de 4000 acessos em um período de 240 segundos, com a base povoada com 5000 produtos. Valores utilizados nas duas abordagens. Para aumentar o estresse no experimento, os parâmetros de paginação foram randomizados, garantindo que cada requisição traga dados diferentes.

3.4.3 Fluxo de inserção no carrinho. Para evitar processamentos e requisições desnecessárias que pudessem afetar o resultado, antes

do início do teste, todos os usuários da base foram autenticados. A carga utilizada para este teste foi de 3000 acessos em 180 segundos, na qual foram randomizados o usuário, o produto e a quantidade.

3.4.4 Fluxo de checkout. Essa funcionalidade é responsável por transformar os itens do carrinho em um pedido. Esta operação envolve os componentes carrinho, produtos e pedidos. Para executar este teste, foi necessário reutilizar o arquivo de *tokens* que foi gerado e realizar uma inserção no carrinho antes de cada *checkout* para garantir que houvesse produtos a serem transformados em pedidos. Para esse teste, a carga utilizada foi de 4000 acessos em 240 segundos.

3.5 Análise da complexidade estrutural

Para a análise da complexidade estrutural do código das duas versões da aplicação, utilizou-se a métrica de complexidade ciclomática, fornecida pela ferramenta *SonarQube*. Essa métrica quantificou o número de caminhos lógicos independentes existentes em cada método, classe ou módulo, servindo como um indicador direto do esforço necessário para testar, manter e modificar o sistema. A análise foi realizada em dois níveis:

- Comparação por classe: Foi realizada uma comparação classe a classe entre as duas arquiteturas, observando como cada domínio se comporta em termos de complexidade.
- Comparação global: Considerou-se a complexidade total acumulada do projeto em cada abordagem. Isso permitiu uma visão geral da estrutura do sistema, indicando o impacto da decomposição em serviços independentes na carga cognitiva total do projeto

4 Resultados

4.1 Implementação da aplicação monolítica modular e da arquitetrua de microserviços

A aplicação foi desenvolvida utilizando o *framework Spring Boot*, com suporte do *Spring Modulith*, uma extensão projetada para viabilizar a organização modular dentro de um monólito, permitindo a criação de módulos bem definidos com visibilidade explícita das dependências. A aplicação utilizou o PostgreSQL. A modelagem foi concebida de forma a refletir os limites dos domínios da aplicação, priorizando alta coesão dentro de cada contexto funcional e baixo acoplamento entre entidades pertencentes a domínios distintos.

Com a adoção da arquitetura de microsserviços, tornou-se necessário introduzir componentes de infraestrutura adicionais para viabilizar a comunicação e o gerenciamento do ecossistema distribuído. Para esse fim, foram implementados dois serviços transversais: um serviço de descoberta (service discovery), utilizando o Netflix Eureka, e um API Gateway, baseado no Spring Cloud Gateway.

Para comunicação entre os microsserviços, adotou-se uma abordagem híbrida, combinando comunicação síncrona e assíncrona. As interações síncronas, que na aplicação modular eram realizadas por meio das ModuleAPI's, passaram a utilizar o padrão Feign Client, fornecido pelo Spring Cloud OpenFeign. Para a comunicação assíncrona, foi adotado o uso de mensageria baseada em RabbitMQ, utilizando o serviço gerenciado CloudAMQP. Para a execução das aplicações, utilizou-se o Docker Compose na orquestração dos contêineres, permitindo a definição de um ambiente padronizado e reproduzível a partir de arquivos YAML. Para a hospedagem foi utilizada uma instância Ubuntu Amazon EC2 do tipo t3.large.

4.2 Teste de carga da aplicação

4.2.1 Teste de carga - login. A Figura 3 apresenta o comportamento das requisições realizadas durante o teste de *login*, executado com uma carga de 2000 usuários ao longo de 120 segundos. Em termos de latência, a abordagem modular registrou média de 286 ms, com valores mínimo e máximo de 283 ms e 424 ms, respectivamente, enquanto a arquitetura de microsserviços apresentou média de 492 ms, mínimo de 290 ms e pico de 1978 ms, evidenciando maior oscilação no tempo de resposta.

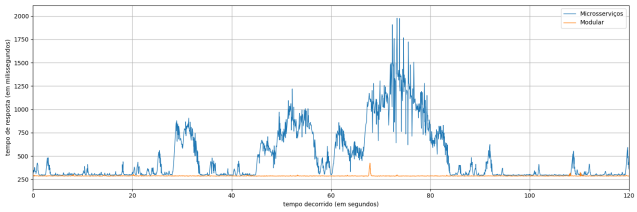

Figura 3: Resultado dos testes de *login*

Por se tratar de uma operação que exige processamento síncrono, o fluxo de *login* é particularmente sensível a qualquer sobrecarga introduzida pela arquitetura. Nesse cenário, a requisição depende da interação entre os componentes de autenticação (*auth*) e usuários (*users*), além do acesso ao banco de dados e da geração do *token*. Como consequência, em uma arquitetura distribuída, parte dessa comunicação passa a ocorrer entre serviços distintos, o que adiciona latência ao processo.

Tabela 1: Consumo de recursos - Login

Métrica	Arquitetura Modular	Microsserviços
Média CPU	47,82%	62,17%
Pico CPU	71,53%	95,77%
Média Memória	1.020,00 MB	2.923,69 MB
Pico Memória	1.020,00 MB	2.946,00 MB

A Tabela 1 indica que, para uma funcionalidade de autenticação com processamento centralizado e alto grau de dependência entre seus componentes, a arquitetura monolítica modular oferece maior eficiência operacional. A ausência de comunicação distribuída entre serviços reduz a sobrecarga de execução e contribui para um comportamento mais previsível sob carga. O teste de *login* evidencia uma vantagem prática da modularidade interna em cenários onde simplicidade de execução e estabilidade de resposta são prioridades.

4.2.2 Teste de carga - listagem de produtos. A Figura 4 apresenta o comportamento das requisições realizadas durante o teste de listagem de produtos, executado com carga de 4000 usuários ao longo de 240 segundos. Observa-se que ambas as arquiteturas responderam de forma satisfatória ao cenário de leitura, com tempos de resposta próximos entre si. Entretanto, a arquitetura monolítica modular apresentou maior estabilidade, enquanto a arquitetura de microsserviços exibiu maior dispersão nos valores de latência, especialmente nos picos máximos registrados.

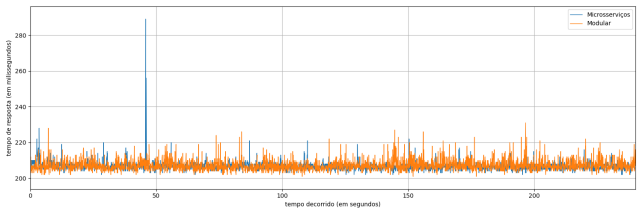

Figura 4: Resultado dos testes de listagem de produtos

Em termos de tempo de resposta, a abordagem modular apresentou média de 207 ms, com valores mínimo e máximo de 202 ms e 231 ms, respectivamente. Já a arquitetura de microsserviços registrou média de 263 ms, mínimo de 202 ms e máximo de 289 ms. Embora a diferença entre as médias seja reduzida, o maior valor observado na arquitetura distribuída indica maior variabilidade sob carga, sugerindo influência do custo adicional de comunicação entre serviços e da sobrecarga do ambiente distribuído.

A Tabela 2 reforça um comportamento que é esperado para uma funcionalidade predominantemente de leitura, com baixa complexidade de processamento. Como a listagem de produtos envolve apenas acesso paginado ao banco de dados, a arquitetura modular consegue atender à demanda com menor sobrecarga operacional. Dessa forma, o teste evidencia que, mesmo em uma funcionalidade simples, a abordagem monolítica modular mantém vantagem em eficiência de recursos, enquanto a arquitetura de microsserviços tende a demandar maior custo para entregar desempenho semelhante.

Tabela 2: Consumo de recursos - Produtos

Métrica	Arquitetura Modular	Microsserviços
Média CPU	7,41%	14,19%
Pico (Máx) CPU	10,55%	17,60%
Média Memória	919,00 MB	2.835,04 MB
Pico (Máx) Memória	919,00 MB	2.845,00 MB

4.2.3 Teste de carga - inserção no carrinho. A Figura 5 apresenta o comportamento das requisições realizadas durante o teste de inserção no carrinho, executado com carga de 3000 usuários ao longo de 180 segundos. Observa-se que ambas as arquiteturas responderam de forma semelhante em termos de latência média, indicando que o custo adicional do fluxo não se refletiu de maneira expressiva no tempo de resposta. Ainda assim, a arquitetura monolítica modular apresentou maior estabilidade nos valores observados, enquanto a arquitetura de microsserviços exibiu maior dispersão no tempo de resposta sob carga.

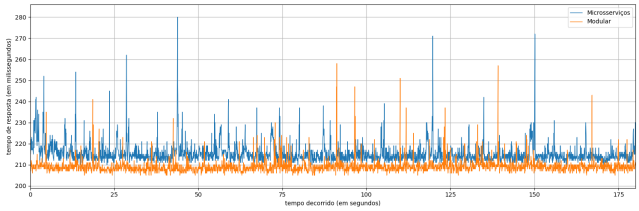

Figura 5: Resultado dos testes de inserção no carrinho

Em termos de latência, a arquitetura monolítica modular apresentou média de 209 ms, com valores mínimo e máximo de 205 ms e 258 ms, respectivamente. Já a arquitetura de microsserviços registrou média de 215 ms, mínimo de 210 ms e máximo de 280 ms. Embora a diferença entre as médias seja pequena, os valores observados indicam que a abordagem modular manteve um comportamento mais estável ao longo do teste, enquanto a arquitetura distribuída apresentou maior dispersão nos tempos de resposta. Esse resultado sugere que, mesmo em uma funcionalidade de baixa complexidade operacional, o custo adicional de comunicação entre serviços pode influenciar a variabilidade da latência.

A diferença mais relevante aparece no consumo de recursos apresentado na Tabela 3. Embora o fluxo de inserção no carrinho seja relativamente simples, ele exige uma verificação síncrona prévia antes da adição dos produtos, envolvendo a comunicação entre os componentes *cart* e *products*. Na arquitetura modular, essa interação ocorre internamente ao processo, com menor sobrecarga de execução. Já na arquitetura de microsserviços, a validação passa a depender de comunicação distribuída, o que aumenta o custo operacional e justifica o maior consumo de CPU e memória observado no experimento.

Tabela 3: Consumo de recursos - Carrinho

Métrica	Arquitetura Modular	Microsserviços
Média CPU	12,95%	19,79%
Pico (Máx) CPU	21,37%	28,26%
Média Memória	937,85 MB	3.266,15 MB
Pico (Máx) Memória	938,00 MB	3.277,00 MB

4.2.4 Teste de carga - checkout. A Figura 6 apresenta o comportamento das requisições realizadas durante o teste de *checkout*, executado com carga de 4000 usuários ao longo de 240 segundos. Observa-se que ambas as arquiteturas apresentaram tempos de resposta reduzidos, o que é coerente com a forma como essa funcionalidade foi implementada. Nesse fluxo, a validação de estoque é realizada de forma síncrona antes da publicação do evento, enquanto a decretação do estoque e a criação do pedido ocorrem de maneira assíncrona, reduzindo o tempo necessário para retornar a resposta à requisição.

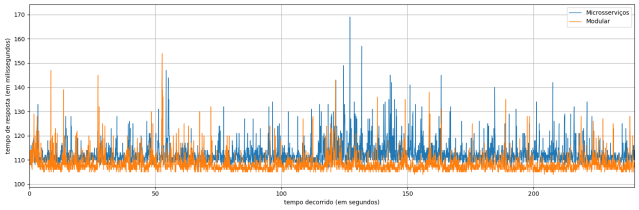

Figura 6: Resultado dos testes de *checkout*

Em termos de latência, a arquitetura monolítica modular apresentou média de 108 ms, com valores mínimo e máximo de 104 ms e 154 ms, respectivamente. Já a arquitetura de microsserviços registrou média de 112 ms, mínimo de 106 ms e máximo de 169 ms. Embora os valores sejam próximos, a abordagem modular manteve

desempenho ligeiramente mais estável, enquanto a arquitetura distribuída apresentou maior dispersão nos tempos de resposta, o que indica o custo adicional associado à coordenação entre serviços.

Os resultados de consumo de recursos apresentados na Tabela 4 mostram que a arquitetura de microsserviços demandou significativamente mais CPU e memória, refletindo o processamento paralelo da criação de pedidos em segundo plano e o custo operacional das comunicações distribuídas. Dessa forma, o teste de *checkout* evidencia que a redução da latência percebida não implica necessariamente menor consumo de recursos, especialmente quando parte relevante da lógica é deslocada para processamento assíncrono. Esse comportamento está diretamente relacionado à natureza assíncrona do fluxo, em que a resposta da operação é entregue antes da conclusão integral do processamento de negócio.

Tabela 4: Consumo de recursos - Checkout

Métrica	Arquitetura Modular	Microsserviços
Média CPU	16,64%	35,50%
Pico (Máx) CPU	24,32%	50,58%
Média Memória	942,46 MB	3.333,46 MB
Pico (Máx) Memória	944,00 MB	3.336,00 MB

4.3 Comparação da complexidade arquitetural

As Tabelas 5 e 6 mostram que a arquitetura de microsserviços apresentou um aumento na complexidade ciclomática total em relação à arquitetura monolítica modular, de 116 para 149. Esse acréscimo relaciona-se principalmente ao custo adicional introduzido pela distribuição dos serviços, uma vez que a comunicação síncrona entre componentes passou a ser realizada por meio de *Feign Clients*, enquanto parte da lógica de negócio foi deslocada para mecanismos assíncronos baseados em mensageria. Além disso, a presença de *producers* e *consumers* em alguns serviços elevou a quantidade de caminhos lógicos internos, refletindo a necessidade de tratar eventos, chamadas remotas e integrações entre contextos distintos.

Tabela 5: Complexidade Ciclomática por Módulo: Arquitetura Modular

Módulo	Controller	Service	Entidade	Total
User	7	14	0	21
Product	6	29	0	35
Cart	4	18	2	24
Order	4	23	1	28
Auth	2	6	0	8
Total				**116**

Tabela 6: Complexidade Ciclomática por Serviço: Arquitetura de Microsserviços

Serviço	Controll.	Service	Entid.	Produc.	Consumer	Total
User	10	16	0	-	-	26
Product	8	28	0	2	4	42
Cart	4	19	2	3	-	28
Order	4	26	1	3	3	37
Auth	4	12	-	-	-	16
Total						**149**

Apesar desse aumento, observa-se que a diferença não representa uma elevação desproporcional da complexidade, mas sim uma consequência natural da separação dos domínios em uma arquitetura distribuída. A arquitetura monolítica modular, por manter a comunicação interna ao processo e reduzir a dependência de mecanismos externos, apresentou uma estrutura mais enxuta e com menor carga de coordenação lógica. Já os microsserviços, embora mais complexos, oferecem maior independência entre os contextos e maior flexibilidade arquitetural, o que justifica o aumento observado.

5 Considerações Finais e Trabalhos Futuros

Este trabalho comparou empiricamente duas abordagens arquiteturais aplicadas à mesma aplicação de comércio eletrônico: uma implementação monolítica modular e sua refatoração para uma arquitetura de microsserviços. A avaliação combinou métricas de desempenho (tempo de resposta, consumo de CPU e de memória) com uma análise estrutural do código (complexidade ciclomática), avaliando aspectos operacionais e de manutenção. Os resultados apontaram que a arquitetura monolítica modular se apresentou como uma alternativa eficiente e de baixo custo operacional para aplicações com domínios relativamente coesos e fluxos com dependências internas. Já a arquitetura de microsserviços trouxe maior independência, escalabilidade potencial e flexibilidade tecnológica, mas com custo adicional em termos de recursos e complexidade. Além disso, foi possível evidenciar que a arquitetura modular serviu como uma boa base para a migração em direção aos microsserviços, permitindo uma transição estruturada e controlada, o que reforça sua adequação para projetos que começam pequenos e evoluem gradualmente em direção a cenários mais distribuídos.

Entre as principais limitações deste estudo, tem-se que a arquitetura de microsserviços foi simulada em um ambiente distribuído executado em uma única máquina física, impondo restrições à análise, impossibilitando explorar escalabilidade horizontal. A execução dos serviços em uma mesma máquina pode ter contribuído para reduzir a latência observada nas chamadas entre componentes, uma vez que parte da comunicação distribuída ocorreu localmente, sem o custo completo da comunicação em rede entre nós distintos. Outra limitação foi a ausência de uma análise financeira entre as duas abordagens. Embora ambas tenham sido hospedadas na infraestrutura da AWS, não foram considerados os custos de execução, armazenamento, tráfego e manutenção associados a cada cenário, o que impede uma avaliação mais ampla sobre a viabilidade econômica. Além disso, o conjunto de testes de carga aplicado foi relativamente restrito, concentrando-se em cenários específicos de uso.

Como trabalhos futuros, propõe-se investigar um cenário distribuído mais real, onde microsserviços sejam executados em instâncias distintas, permitindo avaliar melhor os efeitos da escalabilidade horizontal, bem como o custo e a complexidade associados à criação, manutenção e remoção de réplicas. Outra possibilidade consiste em explorar arquiteturas híbridas, combinando um monólito modular com a extração gradual de serviços críticos, de modo a avaliar estratégias de evolução arquitetural menos abruptas.

Disponibilidade de Artefatos

Artefatos da pesquisa disponibilizados no Anonymous GitHub: aplicação modular [5] e aplicação microsserviços [4]. Dados gerados disponibilizados no ZENODO [8].

Referências

[1] Len Bass, Paul Clements, and Rick Kazman. 2021. *Software architecture in practice*. Addison-Wesley Professional.

[2] Leonardo Braz, Breno França, and Bruno Cafeo. 2025. Dynamic Analysis for Detecting Microservices Antipatterns. In *Anais do XIX Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 67–78. doi:10.5753/sbcars.2025.14576

[3] Luiz Carvalho, Alessandro Garcia, Wesley K. G. AssunÇÃo, Rodrigo BonifÁcio, Leonardo P. Tizzei, and Thelma Elita Colanzi. 2019. Extraction of Configurable and Reusable Microservices from Legacy Systems: An Exploratory Study. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 26–31. doi:10.1145/3336294.3336319

[4] Microservices E commerce Application. 2026. *Anonymous GitHub*. https://anonymous.4open.science/r/Microservices-Ecommerce-3FCA/README.md Acesso em: 12 de julho 2026.

[5] E commerce Modular Monolith. 2026. *Anonymous GitHub*. https://anonymous.4open.science/r/Modular-Ecommece-2304/README.md Acesso em: 12 de julho 2026.

[6] João Daniel, Gabriel Mota, Xiaofeng Wang, and Eduardo Guerra. 2025. Architecture Refactoring Towards Service Reusability in the Context of Microservices. In *Agile Processes in Software Engineering and Extreme Programming*, Sibylle Peter, Martin Kropp, Ademar Aguiar, Craig Anslow, Maria Ilaria Lunesu, and Andrea Pinna (Eds.). Springer Nature Switzerland, Cham, 129–144.

[7] Doğaç Eldenk and H. Alperen Çetin. 2025. Incidents During Microservice Decomposition: A Case Study. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*. Association for Computing Machinery, New York, NY, USA, 805–808. doi:10.1145/3756681.3757005

[8] Artefatos: Uma Análise Comparativa entre Arquitetura Monolítica Modular e Microsserviços em Aplicações Web. 2026. *ZENODO*. https://doi.org/10.5281/zenodo.21328000 Acesso em: 12 de julho 2026.

[9] Guilherme Escher, Maurício Almeida, João Cardozo, Romeu Leite, Ivan Navas, Vinicius Esperança, and Daniel Lucrédio. 2025. JS-Distributor: decomposing monolith applications into microservices. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 879–885. doi:10.5753/sbes.2025.11016

[10] Diogo Faustino, Nuno Gonçalves, Manuel Portela, and António Rito Silva. 2024. Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation. *Performance Evaluation* 164 (2024), 102411.

[11] Martin Fowler. 2015. *MonolithFirst*. https://martinfowler.com/bliki/MonolithFirst.html Acesso em: 1 maio 2026.

[12] Martin Fowler and James Lewis. 2014. *Microservices*. https://martinfowler.com/articles/microservices.html Acesso em: 1 maio 2026.

[13] Francisco Freitas, André Ferreira, and Jácome Cunha. 2021. Refactoring Java Monoliths into Executable Microservice-Based Applications. In *Proceedings of the 25th Brazilian Symposium on Programming Languages* (Joinville, Brazil) *(SBLP '21)*. Association for Computing Machinery, New York, NY, USA, 100–107. doi:10.1145/3475061.3475086

[14] Nuno Gonçalves, Diogo Faustino, António Rito Silva, and Manuel Portela. 2021. Monolith Modularization Towards Microservices: Refactoring and Performance Trade-offs. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. 1–8. doi:10.1109/ICSA-C52384.2021.00015

[15] Evandro Italo Silva Peixoto and Giovani Fonseca Ravagnani Disperati. 2024. Arquitetura Monolítica versus Microsserviços. *Anais da Exposição Anual de Tecnologia, Educação, Cultura, Ciências e Arte do Instituto Federal de São Paulo-Câmpus Guarulhos* 4 (2024).

[16] Rita Peixoto, Filipe F. Correia, Thatiane Rosa, Eduardo Guerra, and Alfredo Goldman. 2026. Refactoring Towards Microservices: Preparing the Ground for Service Extraction. In *Pattern Languages of Programs, People and Practices*, Tsvetelina Plummer, Dennis Dubbert, and Christopher Preschern (Eds.). Springer Nature Switzerland, Cham, 208–250.

[17] Roger S Pressman and Bruce R Maxim. 2021. *Engenharia de software-9*. McGraw Hill Brasil.

[18] Rudra Pratap Singh, Mandhar Thakur, Syed Mehdi Abbas Razavi, Sakshi Sankhla, Krishna Kaushal Singh, and Isha Hiteshbhai Limbasiya. 2025. Monolithic and Microservice Architecture: A Sustainable Approach. In *2025 3rd IEEE International Conference on Industrial Electronics: Developments Applications (ICIDeA)*. 1–6. doi:10.1109/ICIDeA64800.2025.10962984

[19] Ruoyu Su and Xiaozhou Li. 2024. Modular Monolith: Is This the Trend in Software Architecture? *Proceedings of the 1st International Workshop on New Trends in Software Architecture* (2024), 10–13. doi:10.1145/3643657.3643911

[20] Victor Velepucha and Pamela Flores. 2023. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. *IEEE Access* 11 (2023), 88339–88358. doi:10.1109/ACCESS.2023.3305687