# Modernização de Arquitetura de Sistemas: Uma abordagem para transformação digital no sistema financeiro

**Rogério Fernandes da Costa[1], Humberto Lídio Antonelli[2]**

Universidade Federal de Mato Grosso do Sul (UFMS) – Campo Grande – MS – Brasil

rogerio.costa@ufms.br, humberto.antonelli@ufms.br

***Abstract.*** *Currently, financial institutions face the need to develop scalable applications with rapid delivery, highlighting the limitations of monolithic architecture. In this context, adopting emerging technologies, such as microservices architecture, becomes a strategic approach to ensure competitiveness and efficiency. This study proposes a comparative analysis between monolithic and microservices models, considering aspects such as modularity, decoupling, communication, and reuse. Preliminary results indicate that although modularity and maintainability are benefited, decoupling requires greater attention to infrastructure components and route generation that prioritize latency metrics.*

***Resumo.*** *Atualmente, instituições financeiras enfrentam a necessidade de desenvolver aplicações escaláveis e com rápida entrega, o que evidencia as limitações da arquitetura monolítica. Nesse contexto, a adoção de tecnologias emergentes, como a arquitetura de microsserviços, torna-se estratégica para garantir competitividade e eficiência. Este estudo propõe uma análise comparativa entre os modelos monolíticos e de microsserviços, considerando aspectos como modularidade, desacoplamento, comunicação e reuso. Os resultados preliminares indicam que embora a modularidade e a manutenção sejam beneficiadas, o desacoplamento exige maior atenção para componentes da infraestrutura e geração de rotas que priorizem as métricas de latência.*

## 1. Introdução

Paralelamente à evolução histórica da computação, observa-se que as organizações demandam constantemente, por soluções tecnológicas robustas. Ao longo das últimas décadas, essa necessidade impulsionou o desenvolvimento de softwares baseados em distintas arquiteturas, entre as quais se destacam as aplicações monolíticas, o modelo em camadas e a arquitetura de microsserviços (Newman, 2019).

A arquitetura tradicionalmente conhecida como monolítica foi, durante um longo período, amplamente adotada e obteve considerável êxito. Isso se deu, em parte, em razão de sua simplicidade, tanto no desenvolvimento, quanto na implantação (Machado, 2017). Contudo, por concentrar toda a sua lógica de aplicação em um único bloco funcional, as aplicações monolíticas se demonstraram ao longo do tempo, pouco flexíveis, acarretando em maiores desafios relacionados à manutenção e escalabilidade (Newman, 2019; Tapia et al., 2020).

Nos últimos anos, o setor financeiro, assim como outros segmentos, tem seguido a tendência de migrar aplicações monolíticas para arquiteturas baseadas em microsserviços (Mobit, 2023). Essa mudança arquitetônica surge como uma resposta estratégica às limitações do modelo monolítico, possibilitando inclusive, uma migração gradual, na qual novos serviços são incorporados progressivamente para substituir componentes específicos dos sistemas legados. Assim, a arquitetura de microsserviços vem se consolidando como um padrão cada vez mais difundido (Fowler, 2019).

Apesar dos desafios, integrar sistemas legados a tecnologias modernas, é uma estratégia viável para empresas que desejam se modernizar sem abandonar completamente seus processos antigos. Uma abordagem bastante utilizada nesse contexto é o uso de APIs (Interfaces de Programação de Aplicações), que viabilizam a comunicação estruturada entre os sistemas legados, novas aplicações e microsserviços. Essa integração impulsiona mudanças no processo de desenvolvimento de software, promovendo aprendizado organizacional e criando oportunidades para a disseminação de boas práticas e recomendações.

Com base na identificação de uma problemática concreta em uma instituição financeira, esta pesquisa adotou uma abordagem qualitativa se apoiando no método de pesquisa-ação, conforme delineado por Thiollent (2022). Por meio deste estudo, busca-se preencher lacunas no conhecimento técnico e acadêmico, fornecendo informações detalhadas sobre como arquitetos de sistemas e desenvolvedores lidam com a migração a partir de sistemas monolíticos. No contexto prático, destaca-se que êxito dessas iniciativas dependerá da continuidade ao longo do tempo.

## 2. Fundamentação Teórica

Sob a perspectiva da Engenharia de Software, diversos autores têm ressaltado que fatores culturais quando aliados ao ambiente tecnológico, exercem forte influência nas decisões referentes à adoção da arquitetura de microsserviços (Nadareishvili et al., 2016; Fowler, 2019; Newman, 2019; Tapia et al., 2020; Cunha e Santos, 2021; Soldani, 2021; Mobit, 2023; Pathak e Singh, 2023; Ait Said et al., 2024; Bastos, 2024). Neste trabalho focaremos nos aspectos técnicos, relacionados ao ambiente tecnológico.

### 2.1. Aspectos Técnicos

Dentre os aspectos relacionados ao ambiente tecnológico, Fowler (2019) menciona que em um ecossistema bem projetado e sustentável, os microsserviços são separados de toda a infraestrutura. Assim sendo, cabe ao arquiteto de sistema abstrair as questões operacionais de baixo nível, assegurando uma infraestrutura estável e escalável. Desta forma, proporciona-se aos desenvolvedores um ambiente favorável para a construção e execução eficiente desses serviços (Tapia et al., 2020).

No âmbito da arquitetura de software, é essencial reconhecer que a adoção de uma arquitetura baseada em microsserviços impõe desafios organizacionais inerentes. Para que o sistema como um todo opere de maneira adequada, é imprescindível que os serviços interajam de forma coesa e transparente (Fowler, 2019), demandando uma coordenação eficaz entre as equipes, bem como a implementação de mecanismos robustos de comunicação, integração e monitoramento.

Estudos recentes realizados em empresas como Amazon, Spotify e Netflix evidenciaram que o sucesso desse modelo se deu em virtude do alto nível de maturidade organizacional, do domínio de técnicas e ferramentas, bem como, da manutenção constante da infraestrutura distribuída (Nadareishvili et al., 2016; Tapia et al., 2020; Mobit, 2023; Pathak e Singh, 2023). Seguindo esse raciocínio, fica evidente que a migração de sistemas legados para uma arquitetura baseada em microsserviços não deve ser compreendida como uma solução imediata para desafios relacionados ao desempenho ou à escalabilidade. Na visão de Soldani et al. (2021), este tipo de modernização deve ser conduzida como uma reestruturação arquitetural estratégica, que demanda planejamento cuidadoso, testes incrementais e uma compreensão aprofundada dos pontos de integração entre os módulos do sistema.

Compreender esses critérios é o primeiro passo para reunir as informações necessárias que orientarão as decisões das equipes. Entre os principais pontos fortes nos quais a arquitetura de microsserviços se sobressai, temos a compatibilidade, a confiabilidade e a manutenabilidade (Cunha e Santos, 2021; Bastos, 2024). Além disso, esse modelo de arquitetura possibilita a reutilização de código, o que reduz o esforço de desenvolvimento e aumenta a agilidade na entrega de novas funcionalidades, garantindo padronização para a resolução de problemas recorrentes (Ait Said et al., 2024).

Por fim, para que cada microsserviço seja desenvolvido, implantado e mantido como uma aplicação modular e independente, é crucial considerarmos o ferramental necessários para a criação, implantação, gerenciamento e monitoramento. Esses componentes incluem, principalmente, contêineres (como Docker), orquestração de contêineres (como Kubernetes), plataformas como serviço (PaaS) ou infraestrutura como serviço (IaaS), ferramentas como Jenkins, GitLab e AzurePipelines para integração contínua e entrega contínua (CI/CD), bem como, monitoramento e Logging com Grafana, Elasticsearch, Logstash e Kibana (Fowler, 2019; Tapia et al., 2020; Mobit, 2023).

## 3. Arquitetura de Microsserviços

Nesta seção, serão detalhados os principais aspectos da arquitetura proposta para a oferta de produtos financeiros. Considerando que a escalabilidade de microsserviços exige uma infraestrutura robusta e conhecimento especializado, este trabalho parte da premissa de que a utilização da nuvem é um elemento essencial. Nesse contexto, destacam-se as ferramentas de automação DevOps, como Ansible, Docker, Chef e Puppet, cuja adoção, aliada a mecanismos de escalonamento automático, contribui para a otimização do tempo e a redução dos custos operacionais (Tapia et al., 2020).

Com raras exceções, como o Nubank, que já nasceu com DNA tecnológico, à maioria dos bancos no Brasil ainda tem um longo caminho a percorrer para alcançar a digitalização plena. Essa realidade manifesta-se na oferta hibrida de serviços financeiros, em que determinados produtos permanecem disponíveis exclusivamente nos canais físicos, enquanto outros migram progressivamente para plataformas digitais. Diante da necessidade estratégica de expor e integrar funcionalidades dos sistemas legados, a arquitetura de microsserviços emerge como a abordagem mais adequada para viabilizar a modernização tecnológica dessas instituições (Newman, 2019).

Considerando as características inerentes aos sistemas bancários legados, a modernização normalmente ocorre de forma gradual, extraindo as regras de negócio das aplicações monolíticas e transformando-as em serviços independentes, evoluindo até a desativação completa do legado (*Strangler Pattern*). Assim, os clientes seguem utilizando a mesma interface, sem perceber a migração, conforme ilustrado na figura 1.

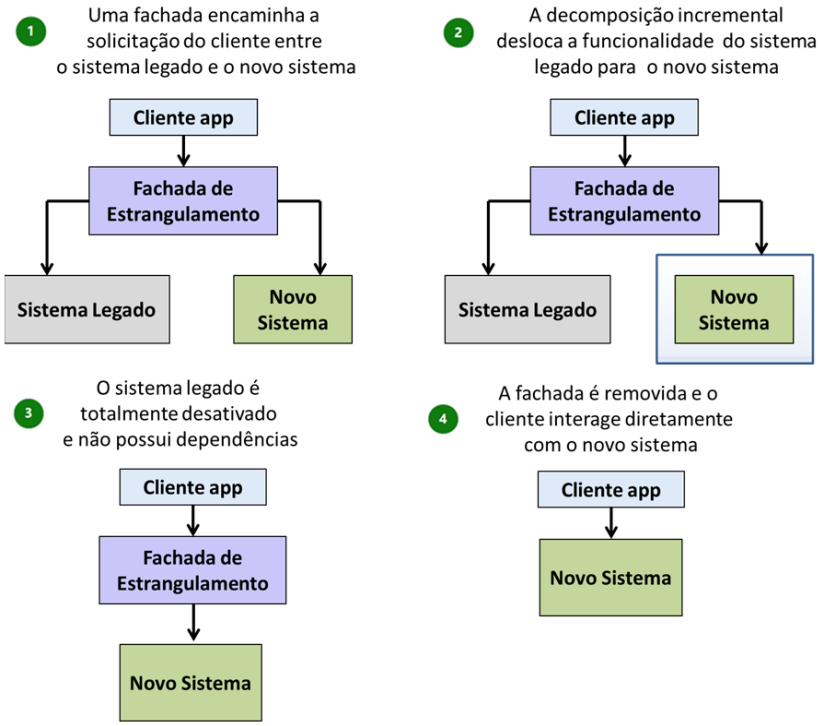


**Figura 1. Strangler Pattern**

A arquitetura baseada em *Strangler Pattern* adota uma estratégia de modernização gradual e controlada, integrando microsserviços a sistemas legados por meio de APIs. Para manter o funcionamento do aplicativo durante a transição, utiliza-se uma fachada (proxy) que roteia as solicitações para o sistema legado ou para os novos serviços. Neste caso em particular, a delimitação dos microsserviços ocorreu a partir do caso de uso, mapeando as funcionalidades essenciais, interações entre entidades e os dados envolvidos em cada interação. Esta atividade apoiou-se no seguinte macro-fluxo:

1. Após se logar no app, o cliente deve visualizar apenas ofertas “pré-aprovadas”.
2. Existe a necessidade de integração entre os microsserviço e o legado, para que de forma síncrona, retorne as ofertas para contratação de produtos financeiros, como por exemplo, determinados tipos de empréstimos e seguros.
3. Estas ofertas personalizadas devem ser exibidas apenas para clientes que atendam determinados critérios de elegibilidade.

## 3.1. Fluxo Funcional Orquestrado

Para atender aos requisitos de negócio, a solução precisou ser projetada para orquestrar as requisições, o que levou o time a discutir os seguintes pontos:

1. Do ponto de vista funcional, o usuário espera ver as ofertas imediatamente, daí a necessidade de uma chamada síncrona do *front-end*.
2. Para aumentar a tolerância a falhas nessa aplicação distribuída, é recomendável usar padrões de projeto como *circuit breaker* e *timeout*.
3. Se o cálculo for custoso e as regras não mudarem com frequência, recomenda-se a utilização de *caching* na validação de elegibilidade.
4. Para atender os critérios de segurança, recomenda-se a implementação de JWT (*JSON Web Token*) entre o *front-end* e o *back-end*. Internamente, o ideal é usar mTLS (*Mutual Transport Layer Security*) ou OAuth2 entre os microsserviços.

Para proporcionar às equipes de desenvolvimento uma compreensão mais clara sobre os papeis dos microsserviços, o fluxo funcional orquestrado foi integrado e analisado dentro de uma visão de arquitetura técnica, conforme ilustrado na Figura 2.

1. Autenticação do cliente: *Front-End* → Autenticação do Serviço → *Token* JWT.
2. Solicitação de ofertas: *Front-End* → BFF/API *Gateway* com *token*.
3. Orquestração de ofertas:
    a. Buscar dados do cliente no legado. Chamada ao *Legacy Adapter* para obter dados de conta/transações
    b. Enviar os dados ao **microsserviço de elegibilidade** para retornar uma lista de produtos permitidos. Ex.: `"emprestimo_garantia", "seguro_auto"`.
    c. Chamar o(s) **serviço(s) de produto** com base na elegibilidade. Exemplo: se elegível para "emprestimo_garantia", chama o serviço correspondente e retorna a(s) oferta(s).
    d. Montar resposta final. O BFF consolida as ofertas e envia para o *front-end*.

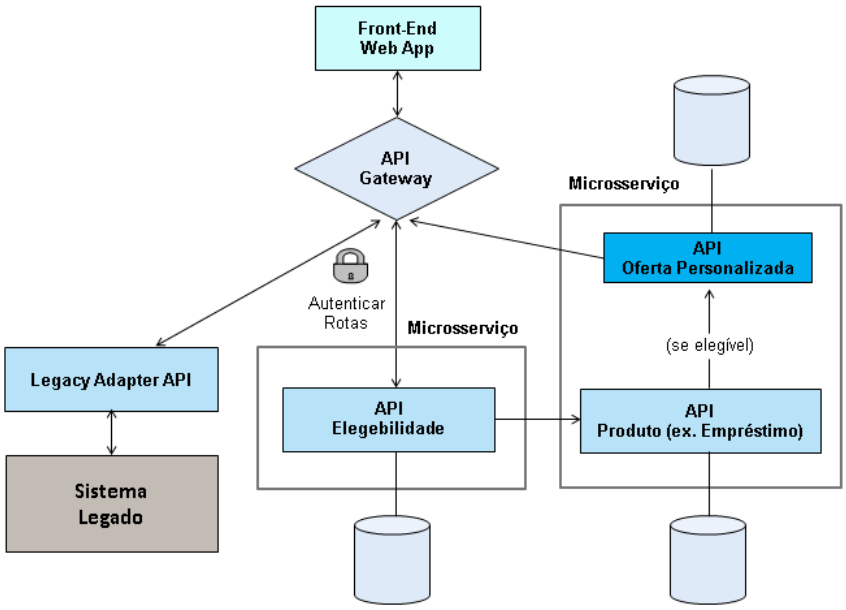


**Figura 2. Fluxo Orquestrado**

## 4. Stack Tecnológica

Esta seção aborda alguns detalhes sobre o conjunto de tecnologias e ferramentas envolvidas no desenvolvimento e implantação de um microsserviço. Partimos do pressuposto que, contrastando com arquiteturas monolíticas, onde geralmente há uma única *stack* para toda a aplicação, na arquitetura de microsserviços podemos escolher as tecnologias mais adequadas, de acordo com cada serviço.

Para se ter uma ideia sobre as principais *stacks* utilizadas no cenário de microsserviços, elas incluem as linguagens de programação como Java, JavaScript, TypeScript, Python, Go (Golang) e C#; envolve também Plataformas, Runtime e Framework como .NET Core, Node.js, Spring Boot e Quarkus. Além disso, para empacotar aplicações e dependências de forma isolada e portátil podemos usar Containers com Docker, apoiando-se no Kubernetes para realizar a orquestração, gerenciando a implantação, escalabilidade e resiliência.

Em arquiteturas de microsserviços, é comum o uso de bancos de dados como PostgreSQL (relacional), MongoDB (NoSQL) e Redis (em memória), geralmente integrados através de APIs REST ou gRPC. Para atender o cenário de comunicação assíncrona podemos usar mensageria com Kafka ou RabbitMQ. No caso do repositório de código e das ferramentas usadas em pipelines de CI/CD, contamos com GitHub, GitLab, GitActions, SonarQube e Jenkins. Completam o ecossistema as ferramentas de Observabilidade e Monitoramento como Prometheus/Grafana para métricas e Loki/ELK para logs.

Na tabela a seguir, um exemplo de *stack* completa na Azure é exibido. Dado esta grande variedade de tecnologias, proporcionar autonomia entre os módulos de um sistema é um ponto importante. Aplicações com maior flexibilidade e portabilidade reduzem o risco de dependência tecnológica (*vendor lock-in*). Essa característica permite que a aplicação seja executada em diferentes plataformas e sistemas operacionais sem grandes modificações no código.

**Tabela 1. Exemplo de stack completa na Azure**

| Camada | Serviço Azure |
|---|---|
| Frontend (App) | Azure AD B2C (para login) |
| API Gateway | Azure API Management (APIM) |
| Microsserviços (.NET) | Azure App Service ou AKS |
| Autorização de acesso | Azure AD B2C + JWT |
| Armazenamento de segredos | Azure Key Vault |
| Logs e Telemetria | Azure Monitor + Application Insights |
| Mensageria (opcional) | Azure Service Bus ou Event Grid |

Ainda que existam algumas ressalvas em relação à escolha da linguagem, Fowler (2019) destaca que independentemente da forma como os demais microsserviços de uma determinada aplicação foram desenvolvidos, graças a esse modelo de arquitetura, os times podem escolher a *stack* mais adequada para cada serviço. Com essa

autonomia, os desenvolvedores passaram a ter mais liberdade, inclusive, para escrever a lógica interna dos microsserviços com qualquer linguagem de programação, usando qualquer banco de dados.

## 5. Abstração e Padronização de Contratos de API

Para viabilizar a independência da *stack*, a arquitetura de sistemas adota camadas de abstração, separando as responsabilidades e encapsulando a lógica interna dos serviços. Por isso, em uma arquitetura de microsserviços, as APIs desempenham um papel fundamental, garantindo que um erro em um microsserviço não necessariamente afetará os outros, ocasionando falhas em cascata. Em suma, estas interfaces técnicas passam a ser vistas como "contratos formais", tornando a comunicação entre microsserviços mais previsível e confiável, facilitando a detecção de erros e a manutenção do sistema.

Considerando esse contexto, cabe ressaltar que a mera existência da interface não garante, por si só, um contrato formal e explicitamente definido. É importante observar que APIs REST e sistemas de mensageria podem estabelecer contratos, porém sua efetividade depende da adoção de práticas adequadas, tais como a utilização de OpenAPI ou Schema Registry. Por outro lado, o protocolo gRPC impõe naturalmente a definição de contratos por meio de arquivos `.proto`.

Tendo em vista que as APIs são frequentemente o principal meio de integração entre microsserviços, e que uma documentação adequada facilita a integração, reduz ambiguidades e acelera o desenvolvimento, torna-se evidente que a documentação da API deve ser concebida como componente integral do produto, não como um elemento acessório. Este princípio reforça a importância da utilização de ferramentas especializadas como Swagger e Postman para especificar os contratos e facilitar testes manuais ou automatizados (Almeida, 2025). Complementarmente, destaca-se a relevância de práticas como a abordagem API-First[1], que prioriza o desenvolvimento de contratos como etapa inicial do ciclo de vida do software.

A abordagem API-First e o Swagger estabelecem uma relação sinérgica no desenvolvimento moderno de APIs. Enquanto o API-First prioriza o design e a especificação da API antes da implementação do código, o Swagger (atualmente denominado OpenAPI Specification) representa uma ferramenta e especificação amplamente adotada para documentar, projetar e testar APIs RESTful. O exemplo a seguir demonstra a aplicação prática dessa metodologia.

### Endpoints Swagger/OpenAPI

**Estrutura básica: Microsserviço de Elegibilidade**

```
openapi: 3.0.0
info:
  title: Serviço de Elegibilidade
  version: 1.0.0
paths:
  /elegibilidade:
    get:
```

[1] Disponível em https://swagger.io/resources/articles/adopting-an-api-first-approach/

```
summary: Verifica elegibilidade de produtos para o cliente
parameters:
  - name: clienteId
    in: query
    required: true
    schema:
      type: string
responses:
  '200':
    description: Lista de produtos elegíveis
    content:
      application/json:
        schema:
          type: object
          properties:
            clienteId:
              type: string
            produtosElegiveis:
              type: array
              items:
                type: string
                enum: [EMPRESTIMO, SEGURO, CARTAO]
```

Na arquitetura de microsserviços, a comunicação entre os componentes é estabelecida por meio de contratos bem definidos, tais como APIs REST ou sistemas de mensagens, assegurando que cada serviço interaja de forma previsível e consistente com os demais. Desta forma, os serviços podem evoluir independentemente em relação à sua stack, sem impactar negativamente uns aos outros. Considerando os benefícios dos contratos formais e reconhecendo que a existência de um contrato formal e explícito não é característica inerente a toda API, a Tabela 2 apresenta as especificidades dos contratos organizadas de acordo com a sua respectiva interface:

**Tabela 2. Contratos formais e desacoplamento**

| | **API REST** | **gRPC** | **Mensageria** |
|---|---|---|---|
| **Descrição** | Padrão baseado no uso de métodos HTTP (GET, POST, PUT, DELETE), proporciona a interação entre recursos a partir de uma estrutura de dados JSON, tornando a comunicação mais previsível e fácil de usar. | Interface de chamada remota baseada em HTTP/2 e Protocol Buffers (protobuf). Altamente performática e eficiente em comunicação binária. | RabbitMQ: Broker de mensagens baseado em filas;<br>Kafka: Plataforma de streaming distribuída que trabalha com logs de eventos baseada em tópicos. |
| **Tipo de Comunicação** | Síncrona | Síncrona, suporta streaming bidirecional. | Assíncrona |

| | | | |
|---|---|---|---|
| **Nível de Desacoplamento** | Médio: exige que o serviço consumidor conheça os endpoints e estrutura dos dados. | Médio: depende de contrato compartilhado (arquivos .proto) | Alto: produtores e consumidores não precisam conhecer detalhes uns dos outros. |
| **Obrigatoriedade de Contrato formal** | Não é obrigatório, mas é recomendado (com OpenAPI/Swagger) | O contrato é definido de forma clara e obrigatória (arquivos .proto). | Não é obrigatório. Para padronização, recomenda-se o uso de Confluent Schema Registry (RabbitMQ) ou Registry (Kafka). |

## 6. Análise do cenário e Resultados

Embora não constitua uma regra absoluta, em arquiteturas baseadas em microsserviços distribuídos em contêineres, é possível observar tempos de resposta superiores em comparação com aplicações monolíticas. Consequentemente, torna-se fundamental realizar uma análise minuciosa sobre a complexidade da comunicação entre os serviços distribuídos e o impacto da latência da rede.

Mediante a execução de testes específicos, fundamentados em métricas do Kubernetes e com o apoio do serviço Azure Traffic Manager, os resultados preliminares evidenciaram a necessidade de ampliar a disponibilidade dos microsserviços e reduzir a latência. Para mitigar essas questões, foram revisados os parâmetros de escalonamento automático de Pods, além disso, adotamos o balanceamento de carga e a utilização de cache para a validação de regras de elegibilidade. Nesse último caso, a API realiza a consulta primeiramente no cache e, na ausência da informação, acessa o banco de dados, atualizando em seguida o próprio cache.

## 7. Conclusão e Trabalhos Futuros

Este estudo descreve alguns dos principais aspectos técnicos envolvidos na abordagem arquitetônica em projetos de modernização e migração de sistemas monolíticos. Embora a literatura apresente diversos estudos relacionados a software e microsserviços, tais estudos raramente abordam o contexto prático no processo de concepção e implementação dessas arquiteturas. Nesse sentido, a presente investigação adota a perspectiva de arquitetos de sistemas e desenvolvedores, uma vez que esses profissionais lidam diretamente com os desafios práticos de escalabilidade, manutenção e integração, oferecendo, portanto, subsídios mais aderentes à realidade do desenvolvimento de soluções tecnológicas.

Como proposta para trabalhos futuros, pretende-se conduzir uma pesquisa junto aos gestores organizacionais para mensurar os resultados financeiros alcançados a partir dessa iniciativa de modernização, contribuindo desta forma, para o debate sobre um tema crucial na atualidade, com impacto direto na formação de profissionais.

## Referências

Ait Said, M. A., Belouaddane, L., Mihi, S., & Ezzati, A. (2024). A Systematic Framework To Enhance Reusability In Microservice Architecture. Int. J. Comput. Digital Syst, 16(1), 189-203.

Almeida, R. M. G. (2025). Boas Práticas em Desenvolvimento de APIs para Integração de Sistemas. Lumen et Virtus, 16(45), 1636-1655.

Bastos, P. A. C., Silva Filho, S. S., de Almeida, D. D., de Oliveira, T. A., & Marinho, M. R. (2024, November). Roteiro de estudos para avançar na Programação: Sugestões iniciais voltadas a Microserviços. In Escola Regional de Informática de Mato Grosso (ERI-MT) (pp. 186-188). SBC.

Cunha, D. S. and Santos, W. H. d. S. (2021). Analise do processo de tomada de decisão para adoção da arquitetura de microsservicos, baseado em requisitos da norma iso-25010 e do s-cube. Pontifícia Universidade Católica de Minas Gerais.

Fowler, S. J. (2019). Microsserviços prontos para a produção: Construindo sistemas padronizados em uma organização de engenharia de software. Novatec Editora.

Machado, M. G. (2017). Micro Serviços: Qual a diferença para arquitetura monolítica.

Mobit, I. A. C. (2023). Technological, organizational, and environmental factors and the adoption of microservices in the financial services sector. Robert Morris University.

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice architecture: aligning principles, practices, and culture. " O'Reilly Media, Inc.".

Newman, S. (2019). Monolith to microservices: evolutionary patterns to transform your monolith. O'Reilly Media.

Pathak, G., & Singh, M. (2023, July). A review of cloud microservices architecture for modern applications. In 2023 World Conference on Communication & Computing (WCONF) (pp. 1-7). IEEE.

Soldani, J., Muntoni, G., Neri, D., and Brogi, A. (2021). The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. Software: Practice and Experience, 51(7):1591–1621.

Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E., & Toulkeridis, T. (2020). From monolithic systems to microservices: A comparative study of performance. Applied sciences, 10(17), 5797.

Thiollent, M. (2022). Metodologia da pesquisa-ação. Cortez editora.