IMPACTO DA ARQUITETURA DE SOFTWARE NA PERFORMANCE E DESENVOLVIMENTO DE APLICAÇÕES WEB

Camile Luz de Alano[1], Matheus Leandro Ferreira[2]

Resumo: A arquitetura de software é um elemento estratégico que influencia diretamente o desempenho, a manutenibilidade e a escalabilidade de aplicações web. Este estudo realiza uma análise comparativa entre três abordagens arquiteturais, model-view-controller, microsserviços e arquitetura limpa, a fim de identificar seus principais trade-offs. Foram desenvolvidas três APIs funcionalmente equivalentes em um sistema de gerenciamento de tarefas utilizando o framework Spring Boot. A metodologia incluiu a execução de testes de estresse com 5.000 usuários concorrentes, utilizando o JMeter para avaliar a latência e a vazão sob carga, e o Prometheus com Grafana para monitorar o consumo de CPU e memória. Os resultados indicaram melhor desempenho da arquitetura model-view-controller, com tempo de resposta até 130% superior ao obtido em microsserviços. A arquitetura limpa destacou-se pela organização e baixo acoplamento, enquanto microsserviços apresentaram maior consumo de recursos, embora ofereçam escalabilidade horizontal. Conclui-se que a escolha da arquitetura depende do contexto do projeto e do equilíbrio entre desempenho, complexidade e facilidade de manutenção.

Palavras-chave: Arquitetura de software; Análise de performance; Microsserviços.

---

[1]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), alanocamile@unesc.net

[2]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), mlf@unesc.net

ABSTRACT: Software architecture is a strategic element that directly influences the performance, maintainability, and scalability of web applications. This study performs a comparative analysis between three architectural approaches—model-view-controller, microservices, and clean architecture—in order to identify their main trade-offs. Three functionally equivalent APIs were developed in a task management system using the Spring Boot framework. The methodology included stress tests with five thousand concurrent users, using JMeter to measure latency and throughput, and Prometheus with Grafana to monitor CPU and memory usage878iu. The results indicated better performance for the model-view-controller architecture, with response times up to 130% higher than those obtained with microservices. The clean architecture stood out for its organization and low coupling, while microservices presented higher resource consumption, although they offer horizontal scalability. It is concluded that the choice of architecture depends on the project context and the balance between performance, complexity, and ease of maintenance.

Keywords: Software architecture; Performance analysis; Microservices;

1 INTRODUÇÃO

O avanço das aplicações web redefiniu profundamente a forma como sistemas são concebidos, desenvolvidos e consumidos em escala global, consolidando-se como elementos indispensáveis em setores corporativos, educacionais e de serviços voltados ao usuário final. Nesse contexto, a arquitetura de software emerge como fator estratégico, pois influencia diretamente o desempenho, a escalabilidade e a manutenção de aplicações que precisam atender a cenários de alta demanda e de rápida evolução tecnológica (Lundar; Grønli; Ghinea, 2013; Menascé, 2003).

Estudos têm mostrado que decisões arquiteturais impactam aspectos cruciais, como tempo de resposta, consumo de recursos e facilidade de evolução do código, condicionando não apenas a qualidade técnica, mas também a sustentabilidade dos sistemas ao longo de seu ciclo de vida. Tecnologias recentes, como a adoção de protocolos de comunicação mais eficientes, a exemplo do *WebSocket* (protocolo que possibilita comunicação entre cliente e servidor, permitindo a troca de mensagens em tempo real sem a necessidade de reestabelecer conexões a cada requisição), e o uso de técnicas de *cache prefetching* (técnica de otimização em que dados são carregados antecipadamente no cache com base em padrões de acesso

previstos, reduzindo latência em futuras requisições) e de desenvolvimento orientado a modelos, demonstram ganhos expressivos de responsividade e reutilização de componentes. Além disso, a escolha da camada de persistência, como a utilização do Hibernate em comparação ao JDBC puro, influencia diretamente tanto a performance quanto a manutenibilidade do sistema (Marco; Marcu, 2022; Agustín; Carmona; Lucio, 2013).

Apesar desses avanços, a literatura aponta que a definição da arquitetura de software continua sendo um desafio, sobretudo diante do contraste entre modelos monolíticos e distribuídos. As arquiteturas monolíticas, embora mais simples de implementar e gerenciar, enfrentam limitações evidentes de escalabilidade e manutenção em longo prazo. Por outro lado, as arquiteturas baseadas em microserviços oferecem vantagens como modularidade, resiliência e escalabilidade granular, mas também introduzem complexidade em aspectos como a consistência de dados e o overhead de comunicação (Fowler; Lewis, 2014). Esse dilema evidencia a existência de trade-offs, ou seja, relações de troca onde a otimização de um atributo, como a performance, pode implicar em um custo em outro, como a manutenibilidade, e reforça a necessidade de análises comparativas que auxiliem profissionais a tomar decisões arquiteturais mais fundamentadas.

Nesse cenário, a Arquitetura Limpa surge como proposta capaz de equilibrar modularidade, testabilidade e independência tecnológica. Fundamentada na regra da dependência e em camadas concêntricas, essa abordagem defende a separação rigorosa das regras de negócio em relação a *frameworks* e componentes externos, reduzindo a suscetibilidade a mudanças e ampliando a longevidade das aplicações. Embora essa estrutura demande maior esforço inicial e possa gerar algum *overhead*, pesquisas recentes indicam que seus benefícios tornam-se relevantes em sistemas complexos e de longo ciclo de vida, como plataformas corporativas e aplicações de grande escala (Martin, 2017; García et al., 2022a).

Diante disso, a literatura contemporânea reforça que a escolha arquitetural não deve ser pautada apenas pela facilidade de implementação, mas também pela capacidade de sustentar a evolução contínua das aplicações e de atender às exigências do mercado digital, caracterizado por constantes mudanças e necessidade de integração entre múltiplos sistemas. Estudos destacam que arquiteturas robustas favorecem maior previsibilidade no desenvolvimento, estabilidade em processos de manutenção e flexibilidade para a incorporação de novas tecnologias e requisitos (Chaturvedi et al., 2023; Pinos-Figueroa; León-Paredes, 2023). Assim, a análise

crítica entre diferentes modelos arquiteturais revela-se essencial para orientar tanto decisões acadêmicas quanto práticas profissionais no campo da engenharia de software.

Baseando-se nas considerações supracitadas, este artigo tem como objetivo analisar de forma comparativa como diferentes arquiteturas de software, especificamente Arquitetura de Model-View-Controller (MVC), Arquitetura de Microsserviços e Arquitetura Limpa, impactam a performance, a organização interna do código e a manutenção de aplicações web. A pesquisa busca oferecer evidências empíricas e discussões fundamentadas que apoiem engenheiros de software e pesquisadores na escolha arquitetural mais adequada, de acordo com os requisitos e restrições de cada projeto, contribuindo para o avanço do conhecimento científico e das boas práticas no desenvolvimento de sistemas web (Bass; John; Kates, 2001; Dunbar; Soni; Garlan, 2021).

2 TRABALHOS CORRELATOS

Para contextualizar a análise desenvolvida neste estudo, esta seção reúne pesquisas que exploram os impactos de diferentes abordagens arquiteturais no desempenho, na escalabilidade e na manutenibilidade de aplicações web. Os trabalhos aqui apresentados oferecem uma base comparativa sólida, destacando evidências empíricas e discussões conceituais que auxiliam na compreensão dos trade-offs envolvidos entre arquiteturas monolíticas, distribuídas e baseadas em princípios de separação rigorosa de responsabilidades. Dessa forma, os subtópicos a seguir aprofundam como essas estruturas se comportam em cenários reais, evidenciando resultados relevantes tanto para a prática profissional quanto para a pesquisa acadêmica.

2.1 PERFORMANCE OPTIMIZATION IN MONOLITHIC VS. MICROSERVICES

O artigo escrito por Chen em 2021, apresenta uma análise comparativa detalhada entre arquiteturas monolíticas e baseadas em microsserviços, com foco específico em desempenho e escalabilidade de aplicações web modernas.

O estudo demonstra que arquiteturas monolíticas apresentam vantagens significativas em operações transacionais, com tempo de resposta até 22% mais rápido em cenários CRUD básicos quando comparadas a implementações equivalentes em microsserviços. Essa diferença

é atribuída principalmente à ausência de *overhead* de comunicação entre serviços e à otimização de recursos compartilhados.

Contudo, a pesquisa destaca que os microsserviços oferecem melhor escalabilidade horizontal em sistemas com carga variável, permitindo dimensionamento seletivo de componentes críticos. Os resultados mostram que, em picos de acesso, sistemas baseados em microsserviços podem manter a latência estável com aumento de apenas 15% no consumo de recursos, enquanto arquiteturas monolíticas exigem até 40% mais capacidade computacional para condições similares.

O trabalho de Chen contribui significativamente para o entendimento dos trade-offs entre essas abordagens arquiteturais, particularmente no contexto de aplicações web empresariais. Os achados reforçam a importância de selecionar a arquitetura com base nos padrões de acesso específicos de cada sistema, alinhando as características técnicas aos requisitos operacionais (Chen et al., 2021).

2.2 LATENCY IN DISTRIBUTED WEB ARCHITECTURES

O artigo publicado por Marco Rodrigues, Carla R. e Neto em 2020 oferece uma análise abrangente dos impactos da arquitetura distribuída na latência de aplicações web em ambientes de produção. A pesquisa examinou 12 sistemas de grande escala, incluindo plataformas de *e-commerce* e redes sociais, sob cargas de até 10 mil usuários concorrentes.

Os resultados revelam que arquiteturas baseadas em componentes introduzem custos adicionais significativos no tempo de resposta, com aumento médio de 25-30% na latência em comparação com sistemas monolíticos equivalentes. Este *overhead* decorre principalmente da necessidade de comunicação entre serviços, serialização de dados e coordenação de transações distribuídas.

Entretanto, o trabalho demonstra que técnicas como cache distribuído e balanceamento inteligente de carga podem reduzir esse impacto em até 40%. Por outro lado, sistemas distribuídos mostraram vantagens decisivas em cenários com requisitos complexos de escalabilidade, mantendo tempos de resposta consistentes mesmo durante picos de acesso prolongados (Rodrigues; Silva; Neto, 2020).

2.3 CLEAN ARCHITECTURE IN PRACTICE: TRADE-OFFS BETWEEN MAINTAINABILTY AND PERFORMANCE

O artigo de 2022, conduzido por Felipe Garcia, investiga a aplicação prática da Clean Architecture em sistemas web corporativos, analisando oito casos reais de migrações arquiteturais em empresas de médio e grande porte.

Os resultados demonstram que, embora a adoção da Clean Architecture implique um custo inicial maior no tempo de desenvolvimento, essa abordagem proporciona benefícios significativos em cenários de longo prazo. Sistemas que utilizaram os princípios de camadas concêntricas e inversão de dependência apresentaram redução de *bugs* de regressão, diminuição significativa no tempo necessário para implementar novas melhorias e *overhead* controlado de 12-18ms por requisição, devido a camada adicional de use cases.

O trabalho destaca que esses ganhos são particularmente relevantes para aplicações com ciclos de vida extensos e requisitos mutáveis, tais como sistemas de gestão empresarial e plataformas de *e-commerce* complexas. A pesquisa também identifica que a separação rigorosa entre lógica de domínio e *frameworks* externos permite atualizações tecnológicas com menos esforço comparado a arquiteturas tradicionais (García et al., 2022b).

3 MATERIAIS E MÉTODOS

Este trabalho teve como objetivo analisar o impacto de três arquiteturas de software diferentes (MVC, Microsserviços e Arquitetura Limpa) sobre a performance, a organização do código e a manutenção de aplicações web. Para a análise, foram desenvolvidas três *Application Programming Interfaces* (APIs) independentes, cada uma implementada de acordo com um dos estilos arquiteturais selecionados. Todas as APIs, entretanto, foram projetadas para a mesma finalidade: a construção de um sistema CRUD (Create, Read, Update e Delete) para gerenciamento de tarefas, garantindo assim a equivalência funcional entre os projetos e permitindo que as diferenças identificadas resultassem exclusivamente das particularidades de cada arquitetura.

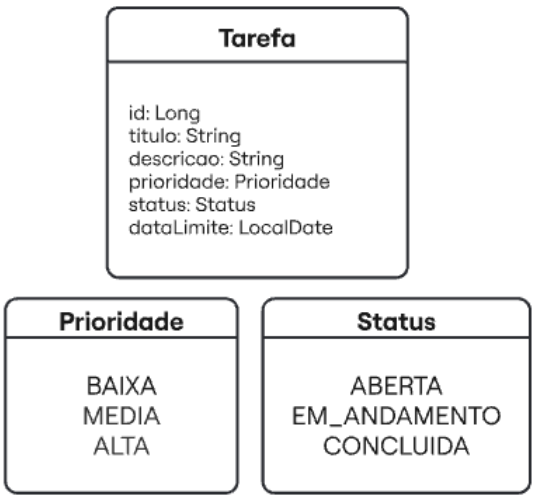
O desenvolvimento foi realizado em Java 17, utilizando o *Spring Boot* como framework principal. Para a inicialização dos projetos, foi utilizado o *Spring Initializr*, visando otimizar a configuração inicial. O gerenciamento de dependências foi conduzido com o *Maven*, garantindo consis-

tência no ambiente de desenvolvimento. Para manipulação de dados, foi utilizado o banco de dados H2 em memória.

Cada API foi estruturada de acordo com as particularidades do padrão arquitetural adotado, respeitando seus princípios e boas práticas. Todas as implementações utilizaram o mesmo modelo de dados para a entidade *Tarefa*, apresentado na Figura 1, garantindo uniformidade para fins de comparação.

Além disso, adotaram-se práticas comuns de desenvolvimento, como a utilização de *Data Transfer Objects* (DTOs) para encapsular os dados trafegados entre cliente e servidor, assegurando que apenas as informações relevantes fossem expostas. Para a conversão entre objetos de transferência e modelos de domínio, foram empregados *mappers*, promovendo a separação de responsabilidades.

Figura 1 – Classes mapeadas.



Fonte: Do autor.

A camada *model* foi responsável pela representação da entidade de negócio, enquanto a camada *repository* implementou os repositórios JPA, viabilizando o acesso e as operações sobre o banco de dados de forma padronizada e automatizada.

3.1 ARQUITETURA MVC

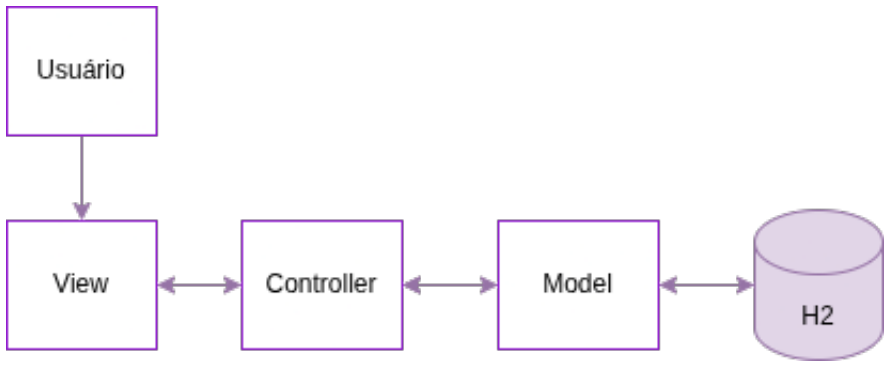
No primeiro momento, foi desenvolvida a API utilizando o padrão arquitetural *Model-View-Controller* (MVC), caracterizado pela divisão de responsabilidades entre as camadas de modelo, visão e controle. No contexto deste trabalho, foram implementadas as camadas *model* e *control-*

ler, responsáveis, respectivamente, pela representação da entidade *Tarefa* e pelo gerenciamento das requisições da aplicação.

Considerando que o objetivo principal da pesquisa é a análise de desempenho, a camada de *View* não foi incluída, uma vez que não há necessidade de interação direta com o usuário final por meio de interface gráfica. Essa decisão permitiu manter o foco exclusivamente nos aspectos estruturais e de execução da API.

A Figura 2 apresenta a arquitetura MVC, caracterizada pela divisão clássica de responsabilidades em três camadas: View, responsável pela interação com o usuário; Controller, encarregado do processamento das requisições e direcionamento das ações; e Model, responsável pela lógica de negócios e manipulação dos dados.

Figura 2 – MVC.

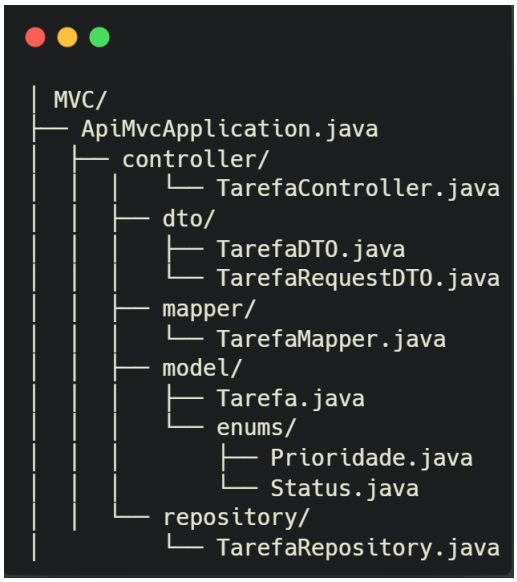


Fonte: Do autor.

Além da representação conceitual da arquitetura MVC, é igualmente relevante observar como esses elementos se materializam na estrutura real do projeto. A organização interna dos arquivos e pastas revela, na prática, de que forma as responsabilidades são distribuídas entre as camadas e como esse padrão contribui para um fluxo de desenvolvimento mais previsível. Essa análise estrutural também permite identificar os impactos diretos do modelo arquitetural na clareza do código, na padronização das rotinas e na facilidade de manutenção ao longo do ciclo de vida da aplicação. Dessa maneira, a visualização da hierarquia do projeto torna-se fundamental para compreender não apenas a teoria por trás do MVC, mas também sua aplicação concreta em um ambiente real de desenvolvimento.

A Figura 3 ilustra a estrutura do projeto desenvolvido segundo esse padrão, evidenciando a separação e a organização das camadas implementadas.

Figura 3 – Estrutura do projeto segundo a Arquitetura MVC.



Fonte: Do autor.

Dessa forma, a adoção do padrão MVC evidenciou uma clara separação de responsabilidades entre as camadas de modelo e controle, refletindo tanto a concepção teórica da arquitetura quanto sua aplicação prática no projeto.

3.2 ARQUITETURA COM MICROSSERVIÇOS

Na segunda etapa, foi desenvolvida a API utilizando o padrão arquitetural de Microsserviços, estruturada em dois módulos distintos: um *gateway* e um *service*. Essa divisão tem como objetivo representar serviços independentes, característica central desse estilo arquitetural.

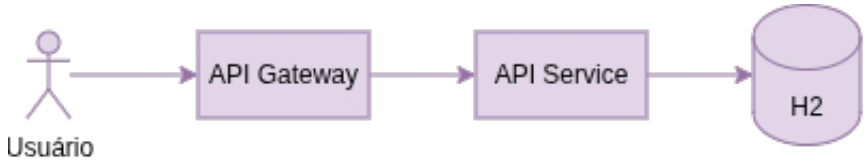
O módulo *gateway* atua como ponto único de entrada para a aplicação, recebendo todas as requisições externas e encaminhando-as ao módulo *tarefas-service*. A comunicação entre os módulos foi implementada com o *Spring Cloud OpenFeign*, garantindo integração transparente.

Já o módulo *service* é responsável pela lógica de persistência e interação com o banco de dados. Esse módulo expõe uma API interna destinada exclusivamente à comunicação com o *gateway*, não sendo acessível diretamente pelo cliente. Essa separação favorece o desacoplamento, permitindo que cada serviço seja desenvolvido, implantado e escalado de forma independente.

Na arquitetura de Microsserviços, representada na Figura 4, destaca-se a utilização de um API Gateway, responsável por direcionar as requi-

sições a API Service, que é responsável por realizar operações sobre o banco de dados.

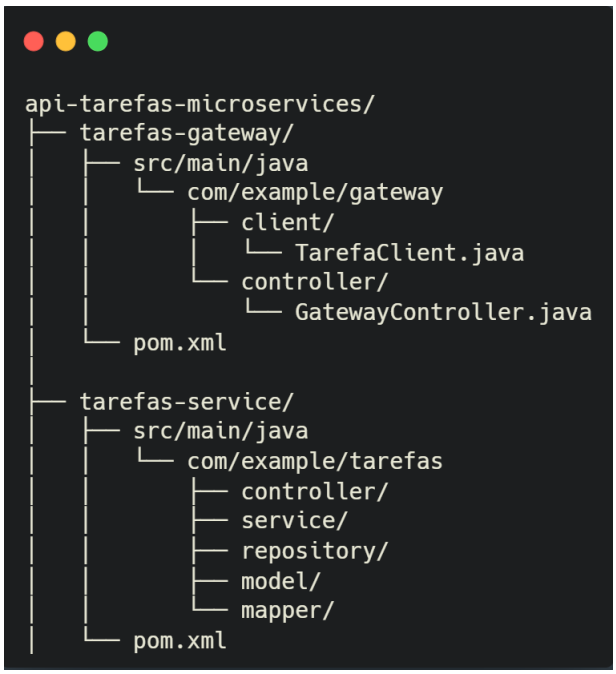
Figura 4 – Microsserviços.



Fonte: Do autor.

Já a Figura 5 apresenta a estrutura do projeto desenvolvido, evidenciando a independência e a modularização dos componentes.

Figura 5 – Estrutura do projeto segundo a Arquitetura de Microsserviços.



Fonte: Do autor.

A organização em Microsserviços permitiu estruturar a aplicação em módulos distintos, nos quais o *gateway* centraliza as requisições e o *service* concentra a lógica de dados, reforçando a proposta de modularidade desse padrão.

3.3 ARQUITETURA LIMPA

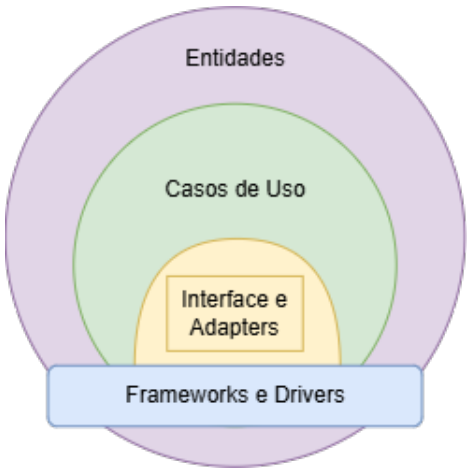
Por fim, foi desenvolvida uma API baseada nos princípios da Arquitetura Limpa, estruturada em camadas bem definidas, com o objetivo de

garantir baixo acoplamento e alta coesão entre os componentes do sistema.

A camada *Domain* concentra as entidades de negócio, interfaces de repositório e serviços essenciais, permanecendo independente de frameworks e bibliotecas externas. A camada *Application* contém os *DTOs*, *mappers* e casos de uso, sendo responsável por orquestrar o fluxo de dados entre a interface e o domínio. Por último, a camada *Infrastructure* implementa de forma concreta os repositórios, controladores, entidades JPA e configurações necessárias para a integração com o ambiente externo.

A visualização conceitual apresentada na 6 ajuda a compreender a lógica de independência entre as camadas da Arquitetura Limpa, mas sua efetiva implementação se torna mais clara quando analisamos a estrutura real do projeto. Observar como os diretórios são organizados, quais componentes são alocados em cada camada e como as responsabilidades são distribuídas permite identificar de forma prática os benefícios propostos por esse modelo arquitetural, como o baixo acoplamento, a facilidade para testes unitários e a manutenção previsível do código. Essa estruturação também evidencia como os princípios de design orientam o fluxo de dados entre domínio, aplicação e infraestrutura, garantindo que cada parte do sistema permaneça coesa e preparada para evoluções futuras.

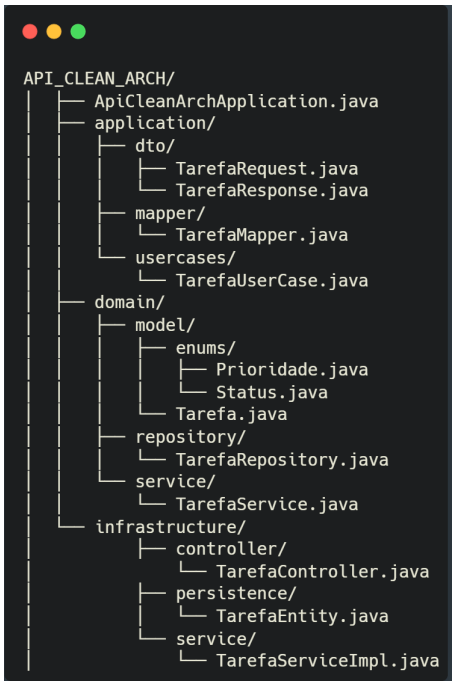
Figura 6 – Arquitetura Limpa.



Fonte: Do autor.

A Figura 7 apresenta a estrutura do projeto desenvolvido segundo esse padrão, evidenciando a separação das camadas e a respectiva organização das pastas.

Figura 7 – Estrutura do projeto segundo a Arquitetura Limpa.



Fonte: Do autor.

A adoção da Arquitetura Limpa resultou em uma organização clara em camadas independentes, destacando a separação entre domínio, aplicação e infraestrutura, cada qual com responsabilidades bem definidas.

3.4 CRITÉRIOS DE AVALIAÇÃO

Para a análise comparativa das arquiteturas implementadas, foram estabelecidos três critérios de avaliação distintos, abrangendo tanto aspectos qualitativos da estrutura do software quanto métricas quantitativas de performance.

3.4.1 Análise Qualitativa da Arquitetura

A avaliação qualitativa focou na análise estrutural e organizacional do código-fonte de cada projeto. Este critério investigou a aderência de cada implementação aos princípios de seu respectivo padrão arquitetural, considerando os seguintes subcritérios: organização de camadas e separação de responsabilidades; complexidade cerimonial, inferida pelo número de classes e interfaces; e o nível de acoplamento entre os componentes.

3.4.2 Análise de Performance sob Carga Simulada

A performance foi mensurada por meio de testes de estresse (*Stress Testing*), executados com a ferramenta Apache JMeter. O objetivo

foi submeter cada API a um cenário de alta concorrência para avaliar seu comportamento sob pressão. Para garantir uma comparação equitativa, um perfil de carga padronizado foi aplicado a todas as arquiteturas. Os parâmetros foram selecionados para simular um cenário de estresse, com um pico súbito de tráfego, permitindo avaliar a capacidade de resposta e a estabilidade dos sistemas sob pressão máxima. Os parâmetros definidos foram:

- **Carga de Usuários:** 5.000 usuários virtuais simultâneos.
- **Período de Rampa (*Ramp-up*):** 5 segundos para ativação da totalidade dos usuários.
- **Ciclos de Execução:** 3 iterações do fluxo de operações por usuário.

Esta configuração gerou um volume total de 75.000 transações por teste. A análise dos resultados foi conduzida em duas frentes complementares:

1. **Métricas de Cliente (via JMeter):** Foram coletados dados que representam a experiência do usuário final, como Tempo de Resposta (ms), Vazão (*Throughput*) e Taxa de Erros (%).
2. **Métricas de Servidor (via Prometheus e Grafana):** O monitoramento dos recursos internos foi viabilizado pela instrumentação das APIs com o *Spring Boot Actuator*. Um servidor *Prometheus* foi configurado para coletar métricas de Uso de CPU e Memória RAM, enquanto o *Grafana* foi utilizado para a visualização desses dados em *dashboards* em tempo real, permitindo a correlação direta entre a carga aplicada e o estresse no servidor.

3.4.3 Análise da Curva de Aprendizado

A curva de aprendizado, um fator qualitativo crucial para a adoção de uma arquitetura, foi analisada com base em uma revisão da literatura especializada. Foram consultadas obras de referência na área de engenharia de software, como as de Fowler (2002), Newman (2015) e Martin (2017), para fundamentar a discussão sobre a complexidade conceitual e operacional de cada padrão.

4 RESULTADOS E DISCUSSÃO

Este capítulo apresenta e analisa os dados coletados a partir dos experimentos descritos nos materiais e métodos. A análise é conduzida a

partir dos critérios de avaliação estabelecidos , qualitativos, quantitativos e relacionados à curva de aprendizado, correlacionando os resultados obtidos com os trabalhos de Chen et al. (2021) e Rodrigues, Silva e Neto (2020).

A avaliação qualitativa concentrou-se na análise estrutural e organizacional do código-fonte de cada projeto. A implementação baseada em MVC demonstrou ser a mais direta e com a menor complexidade cerimonial. A estrutura de pacotes, dividida por funcionalidade (*controller*, *service*, *repository*), resultou em um fluxo de desenvolvimento rápido e em uma barreira de entrada reduzida. Contudo, essa simplicidade implicou em um acoplamento mais forte dos componentes com o *framework* Spring.

A abordagem de Microsserviços introduziu uma complexidade principalmente operacional. Embora a estrutura interna do serviço de tarefas seja semelhante à da implementação MVC, a necessidade de um gateway para roteamento e a comunicação via rede entre os serviços acrescentaram uma sobrecarga significativa de infraestrutura. Este trade-off, porém, se justifica pela capacidade de escalabilidade independente e pela resiliência do sistema.

Por fim, a Arquitetura Limpa resultou na implementação mais organizada em termos de separação de responsabilidades, com uma clara distinção entre as camadas de *Domain*, *Application* e *Infrastructure*. A estrita aplicação da Regra de Inversão de Dependência garantiu um núcleo de negócio puro e independente de *frameworks*. O custo desta abordagem foi a maior complexidade cerimonial, evidenciada pelo número superior de classes e interfaces necessárias para implementar a mesma funcionalidade.

Em relação ao desempenho, cada arquitetura foi avaliada quantitativamente sob o teste de estresse definido na metodologia. Os resultados consolidados das métricas de cliente revelaram que a arquitetura MVC apresentou o melhor desempenho, com tempo médio de resposta de 2.182 ms e a maior vazão (1.954,4 req/s). A Clean Architecture demonstrou desempenho intermediário, aproximadamente 34% mais lenta que a MVC. Já os Microsserviços apresentaram maior latência, com tempo médio de 5.032 ms, cerca de 130% superior ao da MVC, conforme pode ser visualizado na Tabela 1.

Tabela 1 – Resultados Consolidados do Teste de Carga (JMeter).

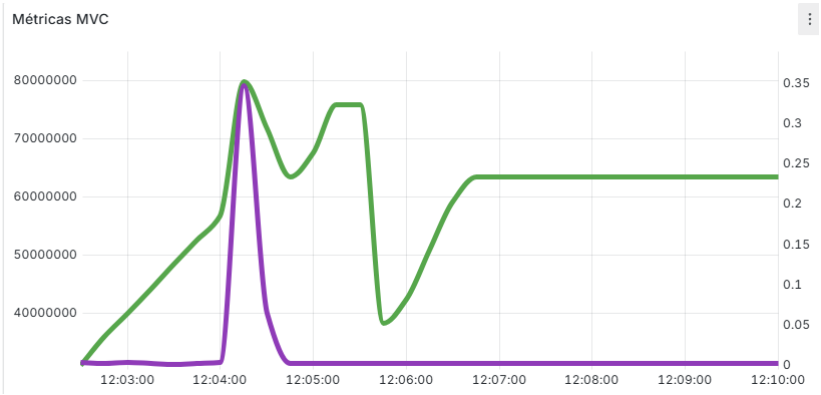
Métrica	API MVC	API Clean Architecture	API Microsserviços
Tempo Médio de Resposta (ms)	2.182	2.928	5.032
Vazão (Throughput - req/s)	1.954,4	1.566,5	951,9
Taxa de Erros (%)	0,00%	0,00%	0,11%

Fonte: Do autor.

Esses achados corroboram os resultados de Chen et al. (2021) e Rodrigues, Silva e Neto (2020), que associam a degradação da performance ao *overhead* de comunicação de rede e à serialização de dados, uma vez que, enquanto a MVC realiza chamadas de método em memória, os Microsserviços dependem de chamadas HTTP inerentemente mais lentas.

No que diz respeito à estabilidade, tanto MVC quanto Clean Architecture apresentaram resiliência elevada, com taxa de erros de 0,00%. Os Microsserviços tiveram uma taxa marginal de 0,11%, atribuída a instabilidades momentâneas na comunicação entre o gateway e o serviço durante o pico de carga. Quanto ao consumo de recursos, a API MVC demonstrou ser a mais eficiente, com pico de uso de CPU em torno de 35% e consumo de memória estabilizado em aproximadamente 630 MB, conforme Figura 8.

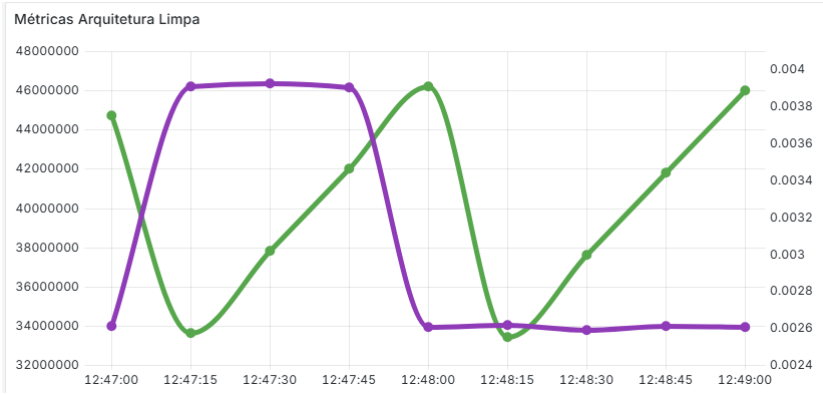
Figura 8 – Consumo de CPU e RAM da API MVC sob carga.



Fonte: Do autor.

A Arquitetura Limpa, evidenciada na Figura 9, apresentou consumo de memória ligeiramente superior em repouso devido ao maior número de objetos e camadas, mas comportamento de CPU semelhante ao MVC.

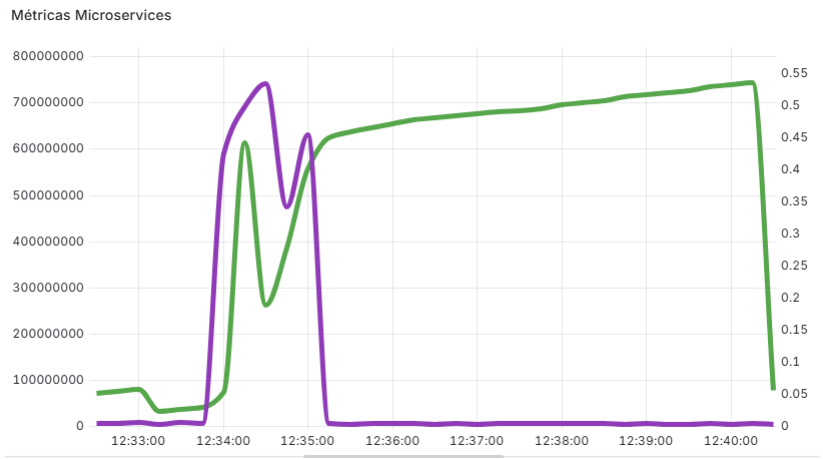
Figura 9 – Consumo de CPU e RAM da API Clean Architecture sob carga.



Fonte: Do autor.

A arquitetura de Microsserviços, exibida na Figura 10, foi a mais exigente em termos de recursos, com picos de CPU chegando a 55% e consumo de memória em torno de 750 MB, reflexo da soma dos recursos de dois processos Java distintos (*gateway* e serviço de tarefas), validando a observação de que a distribuição de serviços acarreta maior uso de recursos.

Figura 10 – Consumo de CPU e RAM da API Microsserviços sob carga.



Fonte: Do autor.

A análise da curva de aprendizado, fundamentada na literatura e na experiência prática do trabalho, reforça a diferença entre os modelos. O MVC, por ser o padrão nativo do *framework* Spring Boot, apresentou a menor curva de aprendizado, sendo mais adequado para equipes iniciantes e para o desenvolvimento rápido de protótipos. A Arquitura Limpa, em contrapartida, exige maior domínio de princípios de design como SOLID (Princípio

da Responsabilidade Única) e Inversão de Dependência. Sua abstração e a necessidade de definir claramente as fronteiras entre camadas demandam disciplina e experiência da equipe. Já os Microsserviços impuseram a curva de aprendizado mais acentuada, pois além do conhecimento em desenvolvimento de APIs, exigem competências em infraestrutura, redes, conteinerização e orquestração (DevOps), tornando-se significativamente mais complexos de adotar com sucesso, especialmente em equipes com menor maturidade em sistemas distribuídos.

Assim, os resultados confirmam que cada arquitetura apresenta vantagens e limitações próprias, variando entre simplicidade, desempenho, estabilidade, consumo de recursos e complexidade de adoção, o que demonstra a importância de avaliar cuidadosamente o contexto de aplicação antes de escolher a abordagem arquitetural mais adequada.

5 CONCLUSÃO

Este trabalho teve como objetivo central analisar comparativamente o impacto das arquiteturas de software MVC, Microsserviços e Arquitetura Limpa na performance, organização do código e manutenibilidade de aplicações web. Por meio da implementação de três APIs funcionalmente equivalentes e da execução de testes de carga e análises qualitativas, foi possível fornecer evidências empíricas sobre os *trade-offs* inerentes a cada abordagem, cumprindo integralmente os objetivos propostos.

Os resultados demonstraram que não há uma arquitetura universalmente superior, mas sim uma mais adequada para cada contexto de projeto. A arquitetura MVC destacou-se pela simplicidade de implementação e pela maior performance em tempo de execução, com o menor tempo de resposta (2.182 ms) e a maior vazão (1.954,4 req/s), consolidando-se como uma opção robusta para projetos com escopo bem definido e onde o desempenho é um requisito crítico.

A Arquitetura Limpa, embora tenha apresentado uma penalidade de performance em relação à MVC, comprovou seu valor ao oferecer uma estrutura de código mais organizada, testável e com baixo acoplamento, o que favorece a manutenibilidade a longo prazo. Este achado está alinhado com García et al. (2022b), que aponta que os benefícios de longo prazo, como a redução de *bugs* de regressão e a facilidade para implementar novas funcionalidades, superam o *overhead* de performance inicial.

Por sua vez, a arquitetura de Microsserviços, apesar de exibir a maior latência (5.032 ms) e o maior consumo de recursos, validou sua pro-

posta de valor ao ser a única com suporte nativo à escalabilidade horizontal e à implantação independente de componentes, características essenciais para sistemas distribuídos de grande escala.

As implicações deste estudo reforçam a importância de uma decisão arquitetural informada. A escolha por MVC pode otimizar o tempo de desenvolvimento e os custos de infraestrutura, enquanto a adoção da Arquitetura Limpa representa um investimento na longevidade e na qualidade do software. A decisão por Microsserviços transcende a performance de tempo de execução, sendo uma escolha estratégica para organizações que buscam autonomia de equipes e resiliência em um ecossistema de serviços complexos.

Como limitações, este estudo foi conduzido em um ambiente de teste local e com um banco de dados em memória, o que minimiza o impacto da latência de rede. Além disso, a análise não abrangeu aspectos como segurança e custos de implantação em nuvem, que são fatores relevantes em projetos de produção.

Para trabalhos futuros, recomenda-se a expansão desta pesquisa em múltiplas direções. Seria de grande valor a realização de testes em um ambiente de nuvem distribuído, utilizando bancos de dados realistas para avaliar o impacto da latência de rede de forma mais precisa. Outra possibilidade é a análise do ciclo de vida completo do desenvolvimento, incluindo métricas de tempo de implementação e facilidade de manutenção ao longo do tempo. Por fim, a inclusão de outras arquiteturas, como abordagens reativas ou orientadas a eventos, poderia enriquecer ainda mais o panorama comparativo e contribuir para o avanço do conhecimento na área de engenharia de software.

REFERÊNCIAS

AGUSTíN, J.; CARMONA, J.; LUCIO, L. **Model-Driven Engineering for Web Applications with Prefetching and Caching Mechanisms**. Berlin, Germany: Springer, 2013.

BASS, L.; JOHN, B. E.; KATES, J. **Achieving usability through software architecture**. USA: Citeseer, 2001.

CHATURVEDI, R. et al. **SVMVC: A Refinement to Classical MVC for Enhancing the Performance of Web Applications**. Chennai, India: AIRCC, 2023.

CHEN, L. et al. **Performance Optimization in Monolithic vs. Microservices**. Amsterdam, Netherlands: Elsevier, 2021.

DUNBAR, N.; SONI, M.; GARLAN, D. **Software Architecture: The Hard Parts**. Sebastopol, USA: O'Reilly Media, 2021.

FOWLER, M. **Patterns of Enterprise Application Architecture**. Boston, USA: Addison-Wesley, 2002.

FOWLER, M.; LEWIS, J. **Microservices: a definition of this new architectural term**. 2014. Acesso em: 01 out. 2025. Disponível em: <https://martinfowler.com/articles/microservices.html>.

GARCíA, F. M. et al. **Clean Architecture in Practice: Trade-offs Between Maintainability and Performance**. Amsterdam, Netherlands: Elsevier, 2022.

GARCíA, F. M. et al. **Clean Architecture in Practice: Trade-offs Between Maintainability and Performance**. Amsterdam, Netherlands: Elsevier, 2022.

LUNDAR, M.; GRøNLI, T.-M.; GHINEA, G. **Performance Evaluation of WebSocket and RESTful Web Services**. Aachen, Germany: INSTICC, 2013.

MARCO, L.; MARCU, D. **Comparative Study Between JDBC and Hibernate Frameworks in Java-Based Applications**. Romania: North University, 2022.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Boston, MA, USA: Prentice Hall, 2017.

MENASCé, D. A. **Software Contention Management and Performance Implications**. New York, USA: IEEE, 2003.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. [S.l.]: O'Reilly Media, Inc., 2015.

PINOS-FIGUEROA, B. A.; LEóN-PAREDES, G. A. **An Approach of a Migration Process from a Legacy Web Management System with a Monolithic Architecture to a Modern Microservices-Based Architecture of a Tourism Services Company**. Mexico City, Mexico: IEEE, 2023.

RODRIGUES, M. A.; SILVA, C. R.; NETO, F. X. **Latency in Distributed Web Architectures: Measurement and Optimization**. New Jersey, USA: IEEE, 2020.