INSTITUTO FEDERAL DE CIÊNCIA E TECNOLOGIA DE PERNAMBUCO

Campus Garanhuns

Análise e Desenvolvimento de Sistemas

MARCOS VINICIUS DE MELO SOUTO

ARQUITETURA ESCALÁVEL PARA SISTEMAS DE DELIVERY: Uma Abordagem Baseada em Eventos, Cache Distribuído e Indexação Geoespacial

Garanhuns

2026

MARCOS VINICIUS DE MELO SOUTO

ARQUITETURA ESCALÁVEL PARA SISTEMAS DE DELIVERY: Uma Abordagem Baseada em Eventos, Cache Distribuído e Indexação Geoespacial

Trabalho de conclusão de curso em Análise e Desenvolvimento de Sistemas do Instituto Federal de Ciência e Tecnologia de Pernambuco, como requisito para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Dr. Rafael Galvão de Mesquita

Garanhuns

2026

S728a Souto, Marcos Vinicius de Melo

Arquitetura escalável para sistemas de delivery: uma abordagem baseada em eventos, cache distribuído e indexação geoespacial / Marcos Vinicius de Melo Souto ; orientador Rafael Galvão de Mesquita, 2026.

72f. : il.

Orientador: Rafael Galvão de Mesquita.

Trabalho de Conclusão de Curso (Graduação) – Instituto Federal de Pernambuco. Pró-Reitoria de Ensino. Diretoria de Ensino. Campus Garanhuns. Coordenação do Curso de Tecnólogo em Análise e Desenvolvimento de Sistemas. Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, 2026.

1. Software - Desenvolvimento. 2. Sistemas operacionais distribuídos (computadores). 3. Arquitetura de software. I. Título. II. Mesquita, Rafael Galvão de (orientador). III. Instituto Federal de Pernambuco.

CDD 005.1

Louise Machado Freire Dias – CRB4/2267

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DE PERNAMBUCO CAMPUS GARANHUNS
COORDENAÇÃO DO CURSO SUPERIOR DE TECNOLOGIA EM
ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

MARCOS VINICIUS DE MELO SOUTO

ARQUITETURA ESCALÁVEL PARA SISTEMAS DE DELIVERY: Uma Abordagem Baseada em Eventos, Cache Distribuído e Indexação Geoespacial

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, IFPE-Campus Garanhuns, como parte das exigências para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Garanhuns-PE, 20 de julho de 2026.

BANCA EXAMINADORA

_____
(Nome do Orientador)
Filiação

_____
(Nome do Avaliador)
Filiação

_____
(Nome do Avaliador)
Filiação

Dedico este trabalho à minha avó, Maria José de Barros Souto, pelo carinho, pelos ensinamentos e pelo apoio ao longo da minha vida. Sua dedicação à família e seus valores sempre foram fonte de inspiração para mim.

AGRADECIMENTOS

Agradeço aos meus pais, Valdir e Jasilda, por todo o amor, apoio e incentivo ao longo da minha vida sem vocês, esta conquista não seria possível. Agradeço também às minhas irmãs, Eduarda e Júlia, pelo carinho, pelo companheirismo e por estarem presentes em todos os momentos desta jornada.

Ao meu orientador, Prof. Rafael, agradeço pela orientação, pela dedicação e pelos conhecimentos compartilhados durante a realização deste trabalho. Estendo esse agradecimento aos demais professores do IFPE, que contribuíram para minha formação acadêmica e profissional ao longo do curso.

Agradeço, ainda, aos meus colegas de trabalho Carol, Gian, Paula, Paulena e tantos outros pelo apoio, pelo aprendizado e pela troca de experiências ao longo desta caminhada. Registro também um agradecimento especial à Ariana Grande, cuja música esteve presente em diferentes momentos da minha vida, servindo como companhia, inspiração e motivação durante esta trajetória.

Por fim, agradeço a todos que, direta ou indiretamente, contribuíram para a realização deste sonho. Meu sincero obrigado.

"*Any fool can write code that a computer can understand. Good programmers write code that humans can understand*"

(Martin Flower)

RESUMO

Este trabalho apresenta o desenvolvimento de uma plataforma backend para um sistema de marketplace de delivery de alimentos, projetada para gerenciar de forma eficiente todo o ciclo de vida dos pedidos, desde a seleção de produtos até a entrega e avaliação do serviço. A solução proposta utiliza uma arquitetura de monólito modular com forte adoção dos princípios da arquitetura hexagonal, promovendo desacoplamento entre regras de negócio e dependências externas. Além disso, o sistema incorpora conceitos de arquitetura orientada a eventos, permitindo o processamento assíncrono de tarefas críticas por meio de sistema de mensageria, contribuindo para a escalabilidade e resiliência da aplicação. Como suporte à performance, são empregadas tecnologias de cache em memória e estratégias de gerenciamento de estado, reduzindo a latência em operações sensíveis. Destaca-se também o uso de indexação geoespacial baseada em H3 para otimização do processo de despacho de entregadores, possibilitando consultas eficientes de proximidade. A metodologia adotada baseou-se na definição de requisitos funcionais e não funcionais, Design Orientado a Domínio (DDD) e implementação incremental com uso de contêineres para padronização do ambiente. Os resultados demonstram que a solução é capaz de atender a cenários de alta demanda, oferecendo desempenho adequado, organização arquitetural e facilidade de evolução. Conclui-se que a combinação de arquitetura limpa, processamento assíncrono e técnicas de geolocalização constitui uma abordagem eficaz para o desenvolvimento de sistemas modernos de delivery, apresentando-se como uma base sólida tanto para aplicações reais quanto para estudos acadêmicos.

Palavras-chave: Arquitetura de Software. Sistemas Distribuídos. Event-Driven Architecture. Delivery. Escalabilidade.

ABSTRACT

This work presents the development of a backend platform for a food delivery marketplace system, designed to efficiently manage the entire order lifecycle, from product selection to delivery and service evaluation. The proposed solution adopts a modular monolith architecture with strong adherence to Hexagonal Architecture principles, promoting a clear separation between business logic and external dependencies. Furthermore, the system incorporates concepts of Event-Driven Architecture, enabling asynchronous processing of critical tasks through distributed messaging, which enhances scalability and resilience. To support performance, in-memory caching technologies and state management strategies are employed, reducing latency in critical operations. Additionally, the system leverages geospatial indexing based on H3 to optimize the delivery dispatch process, enabling efficient proximity queries. The adopted methodology is based on the definition of functional and non-functional requirements, domain-driven design modeling, and incremental implementation using containerization to ensure environment consistency. The results indicate that the solution is capable of handling high-demand scenarios, providing adequate performance, architectural organization, and ease of evolution. It is concluded that the combination of clean architecture, asynchronous processing, and geolocation techniques constitutes an effective approach for building modern delivery systems, serving as a solid foundation for both real-world applications and academic studies.

Keywords: Software Architecture. Distributed Systems. Event-Driven Architecture. Delivery Systems. Scalability.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

IFPE	Instituto Federal de Pernambuco
ABNT	Associação Brasileira de Normas Técnicas

SUMÁRIO

1 INTRODUÇÃO

O crescimento acelerado das plataformas digitais de entrega de alimentos trouxe desafios significativos relacionados à escalabilidade, confiabilidade e organização de sistemas *backend*. Aplicações desse tipo precisam lidar simultaneamente com múltiplos atores, como clientes, estabelecimentos comerciais, entregadores e administradores, além de garantir processamento eficiente de pedidos, pagamentos e logística de entrega.

1.1 Objetivo Geral

O objetivo geral deste trabalho é projetar, implementar e documentar uma arquitetura *backend* modular e escalável para uma plataforma de *delivery*, aplicando conceitos contemporâneos de engenharia de *software* e sistemas distribuídos, de forma a demonstrar, na prática, como diferentes padrões arquiteturais e tecnologias podem ser combinados para resolver problemas reais de desempenho, organização e confiabilidade.

1.2 Objetivos Específicos

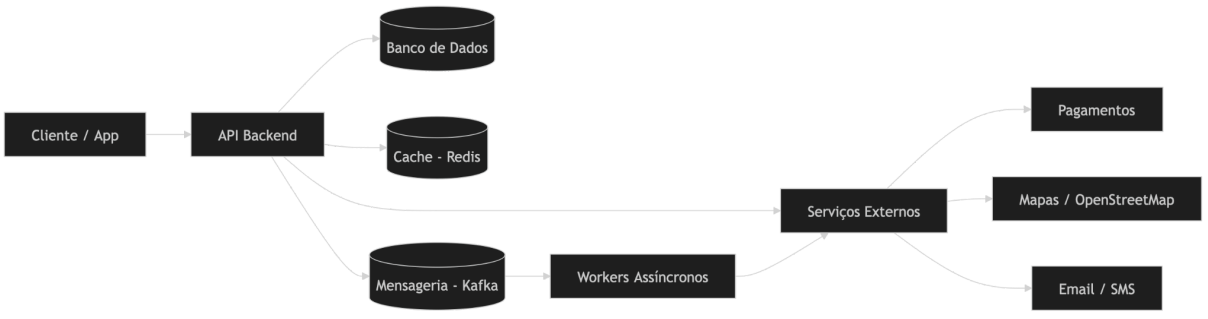
a) Modelar o domínio da aplicação considerando os principais atores e fluxos de um sistema de *delivery.*
b) Definir uma arquitetura de *software* baseada em princípios de desacoplamento e modularização.
c) Implementar *Application Programming Interfaces* (APIs) para comunicação síncrona entre os componentes do sistema.
d) Integrar mecanismos de processamento assíncrono utilizando filas de mensagens e *workers*.
e) Utilizar técnicas de *cache* para otimização de desempenho e redução de latência.
f) Incorporar geolocalização para apoio à lógica de entrega e distribuição.

g) Garantir a separação de responsabilidades por meio de camadas bem definidas.
h) Desenvolver e executar testes de integração para validação das funcionalidades implementadas.
i) Avaliar a solução desenvolvida sob a perspectiva estrutural de escalabilidade e manutenção, com base na organização arquitetural do código e não em testes quantitativos de desempenho sob carga.

A relevância deste estudo está na aplicação prática de conceitos avançados de engenharia de *software*, como arquitetura hexagonal, uso de filas de mensagens, *cache* distribuído e geolocalização para otimização de entregas. Além disso, o sistema foi projetado considerando aspectos reais de mercado, como escalabilidade horizontal, tolerância a falhas e separação de responsabilidades entre diferentes camadas da aplicação.

Dessa forma, este trabalho contribui para a compreensão de como projetar sistemas distribuídos modernos, apresentando uma solução robusta e extensível para o domínio de *delivery*, A Figura 1 apresenta uma visão geral simplificada da arquitetura do sistema, destacando seus principais componentes e interações.

Figura 1 – Visão geral simplificada da arquitetura



Fonte: Elaborado pelo autor (2026).

2 FUNDAMENTAÇÃO TEÓRICA

O desenvolvimento do sistema proposto fundamenta-se em conceitos consolidados da engenharia de *software*, arquitetura de sistemas distribuídos e modelagem de domínio, visando garantir escalabilidade, manutenibilidade e robustez.

2.1 Arquitetura Hexagonal

A arquitetura hexagonal, também conhecida como *Ports and Adapters*, foi utilizada com o objetivo de isolar o núcleo da aplicação de suas dependências externas, promovendo um alto grau de desacoplamento entre a lógica de negócio e os detalhes de infraestrutura. Nesse modelo, o sistema é organizado em torno de um domínio central, que se comunica com o mundo externo por meio de portas (interfaces) e adaptadores (implementações concretas), permitindo que tecnologias como bancos de dados, *frameworks web* ou sistemas de mensageria sejam substituídas sem impacto direto nas regras de negócio.

Esse princípio está alinhado com os conceitos de separação de responsabilidades e independência de camadas discutidos por Len Bass, Paul Clements e Rick Kazman, que destacam a importância de arquiteturas capazes de reduzir o impacto de mudanças e facilitar a manutenção do *software* (BASS; CLEMENTS; KAZMAN, 2012). Da mesma forma, a centralização das regras de negócio no domínio segue os princípios apresentados por Eric Evans, posteriormente aprofundados por Vaughn Vernon, ao enfatizar que a complexidade do *software* deve ser tratada prioritariamente no modelo de domínio (EVANS, 2003; VERNON, 2013). Além disso, a redução do acoplamento estrutural e a evolução independente dos componentes também dialogam com os padrões de arquitetura corporativa apresentados por Martin Fowler (FOWLER, 2002)

Ao considerar alternativas arquitetônicas, destaca-se inicialmente o modelo tradicional em camadas, também conhecido como arquitetura em três camadas. Esse modelo organiza o sistema em três partes principais: apresentação (interface

com o usuário), lógica de negócio (regras da aplicação) e acesso a dados (persistência). Seu principal objetivo é estruturar o código de forma hierárquica e facilitar a manutenção por meio da separação de responsabilidades. No entanto, na prática, é comum que a camada de negócio acabe dependendo diretamente de bibliotecas de persistência ou *frameworks*, o que reduz o desacoplamento e dificulta a substituição de tecnologias ao longo do tempo.

Outra abordagem relevante é a arquitetura de microserviços, cujo objetivo é decompor o sistema em múltiplos serviços independentes, cada um responsável por um domínio específico do negócio. Esses serviços se comunicam entre si por meio de APIs ou mensageria, permitindo escalabilidade independente, *deploy* isolado e maior autonomia entre equipes de desenvolvimento, conforme descrito por Sam Newman e Chris Richardson (NEWMAN, 2021; RICHARDSON, 2018).

No entanto, essa abordagem introduz desafios significativos, como a necessidade de gerenciar comunicação distribuída, tolerância a falhas, consistência de dados e observabilidade, aspectos amplamente discutidos por Martin Kleppmann ao abordar sistemas distribuídos orientados a dados e arquiteturas escaláveis (KLEPPMANN, 2017).

Adicionalmente, pode-se citar o estilo arquitetural REST, definido por Roy Fielding, cujo foco está na padronização da comunicação entre sistemas distribuídos por meio de recursos e operações HTTP. Embora seja amplamente utilizado na construção de APIs, o REST não define como a lógica interna da aplicação deve ser estruturada, sendo frequentemente combinado com outros padrões arquitetônicos.

Diante desse cenário, a adoção da arquitetura hexagonal neste trabalho se justifica por oferecer um equilíbrio entre organização interna, flexibilidade e viabilidade prática de implementação. Diferentemente do modelo em camadas tradicionais, ela garante maior independência do domínio em relação às tecnologias externas, reduzindo o impacto de mudanças futuras. Ao mesmo tempo, em comparação com microserviços, permite manter a simplicidade de um único *deploy*, evitando a sobrecarga operacional associada a sistemas altamente distribuídos. Essa característica é particularmente relevante no contexto deste trabalho, cujo foco

está na construção de uma solução robusta e bem estruturada, sem a necessidade de lidar com toda a complexidade de um ambiente distribuído em produção.

Adicionalmente, a arquitetura hexagonal torna o sistema mais testável, uma vez que as dependências externas podem ser facilmente substituídas por implementações simuladas durante a execução de testes. Essa propriedade é essencial em aplicações que envolvem regras de negócio críticas, como plataformas de *delivery*. Por outro lado, reconhece-se como desvantagem o aumento da quantidade de abstrações e da complexidade estrutural do código, o que exige maior disciplina de desenvolvimento e pode representar uma curva de aprendizado mais elevada para novos desenvolvedores.

2.2 Arquitetura Orientada a Eventos

A arquitetura orientada a eventos é um paradigma no qual os componentes de um sistema se comunicam por meio da produção e consumo de eventos, que representam mudanças de estado ou ocorrências relevantes dentro do domínio da aplicação. Nesse modelo, produtores emitem eventos sem conhecimento direto dos consumidores, que, por sua vez, reagem a esses eventos de forma assíncrona. Essa abordagem favorece o desacoplamento entre os componentes e permite a construção de sistemas mais flexíveis e adaptáveis. De acordo com Kleppmann (2017), sistemas baseados em eventos são especialmente adequados para aplicações distribuídas, pois permitem lidar de forma eficiente com grandes volumes de dados e operações concorrentes. Além disso, Richardson (2018) destaca que a comunicação assíncrona baseada em eventos é um dos pilares para a construção de sistemas modernos e escaláveis.

Ao considerar alternativas arquitetônicas, destaca-se inicialmente o modelo de comunicação síncrona, amplamente utilizado em arquiteturas baseadas em APIs REST. Nesse modelo, um componente realiza uma requisição direta a outro e aguarda uma resposta imediata para continuar sua execução. Esse tipo de comunicação é simples de implementar e facilita o entendimento do fluxo de execução, sendo amplamente difundido em sistemas distribuídos, conforme descrito

por Fielding (2000). No entanto, essa abordagem cria um forte acoplamento temporal entre os serviços, uma vez que a disponibilidade de um componente depende diretamente da resposta de outro, o que pode comprometer a resiliência do sistema.

Outra alternativa relevante é o uso de arquiteturas baseadas exclusivamente em microserviços com comunicação síncrona entre serviços. Nesse cenário, o sistema é dividido em múltiplos serviços independentes, mas a interação entre eles ocorre predominantemente por chamadas diretas, geralmente via HTTP. Embora essa abordagem proporciona modularidade e separação de responsabilidades, ela também pode introduzir problemas como aumento da latência, propagação de falhas e dificuldade de escalabilidade em cenários de alta carga, conforme discutido por Newman (2015) e Kleppmann (2017).

Diante desse contexto, a adoção da arquitetura orientada a eventos neste trabalho se justifica pela necessidade de aumentar o desacoplamento entre os componentes e melhorar a escalabilidade do sistema. Diferentemente da comunicação síncrona, o modelo orientado a eventos permite que diferentes partes da aplicação processem informações de forma independente, sem a necessidade de bloqueio ou expectativa por respostas imediatas. Essa característica é particularmente relevante no domínio de *delivery*, no qual diversas operações como atualização de status de pedidos, processamento de pagamentos e despacho de entregadores podem ser executadas de forma paralela e desacoplada.

Adicionalmente, a arquitetura orientada a eventos contribui para a resiliência do sistema, uma vez que falhas em um componente não necessariamente impedem o funcionamento dos demais. Por outro lado, essa abordagem também introduz desafios importantes, como a necessidade de lidar com consistência eventual e maior complexidade no rastreamento de fluxos de execução. Esses aspectos exigem a adoção de mecanismos adicionais de monitoramento, observabilidade e controle de eventos, conforme também apontado por Richardson (2018).

2.3 Sistemas de Mensageria e Processamento Assíncrono

O uso de sistemas de mensageria constitui um elemento fundamental na implementação de arquiteturas orientadas a eventos, permitindo a comunicação assíncrona entre diferentes componentes de um sistema distribuído. Nesse modelo, as mensagens são enviadas para intermediários, como filas ou tópicos, que atuam como canais de comunicação desacoplados entre produtores e consumidores. Sistemas de mensageria distribuídos, especialmente aqueles baseados em tópicos, oferecem características como alta disponibilidade, persistência de dados e capacidade de processamento em larga escala. De acordo com Kleppmann (2017), esses sistemas são essenciais para garantir a confiabilidade e a escalabilidade em aplicações modernas orientadas a dados. Além disso, Richardson (2018) destaca que o uso de mensageria é um dos principais mecanismos para viabilizar comunicação assíncrona e desacoplada em arquiteturas baseadas em eventos.

Ao considerar alternativas, uma abordagem comum é o uso de filas simples implementadas em memória ou diretamente em bancos de dados. Nesse modelo, as mensagens são armazenadas de forma centralizada e consumidas sequencialmente pelos processos interessados. Embora essa solução seja mais simples de implementar e adequada para sistemas de menor escala, ela apresenta limitações significativas em termos de escalabilidade, tolerância a falhas e desempenho sob alta carga. Em cenários de maior complexidade, falhas no armazenamento ou gargalos de processamento podem comprometer toda a comunicação entre os componentes.

Outra alternativa é a comunicação direta entre serviços, sem o uso de intermediários de mensageria, geralmente por meio de chamadas síncronas via APIs. Embora esse modelo reduza a necessidade de infraestrutura adicional, ele aumenta o acoplamento entre os serviços e limita a capacidade de processamento assíncrono. Conforme discutido por Newman (2015), esse tipo de abordagem pode dificultar a escalabilidade e tornar o sistema mais sensível a falhas em cadeia, especialmente em ambientes distribuídos.

Diante desse cenário, a adoção de um sistema de mensageria distribuído baseado em tópicos neste trabalho se justifica pela necessidade de suportar alta demanda, garantir durabilidade das mensagens e permitir o processamento assíncrono de tarefas. Essa abordagem possibilita que operações mais custosas, como processamento de pedidos, envio de notificações e atualização de status, sejam executadas fora do fluxo principal de requisições, reduzindo o tempo de resposta percebido pelo usuário e melhorando a experiência geral do sistema.

Adicionalmente, o uso de mensageria contribui para o aumento da resiliência, uma vez que os componentes passam a operar de forma mais independente, podendo processar mensagens conforme sua disponibilidade. No entanto, essa abordagem também introduz desafios importantes, como a necessidade de garantir idempotência no processamento, controle de duplicidade de mensagens e tratamento adequado de falhas. Esses aspectos aumentam a complexidade do sistema e exigem a implementação de mecanismos adicionais de controle e monitoramento, conforme também destacado por Kleppmann (2017).

2.4 Cache Distribuído e Gerenciamento de Estado

A utilização de cache distribuído é uma estratégia amplamente empregada em sistemas distribuídos com o objetivo de reduzir latência e otimizar o desempenho de operações críticas, especialmente aquelas que envolvem acesso frequente a dados. Nesse modelo, informações frequentemente requisitadas são armazenadas em memória, permitindo acesso mais rápido em comparação com consultas diretas a bancos de dados persistentes. De acordo com Kleppmann (2017), o uso de cache é uma técnica essencial para melhorar a eficiência de leitura em sistemas de alta escala, reduzindo a carga sobre componentes mais custosos, como bancos de dados relacionais. Além disso, Fowler (2003) descreve o cache como um padrão recorrente em aplicações corporativas para aumentar o desempenho e a escalabilidade.

Ao considerar alternativas, a abordagem mais simples consiste em utilizar exclusivamente o banco de dados como fonte de leitura e escrita, sem a introdução

de uma camada intermediária de *cache*. Nesse modelo, todas as requisições são processadas diretamente pelo banco de dados, o que simplifica a arquitetura e elimina problemas relacionados à consistência entre diferentes camadas. No entanto, essa estratégia pode se tornar inviável em cenários de alta concorrência, nos quais o volume de acessos simultâneos pode gerar gargalos, aumentando a latência e comprometendo o desempenho global do sistema.

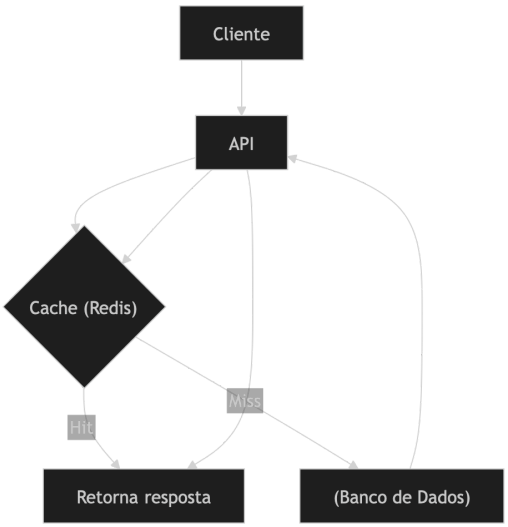
Outra alternativa é o uso de *cache* local (em memória dentro da própria aplicação), no qual cada instância do sistema mantém sua própria cópia dos dados mais acessados. Essa abordagem reduz a necessidade de infraestrutura adicional e pode melhorar o desempenho em aplicações de menor escala. Contudo, em ambientes distribuídos com múltiplas instâncias, o cache local apresenta limitações relacionadas à sincronização de dados, podendo gerar inconsistências entre diferentes nós da aplicação.

Diante desse contexto, a adoção de um *cache* distribuído neste trabalho se justifica pela necessidade de equilibrar desempenho e consistência em um ambiente com múltiplas instâncias da aplicação. Ao centralizar o armazenamento de dados em memória em um sistema compartilhado, é possível garantir maior coerência entre os dados acessados e, ao mesmo tempo, reduzir significativamente a latência das operações. Essa estratégia é particularmente relevante em plataformas de *delivery*, nas quais operações de leitura intensiva, como consulta de produtos, status de pedidos e disponibilidade de entregadores, ocorrem com alta frequência.

Adicionalmente, o uso de *cache* distribuído contribui para a redução da carga sobre o banco de dados relacional, evitando que este se torne um ponto de estrangulamento do sistema. Por outro lado, a introdução dessa camada também traz desafios importantes, especialmente no que diz respeito à consistência dos dados. É necessário garantir a sincronização entre o *cache* e a fonte de verdade, bem como definir estratégias adequadas de expiração e invalidação de dados, a fim de evitar inconsistências e comportamentos inesperados. Esses aspectos exigem cuidados adicionais no projeto e implementação da solução, conforme discutido por Kleppmann (2017).

A utilização de *cache* distribuído para otimização de desempenho é demonstrada na Figura 2.

Figura 2 – Uso de cache para otimização de desempenho



Fonte: Elaborado pelo autor (2026).

2.5 Modelagem de Domínio e Domain-Driven Design (DDD)

A modelagem de domínio adotada neste trabalho baseia-se nos princípios do *Domain-Driven Design* (DDD), uma abordagem que propõe a organização do sistema em torno do domínio do problema, priorizando a representação das regras de negócio e seus comportamentos. Nesse paradigma, o *software* é estruturado a partir de conceitos do mundo real, utilizando elementos como entidades, agregados e serviços de domínio, de forma a refletir com maior fidelidade as necessidades do negócio. Segundo Evans (2003), o DDD tem como objetivo principal alinhar o modelo de *software* à linguagem e às regras do domínio, facilitando a comunicação entre desenvolvedores e especialistas do negócio. Essa abordagem é aprofundada por Vernon (2013), que destaca a importância da modularização do domínio para garantir escalabilidade e manutenção em sistemas complexos.

Ao considerar alternativas arquitetônicas, uma abordagem bastante comum é a modelagem orientada a dados, na qual a estrutura do sistema é definida principalmente a partir do banco de dados e de suas tabelas relacionais. Nesse modelo, o desenvolvimento normalmente começa pela definição das entidades persistidas e de seus relacionamentos, fazendo com que a aplicação seja construída em torno da camada de armazenamento. Como consequência, as classes de domínio frequentemente acabam se tornando apenas representações diretas das tabelas do banco, contendo atributos, métodos simples de acesso (*getters* e *setters*) e operações básicas de persistência.

Nesse tipo de abordagem, as regras de negócio deixam de estar concentradas nas próprias entidades e passam a ser distribuídas entre controladores, serviços, repositórios ou até consultas SQL específicas. Isso faz com que o comportamento do sistema fique fragmentado em diferentes camadas da aplicação, dificultando a compreensão do domínio de negócio como uma unidade coesa. Além disso, alterações em regras operacionais podem exigir modificações em múltiplos pontos do sistema, aumentando o acoplamento e o custo de manutenção.

Esse cenário é frequentemente associado ao conceito de modelos anêmicos (Anemic Domain Model), discutido por Martin Fowler. Segundo Fowler (2003), um modelo anêmico ocorre quando as entidades do domínio funcionam apenas como estruturas de dados, sem encapsular comportamento ou regras de negócio relevantes. Em outras palavras, os objetos representam apenas o estado da aplicação, enquanto toda a lógica fica centralizada em serviços externos. Embora essa estratégia possa simplificar o desenvolvimento inicial e seja amplamente utilizada em aplicações CRUD tradicionais, ela tende a reduzir a expressividade do domínio, dificultar a reutilização das regras de negócio e comprometer a evolução do sistema em cenários mais complexos.

Outra alternativa é a modelagem baseada exclusivamente em camadas técnicas, sem uma separação clara por domínios de negócio. Nesse modelo, o sistema é organizado em camadas como controle, serviço e repositório, mas sem uma preocupação explícita com a delimitação de contextos ou responsabilidades de

negócio. Embora essa abordagem facilite a organização inicial do código, ela pode levar a um alto acoplamento entre diferentes partes do sistema e dificultar a manutenção em aplicações mais complexas.

Diante desse contexto, a adoção do DDD neste trabalho se justifica pela necessidade de representar de forma mais fiel às regras e processos envolvidos no domínio de *delivery*. Ao estruturar o sistema em módulos de domínio, como pedidos, entregas, catálogo e pagamentos, torna-se possível isolar responsabilidades, reduzir o acoplamento e facilitar a evolução independente de cada parte da aplicação. Essa organização contribui diretamente para a clareza do código e para a manutenção do sistema ao longo do tempo.

Adicionalmente, a modelagem orientada ao domínio favorece a implementação de regras de negócio mais robustas e consistentes, uma vez que estas passam a estar centralizadas em componentes específicos do domínio. Por outro lado, reconhece-se que essa abordagem exige um maior esforço inicial de modelagem e um entendimento mais aprofundado do domínio do problema, o que pode aumentar o tempo de desenvolvimento nas fases iniciais do projeto. Ainda assim, esse investimento tende a ser compensado pelos ganhos em organização, escalabilidade e manutenção ao longo do ciclo de vida da aplicação.

2.6 Geolocalização e Indexação Espacial

Em aplicações de logística e *delivery*, operações relacionadas à geolocalização desempenham papel fundamental na tomada de decisões em tempo real. Funcionalidades como seleção de entregadores próximos, definição de áreas de cobertura, cálculo de proximidade e distribuição de demanda dependem da capacidade do sistema de processar consultas espaciais de forma eficiente e escalável. Em cenários com grande volume de usuários, entregadores e pedidos simultâneos, a realização contínua de cálculos geográficos pode se tornar um gargalo computacional significativo, especialmente quando múltiplas comparações precisam ser executadas em baixa latência.

Uma abordagem tradicional para resolver esse problema consiste no cálculo direto de distâncias geográficas por meio de fórmulas matemáticas, como a fórmula de Haversine, originalmente proposta para estimar distâncias entre pontos na superfície terrestre a partir de coordenadas esféricas (ROBUSTO, 1957). Esse método calcula a distância entre dois pontos utilizando latitude e longitude, oferecendo boa precisão para aplicações geoespaciais. Entretanto, em cenários com grande volume de dados, essa abordagem exige que o sistema realize sucessivos cálculos matemáticos para cada comparação de proximidade, aumentando significativamente o custo computacional quando milhares de coordenadas precisam ser avaliadas simultaneamente.

Outra alternativa amplamente utilizada é o uso de índices geoespaciais fornecidos pelos próprios bancos de dados, como estruturas baseadas em árvores, a exemplo de R-trees e GiST. Essas soluções permitem otimizar consultas espaciais diretamente na camada de persistência, facilitando operações como busca por regiões, interseções e proximidade geográfica. Apesar de eficientes em muitos cenários, essas abordagens permanecem fortemente dependentes do mecanismo interno do banco de dados utilizado, podendo apresentar limitações em ambientes distribuídos, arquiteturas escaláveis horizontalmente ou aplicações que exigem processamento intensivo em memória. Além disso, mesmo utilizando índices espaciais, muitas operações ainda dependem de cálculos geométricos adicionais para refinamento dos resultados.

Diante dessas limitações, surgiu a necessidade de uma abordagem capaz de reduzir o custo computacional das consultas geográficas, mantendo eficiência em ambientes distribuídos e aplicações de alta escala. Nesse contexto, a indexação espacial baseada em H3 apresenta-se como uma alternativa especializada para organização geoespacial. Desenvolvido pela Uber, o H3 divide a superfície terrestre em uma grade hierárquica de células hexagonais identificadas por índices únicos. Em vez de realizar cálculos contínuos de distância entre coordenadas geográficas, o sistema passa a operar sobre relações discretas entre células adjacentes, transformando problemas contínuos de geolocalização em operações de indexação espacial (BRODSKY, 2018).

Segundo Brodsky (2018), essa estratégia reduz significativamente a complexidade computacional das consultas espaciais, permitindo executar operações de proximidade e vizinhança de maneira mais eficiente. Cada célula hexagonal representa uma área específica da superfície terrestre, possibilitando agrupar localizações próximas dentro de regiões discretas. Assim, operações como identificação de entregadores próximos deixam de depender exclusivamente de cálculos matemáticos repetitivos e passam a utilizar relações entre células vizinhas.

Um dos principais conceitos do H3 é o de resolução, que define o nível de granularidade da malha hexagonal. Em resoluções mais baixas, os hexágonos cobrem áreas maiores, representando regiões amplas com menor detalhamento. À medida que a resolução aumenta, as células tornam-se progressivamente menores, permitindo representar localizações com maior precisão. Essa estrutura hierárquica possibilita ajustar dinamicamente o equilíbrio entre desempenho e precisão conforme a necessidade da aplicação. Conforme destacado por Brodsky (2018), a hierarquia do H3 facilita operações como agregação regional, expansão geográfica de consultas e busca por vizinhança (*k-ring*), permitindo identificar rapidamente células adjacentes. Complementarmente, Martin Kleppmann ressalta que o uso de índices especializados e hierárquicos constitui uma estratégia importante para otimizar consultas em sistemas distribuídos de grande escala (KLEPPMANN, 2017).

A escolha do H3 neste trabalho está diretamente relacionada às necessidades operacionais de uma plataforma de *delivery*. No contexto da aplicação desenvolvida, o sistema necessita localizar entregadores disponíveis próximos ao ponto de coleta em tempo real, garantindo baixa latência mesmo sob alta concorrência. Para isso, coordenadas geográficas obtidas por meio da integração com o OpenStreetMap, projeto colaborativo de mapeamento geográfico aberto amplamente utilizado em aplicações de geolocalização (OPENSTREETMAP FOUNDATION, 2026), são convertidas em índices H3, permitindo realizar consultas de proximidade utilizando relações entre células vizinhas.

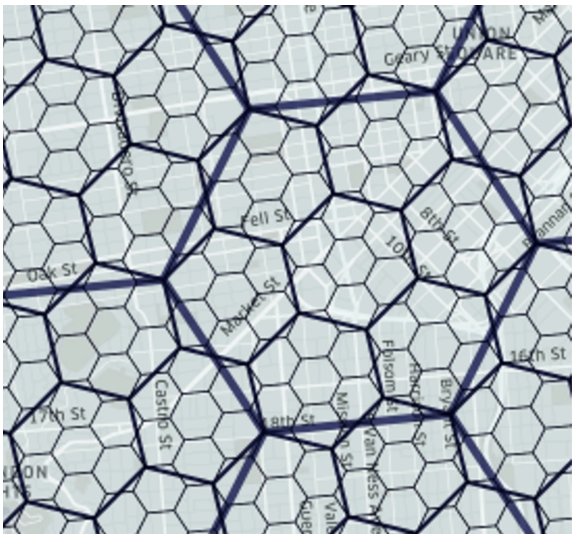
Conforme observado no fluxo de despacho de entregas implementado neste trabalho, a busca por entregadores disponíveis utiliza operações de vizinhança entre índices H3 para localizar candidatos próximos ao ponto de coleta. Essa abordagem

reduz a necessidade de cálculos contínuos de distância e melhora o desempenho das operações de despacho, especialmente em cenários de alta demanda. Além disso, o uso de diferentes níveis de resolução permite adaptar o comportamento do sistema conforme a densidade de entregadores em cada região, possibilitando buscas mais amplas em áreas com baixa disponibilidade e consultas mais precisas em regiões urbanas densas.

Por outro lado, a adoção do H3 também introduz desafios técnicos, como a necessidade de adaptação do modelo de dados para armazenamento de índices geoespaciais e a curva de aprendizado associada às operações específicas da biblioteca. Ainda assim, os ganhos em desempenho, flexibilidade e escalabilidade justificam sua utilização como estratégia de indexação espacial no contexto deste trabalho.

A Figura 3 ilustra a estrutura de indexação espacial do H3, destacando a divisão da superfície terrestre em células hexagonais e seus diferentes níveis de resolução, bem como a relação de vizinhança entre células adjacentes.

Figura 3 – Estrutura hierárquica da indexação espacial H3 com diferentes níveis de resolução e vizinhança entre células hexagonais



Fonte: Brodsky (2018).

2.7 Escalabilidade e Sistemas Distribuídos

O sistema foi projetado com foco em escalabilidade horizontal, uma estratégia que consiste na adição de múltiplas instâncias de um mesmo componente para distribuir a carga de processamento. Diferentemente da escalabilidade vertical, que se baseia no aumento de recursos computacionais (como CPU e memória) de uma única máquina, a abordagem horizontal permite maior flexibilidade, disponibilidade e resiliência em ambientes distribuídos. De acordo com Kleppmann (2017), sistemas distribuídos modernos tendem a privilegiar a escalabilidade horizontal devido à sua capacidade de lidar com grandes volumes de dados e requisições de forma mais eficiente. Além disso, Newman (2015) destaca que a replicação de serviços é um dos principais mecanismos para garantir alta disponibilidade e tolerância a falhas em arquiteturas modernas.

Ao considerar alternativas, a escalabilidade vertical é frequentemente adotada em sistemas mais simples ou em fases iniciais de desenvolvimento. Nesse modelo, o desempenho é aumentado por meio da ampliação dos recursos de hardware de uma única instância, o que simplifica a gestão da infraestrutura e reduz a complexidade inicial. No entanto, essa abordagem apresenta limitações claras, como custos elevados, dependência de hardware específico e menor tolerância a falhas, uma vez que a indisponibilidade da máquina impacta diretamente todo o sistema.

Outra alternativa é a utilização exclusiva de comunicação síncrona entre componentes, geralmente baseada em APIs REST. Nesse modelo, todas as operações dependem de respostas imediatas, o que facilita o controle do fluxo de execução e a consistência dos dados. Contudo, essa abordagem pode gerar gargalos de desempenho e aumentar o acoplamento entre os serviços, além de tornar o sistema mais sensível a falhas em cadeia, conforme discutido por Fielding (2000) e Kleppmann (2017).

Diante desse cenário, a combinação de escalabilidade horizontal com o uso de comunicação síncrona e processamento assíncrono foi adotada neste trabalho como forma de equilibrar desempenho, consistência e flexibilidade. Operações

críticas, que exigem resposta imediata ao usuário, são tratadas por meio de APIs síncronas, garantindo previsibilidade e controle. Por outro lado, tarefas que não demandam resposta instantânea, como processamento de eventos, envio de notificações e atualização de estados, são executadas de forma assíncrona, permitindo melhor aproveitamento dos recursos do sistema e redução da latência percebida.

Adicionalmente, essa abordagem contribui para a resiliência do sistema, uma vez que falhas em componentes assíncronos não comprometem diretamente o fluxo principal da aplicação. No entanto, essa estratégia também introduz desafios importantes, especialmente relacionados à consistência de dados e à sincronização entre componentes distribuídos. Em cenários de falha, é necessário adotar mecanismos adicionais para garantir a integridade das informações e a correta coordenação das operações, aspectos amplamente discutidos por Kleppmann (2017) e Richardson (2018).

3 METODOLOGIA

A metodologia adotada neste trabalho baseia-se no desenvolvimento estruturado de um sistema *backend* utilizando práticas modernas de engenharia de software, com foco em organização arquitetural, escalabilidade, desacoplamento entre componentes e validação prática da solução implementada.

Este trabalho caracteriza-se como uma pesquisa aplicada, pois tem como objetivo a resolução de um problema prático por meio do desenvolvimento de uma solução computacional voltada ao contexto de plataformas de *delivery*.Quanto à abordagem metodológica, trata-se de um estudo de caso de natureza quantitativa, apoiado em métricas objetivas coletadas automaticamente cobertura de testes, número de testes e *assertions*, e linhas de código alteradas em experimentos de substituição de infraestrutura, complementado por avaliações qualitativas realizadas pelo autor sobre a adequação arquitetural das decisões tomadas.. Do ponto de vista dos objetivos, a pesquisa possui caráter exploratório, buscando investigar, aplicar e avaliar conceitos modernos de engenharia de software em um cenário realista de alta complexidade operacional.

Além da implementação prática do sistema, o trabalho também contempla a análise crítica das decisões arquitetônicas adotadas ao longo do desenvolvimento. Dessa forma, a pesquisa não se limita apenas à construção técnica da aplicação, nem exclusivamente à discussão teórica sobre padrões arquitetônicos. O estudo combina ambas as perspectivas, apresentando simultaneamente a implementação da solução e a fundamentação das escolhas realizadas, incluindo aspectos relacionados à arquitetura hexagonal, comunicação assíncrona baseada em eventos, escalabilidade, separação de responsabilidades e indexação geoespacial com H3. Assim, o trabalho busca demonstrar não apenas o funcionamento da aplicação desenvolvida, mas também os critérios técnicos e arquitetônicos que motivaram cada decisão de projeto.

3.1 Levantamento de Requisitos

O desenvolvimento do sistema iniciou-se com o levantamento detalhado dos requisitos, etapa fundamental para garantir que a solução proposta atendesse às necessidades do domínio. Foram identificados tanto requisitos funcionais, relacionados às funcionalidades do sistema, quanto requisitos não funcionais, que envolvem aspectos como desempenho, segurança e escalabilidade.

A motivação para este trabalho decorre da experiência profissional do autor atuando em uma plataforma de *delivery*, contexto no qual emergiu o interesse em investigar, de forma independente, como um sistema desse tipo poderia ser projetado e construído a partir de princípios modernos de engenharia de software. O levantamento de requisitos, portanto, não decorreu de um processo formal de entrevistas com *stakeholders*, mas da vivência prática do autor com os problemas reais desse domínio, somada às habilidades técnicas e à capacidade de arquitetura de *software* desenvolvidas ao longo de sua trajetória profissional, o que permitiu inferir com propriedade quais funcionalidades, atores e regras de negócio um sistema de *delivery* precisa contemplar para ser minimamente completo e realista.

A partir dessa experiência, os requisitos funcionais foram consolidados sob a forma de histórias de usuário (*user stories*), formato amplamente utilizado em metodologias ágeis para expressar necessidades sob a perspectiva de cada ator do sistema. A tabela 1 apresenta as principais histórias de usuário consideradas, organizadas pelos quatro perfis de atores identificados (cliente, parceiro/loja, entregador e administrador).

Tabela 1 – História de Usuários

Ator	História de Usuario
Cliente	Como cliente, quero ver a cotação de entrega antes de finalizar o pedido, para decidir se desejo prosseguir com a compra.
Cliente	Como cliente, quero adicionar itens ao carrinho e aplicar cupons de desconto, para montar meu pedido antes do *checkout*

Cliente	Como cliente, quero finalizar meu pedido (*checkout*), para que ele seja processado e encaminhado à loja.
Cliente	Como cliente, quero avaliar minha experiência com a loja após a entrega, para fornecer retorno sobre a qualidade do produto e da entrega.
Parceiro/Loja	Como parceiro, quero atualizar o status de disponibilidade da minha loja, para abri-la ou fechá-la para novos pedidos.
Parceiro/Loja	Como parceiro, quero gerenciar meu catálogo, categorias e produtos, para manter meu cardápio atualizado.
Entregador	Como entregador, quero registrar minha localização durante a entrega, para que o cliente acompanhe o progresso em tempo real.
Administrador	Como administrador, quero criar e gerenciar campanhas e promoções, para incentivar vendas em datas ou regiões específicas.

Fonte: Elaborado pelo autor (2026).

3.2 Definição da Arquitetura

A escolha por uma arquitetura de monólito modular foi motivada pela necessidade de equilibrar simplicidade de implantação com organização interna do sistema. Diferentemente de uma arquitetura de microserviços, que oferece alto grau de desacoplamento, mas aumenta significativamente a complexidade operacional, especialmente no que diz respeito à orquestração, observabilidade e comunicação, o monólito modular permite centralizar o *deploy*, mantendo a separação lógica entre os componentes por meio de módulos bem definidos.

No diagrama apresentado na Figura 4, é possível observar que o sistema está estruturado em camadas e responsabilidades bem delimitadas. A camada de entrada é composta pelos atores externos (Cliente, Parceiro e Administrador), que interagem exclusivamente com a API da aplicação. Essa API atua como ponto central de entrada, sendo responsável por validação, autenticação, orquestração de regras de negócio e comunicação com os demais componentes internos e externos.

Internamente, o sistema é dividido em três grandes blocos: Aplicação, Tarefas Agendadas (Crons) e Workers (Consumidores de Eventos). A camada de aplicação concentra a API principal, enquanto o módulo de tarefas agendadas, executa rotinas periódicas, como cancelamento automático de pedidos e solicitação de avaliações de lojas. Essas rotinas são essenciais para manter a consistência do sistema sem depender de ações diretas do usuário.

Outro ponto importante evidenciado no diagrama é o uso de comunicação assíncrona por meio do Kafka. A API pública eventos em tópicos, que são consumidos por diferentes *workers* especializados, como criação e atualização de pedidos, controle de estoque, processamento de pagamentos, fluxo financeiro e avaliações. Esse modelo orientado a eventos reduz o acoplamento entre os componentes e melhora a escalabilidade do sistema, permitindo que tarefas mais pesadas sejam processadas de forma independente da requisição principal.

A escolha do Kafka como sistema de mensageria se justifica por características específicas de sua implementação na plataforma: o produtor aguarda a confirmação de entrega antes de retornar o controle à aplicação, garantindo que o evento tenha sido efetivamente publicado; o consumidor desabilita o *commit* automático de *offset*, confirmando o processamento de uma mensagem somente após sua execução bem-sucedida, o que caracteriza uma semântica de entrega *at-least-once*; e mensagens cujo processamento falha são automaticamente redirecionadas para um tópico de *Dead Letter Queue* específico, evitando perda silenciosa de eventos. Essas características tornam o Kafka adequado ao volume e à criticidade dos oito fluxos de negócio assíncronos da plataforma (criação e atualização de pedidos, controle de estoque, processamento de pagamentos, lançamentos financeiros, despacho de entregadores, avaliação de lojas e rastreamento de entregas), descritos em detalhe na Seção 3.6.

A camada de infraestrutura, composta por Redis, banco de dados relacional, sistema de mensageria (Kafka) e armazenamento de arquivos (*bucket*), fornece os serviços necessários para persistência, cache, comunicação e armazenamento. O Redis é utilizado principalmente para cache e controle de estados temporários, sendo empregado não apenas como armazenamento simples de chave-valor, mas também por meio de estruturas de dados nativas como conjuntos ordenados (*sorted sets*), conjuntos (*sets*) e operações atômicas de incremento, necessárias ao domínio de despacho de entregadores. A tabela 2 detalha os principais dados mantidos em cache e seus respectivos tempos de expiração. Já o banco de dados relacional garante a persistência das informações transacionais, e o *bucket* é responsável pelo armazenamento de arquivos, como imagens de produtos.

Tabela 2 – Dados Armazenados em cache Redis

Dado Armazenado	Tempo de Expiração TTL	Estrutura Utilizada
Estado de entrega e do entregador	6 horas	Chave simples
Disponibilidade de entregadores por célula geográfica H3	Sem expiração	Conjunto (*set*)
Ofertas de entrega pendentes para um entregador	30 minutos	Chave simples
Fila de retentativas de despacho	Sem expiração	Conjunto ordenado (*sorted set*), pontuado pelo horário da próxima tentativa
Contador de tentativas de despacho	1 hora	Chave simples

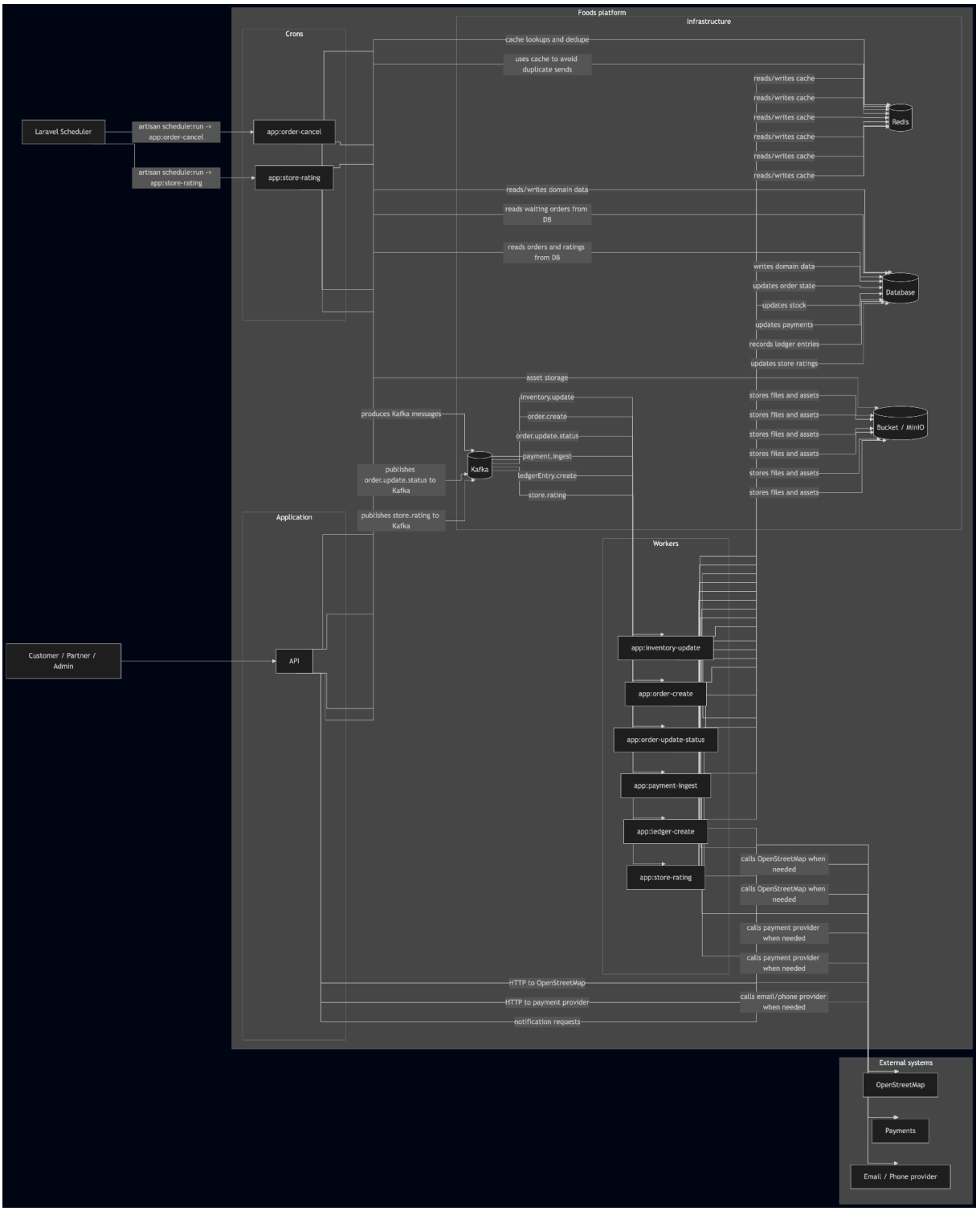
Fonte: Elaborado pelo autor (2026).

Além disso, o sistema integra-se com serviços externos, como APIs de mapas (OPENSTREETMAP FOUNDATION, 2026) para geolocalização, gateways de pagamento para processamento financeiro e serviços de notificação (*e-mail* e SMS), demonstrando a necessidade de interoperabilidade com sistemas terceiros.

A adoção de princípios da arquitetura hexagonal dentro desse modelo reforça o isolamento entre domínio e infraestrutura, garantindo que as regras de negócio permaneçam independentes de detalhes tecnológicos. Dessa forma, mesmo estando em um monólito, o sistema mantém características desejáveis como baixo acoplamento, alta coesão e facilidade de manutenção.

Portanto, a arquitetura apresentada na Figura 4 evidencia uma solução híbrida e pragmática, que combina a simplicidade operacional do monólito com práticas modernas de desacoplamento, como processamento assíncrono e separação por responsabilidades, tornando-se adequada para o contexto e os requisitos do sistema proposto.

Figura 4 – Arquitetura geral da plataforma de delivery



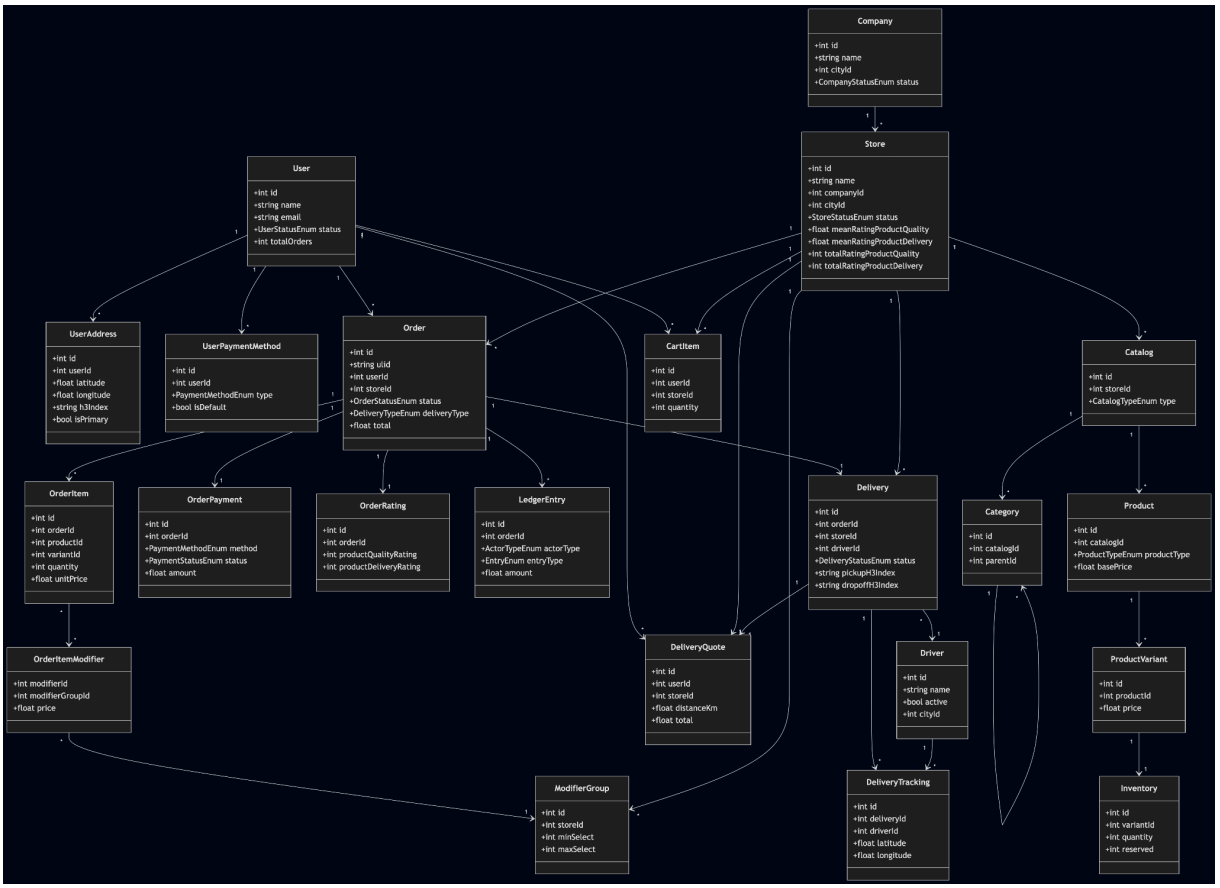
Fonte: Elaborado pelo autor (2026).

3.3 Modelagem do Domínio

A modelagem do domínio foi conduzida a partir da análise dos principais fluxos de negócio de uma plataforma de *delivery*, considerando a interação entre usuários, estabelecimentos, entregadores e processos internos. Com base nisso, foram definidas as principais entidades do sistema como pedidos, pagamentos, entregas e avaliações bem como seus relacionamentos e responsabilidades.

A Figura 5 apresenta o diagrama de classes com as principais entidades do domínio e seus relacionamentos, elaborado a partir da estrutura de objetos de transferência de dados (*DTOs*) que representam o modelo de domínio na camada *core* da aplicação. O diagrama evidencia, entre outras relações, que um pedido (*Order)* agrega itens (*OrderItem*), possui um único registro de pagamento (*OrderPayment)* e uma única avaliação (*OrderRating)*, além de originar lançamentos financeiros (*LedgerEntry)* e um registro de entrega (*Delivery)* associado; e que uma loja (*Store*) pertence a uma empresa (*Company*) e agrega catálogos, grupos de modificadores e cotações de entrega.

Figura 5 – Diagrama das Classes



Também foram estabelecidos os estados e ciclos de vida das entidades, com destaque para o fluxo de pedidos, que envolve etapas como criação, confirmação, preparo, despacho, entrega e finalização. Essa definição permitiu estruturar de forma clara as transições de estado e as regras associadas a cada etapa do processo.

As regras de negócio foram incorporadas diretamente ao domínio, incluindo a precificação de entregas com base em distância e variação de demanda, a distribuição de pedidos considerando proximidade e disponibilidade de entregadores, e o cálculo de avaliações de estabelecimentos por meio de modelos incrementais. Além disso, foram introduzidas configurações por cidade, permitindo ajustar parâmetros como taxas e multiplicadores conforme a localidade, o que aumenta a flexibilidade e aderência do sistema a diferentes contextos operacionais.

O domínio foi organizado em módulos independentes, como pedidos, entregas, catálogo e pagamentos, favorecendo a separação de responsabilidades e a redução de acoplamento. Em comparação a abordagens centradas apenas na estrutura de dados, essa modelagem prioriza o comportamento do sistema, resultando em maior clareza, melhor organização e facilidade de evolução.

3.4 Implementação

A implementação do sistema foi realizada com base nos princípios definidos na arquitetura proposta, especialmente aqueles relacionados à arquitetura hexagonal, separação de responsabilidades e processamento orientado a eventos. O desenvolvimento seguiu uma abordagem incremental, permitindo a evolução contínua das funcionalidades e a validação progressiva dos principais fluxos de negócio.

A solução foi desenvolvida utilizando PHP 8.5 e o framework Laravel 12, escolhido por oferecer uma base sólida para construção de APIs robustas, com suporte a organização em camadas, injeção de dependências e integração facilitada com serviços externos; o código-fonte completo da aplicação está disponível no GitHub (GITHUB, 2026). Para otimização de desempenho, foi utilizado o Laravel Octane (LARAVEL, 2026) em conjunto com o Swoole (SWOOLE LABS, 2026), permitindo a execução persistente da aplicação em memória e reduzindo significativamente o tempo de resposta em comparação ao modelo tradicional baseado em requisição por processo. Essa escolha está alinhada com os objetivos de escalabilidade e eficiência discutidos na fundamentação teórica.

A estrutura do sistema foi organizada em camadas bem definidas, refletindo os princípios da arquitetura limpa. As requisições HTTP são recebidas pelos controladores, que atuam como ponto de entrada da aplicação, sendo responsáveis por validação inicial e encaminhamento para os casos de uso. Esses casos de uso concentram a lógica de aplicação, coordenando as regras de negócio e orquestrando a interação entre os diferentes componentes do sistema.

No núcleo da aplicação, os casos de uso implementam operações específicas do domínio, como criação de pedidos, cálculo de entregas e processamento de pagamentos. Cada operação recebe objetos de entrada estruturados, garantindo padronização dos dados e facilitando validações. A lógica de negócio é mantida isolada de detalhes de infraestrutura, seguindo o princípio de inversão de dependência, no qual o domínio depende de abstrações e não de implementações concretas.

O acesso a dados e a serviços externos foi realizado por meio de interfaces definidas no domínio e implementadas na camada de infraestrutura, utilizando o padrão de repositório e adaptadores. Essa abordagem permite desacoplar a lógica de negócio de tecnologias específicas, como banco de dados, cache e serviços externos, facilitando testes e futuras substituições.

Além do fluxo síncrono, a implementação incorporou processamento assíncrono por meio de mensageria, utilizando o Apache Kafka como sistema de comunicação entre componentes. Após a execução de determinadas operações, como a criação de um pedido, eventos são publicados em tópicos e consumidos por processos especializados (*workers*), responsáveis por executar tarefas como atualização de estoque, processamento financeiro e despacho de entregadores. Esse modelo reduz o acoplamento entre componentes e melhora a escalabilidade do sistema.

Complementarmente, foram implementadas rotinas periódicas (crons), responsáveis por automatizar processos como cancelamento de pedidos por tempo limite, reprocessamento de tarefas e envio de notificações. Esses mecanismos garantem a consistência do sistema mesmo na ausência de interação direta do usuário.

A utilização de cache distribuído, por meio do Redis, foi aplicada para otimização de desempenho, especialmente em operações de leitura frequente e controle de estados temporários. Essa estratégia reduz a carga sobre o banco de dados e melhora o tempo de resposta da aplicação.

Por fim, a integração com serviços externos, como provedores de pagamento, geolocalização e notificações, foi realizada por meio de adaptadores específicos, mantendo o isolamento entre domínio e infraestrutura. Essa abordagem reforça os princípios da arquitetura hexagonal e contribui para a construção de um sistema flexível, escalável e alinhado com boas práticas de engenharia de *software*.

3.5 Integração com Serviços Externos

A integração com serviços externos foi realizada por meio de interfaces definidas na camada de domínio e implementadas na camada de infraestrutura, seguindo o padrão de adaptadores proposto pela arquitetura hexagonal. Essa abordagem permite isolar a lógica de negócio de dependências externas, garantindo maior flexibilidade e facilidade de manutenção.

No contexto deste trabalho, o único serviço externo efetivamente integrado foi o OpenStreetMap (OPENSTREETMAP FOUNDATION, 2026), utilizado principalmente para obtenção de coordenadas geográficas, como latitude e longitude. Essas informações são fundamentais para o funcionamento do sistema, pois servem como base para a indexação espacial utilizando H3, permitindo a realização de cálculos eficientes de proximidade e distribuição de entregadores. A integração foi encapsulada em um adaptador específico, responsável por tratar a comunicação com a API e normalizar os dados geográficos utilizados internamente.

Para os demais serviços, como gateway de pagamentos e sistemas de notificação (e-mail e SMS), foram utilizadas implementações simuladas (mocks). Essa decisão foi motivada por restrições de custo e pela natureza do trabalho, cujo foco está na modelagem arquitetural e na organização do sistema. Ainda assim, essas integrações foram projetadas por meio de interfaces bem definidas, permitindo a substituição por provedores reais futuramente sem impacto na lógica de negócio.

Essa estratégia evidencia a aplicação do princípio de inversão de dependência, no qual o domínio não depende de implementações concretas. Além

disso, possibilita a validação dos fluxos do sistema de forma controlada, garantindo previsibilidade durante o desenvolvimento.

Em comparação a integrações diretas, essa abordagem reduz o acoplamento e facilita a manutenção e evolução do sistema. Dessa forma, mesmo com o uso de *mocks* em parte das integrações, a solução permanece preparada para cenários reais de produção, reforçando sua robustez e extensibilidade.

3.6 Processamento Assíncrono

Para lidar com operações de maior complexidade, foram implementados mecanismos de processamento assíncrono utilizando consumidores de eventos.

Esses componentes são responsáveis por executar tarefas como atualização de status de pedidos, processamento de pagamentos, controle de estoque, despacho de entregadores, além disso, foram implementadas rotinas periódicas (cron jobs) para tratar eventos como cancelamento de pedidos por timeout e reprocessamento de tarefas.

A plataforma implementa oito *workers* Kafka, cada um consumindo um tópico dedicado e delegando o processamento a um consumidor especializado. A tabela 3 apresenta cada *worker*, o tópico consumido e a principal regra de negócio implementada.

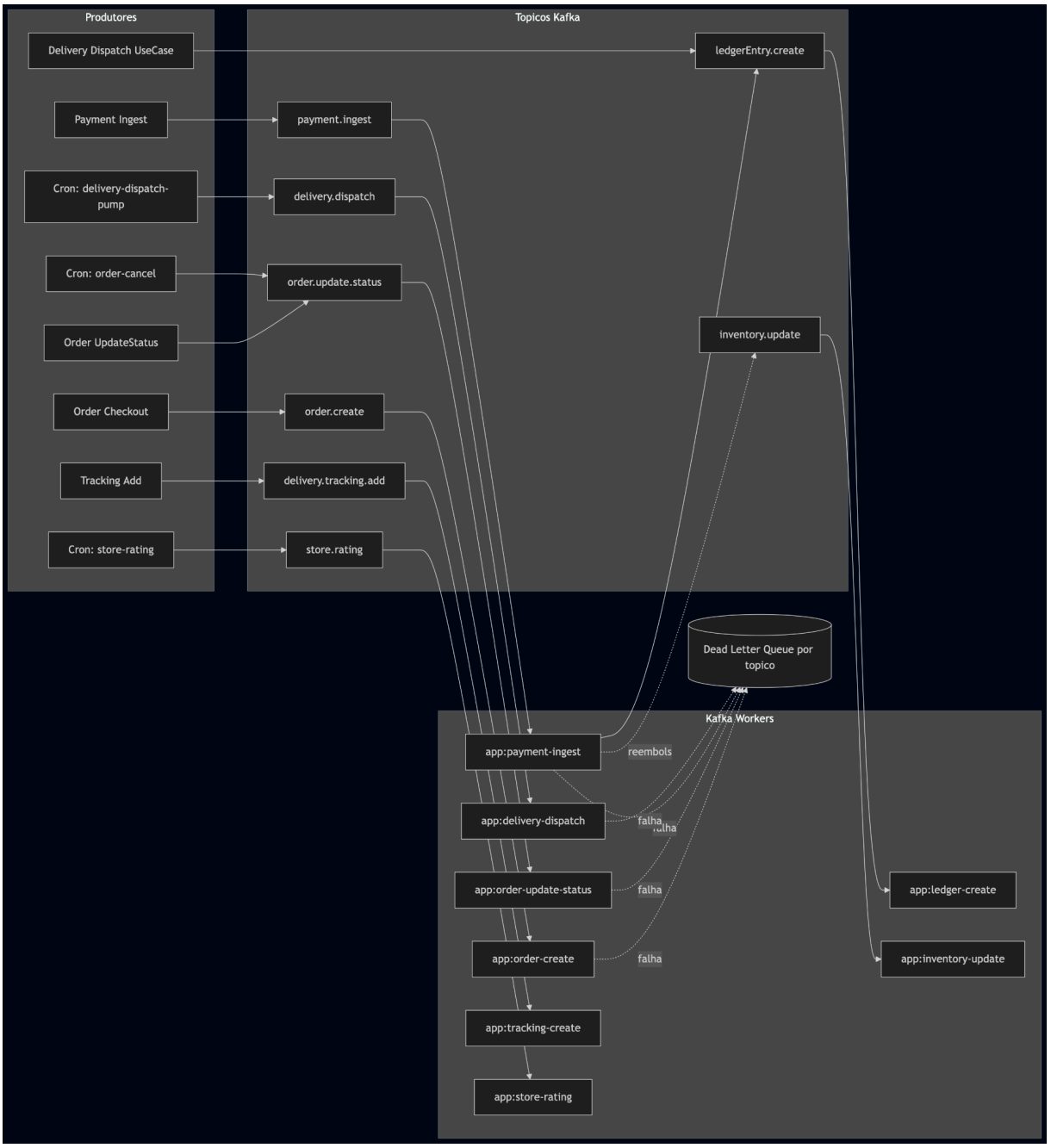
Tabela 3 – Workers Kafka e regras de negócio

Comando	Tópico	Regra de Negócio Principal
app:order-create	*order.create*	Cria o pedido a partir do *job* assíncrono enfileirado no *checkout*.
app:order-update-status	*order.update.status*	Aplica uma máquina de estados que permite apenas transições específicas entre status; ao transicionar para *CANCELLED*, cancela também a transação de pagamento associada antes de persistir a atualização.
app:payment-ingest	*payment.ingest*	Mapeia o status recebido do provedor de pagamento para os status de pedido e pagamento

		correspondentes; em caso de reembolso, repudia um evento de atualização de estoque para cada item do pedido; sempre emite um lançamento financeiro.
app:inventory-update	*inventory.update*	Soma ou subtrai a quantidade em estoque conforme o tipo de operação, impedindo que o saldo se torne negativo.
app:delivery-dispatch	*delivery.dispatch*	Executa o laço de oferta de entrega a entregadores próximos (busca por vizinhança geoespacial H3), ofertando ao próximo candidato disponível e agendando nova tentativa caso a oferta não seja aceita dentro do prazo.
app:tracking-create	*delivery.tracking.add*	Registra o ponto de localização do entregador durante a execução da entrega.
app:store-rating	*store.rating*	Calcula a média ponderada de avaliação da loja (fórmula detalhada na Seção 4.2).
app:ledger-create	*ledgerEntry.create*	Persiste o lançamento financeiro, reunindo o contexto de pedido, loja, empresa, cidade, promoção e entrega associados.

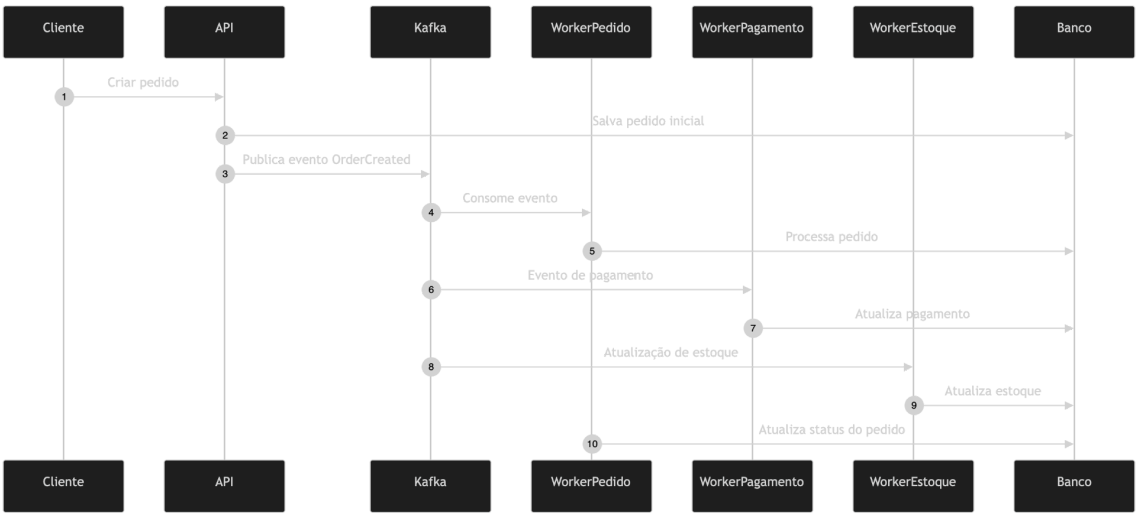
A estrutura de filas segue a convenção “<entidade>.<ação>” para nomeação de tópicos, com relação de um tópico por *worker* e grupo de consumidores (*consumer group*) próprio para cada domínio. A Figura 6 ilustra essa estrutura, evidenciando os produtores, os tópicos e os *workers* consumidores, incluindo o mecanismo de *Dead Letter Queue* acionado em caso de falha no processamento.

Figura 6 – Estrutura de Filas e Tópicos Kafka



Um dos fluxos de processamento assíncrono baseado em eventos é ilustrado na Figura 7, demonstrando a comunicação desacoplada entre os componentes do sistema especificamente no ciclo de vida do pedido.

Figura 7 – Fluxo de processamento orientado a eventos



Fonte: Elaborado pelo autor (2026).

Cabe destacar que o sistema garante semântica de entrega *at-least-once* (confirmação manual de *offset* apenas após o processamento bem-sucedido da mensagem), o que implica a possibilidade teórica de reprocessamento de uma mensagem em cenários de falha entre a execução e a confirmação do *offset*. Para eliminar essa possibilidade, todos os *workers* aplicam um mecanismo de deduplicação centralizado na classe base dos consumidores (*BaseConsumer*): antes de delegar o processamento ao consumidor especializado, uma chave composta pelo tópico, pela partição e pelo *offset* da mensagem identificador único de cada entrega física no Kafka é registrada na camada de cache por meio de uma operação atômica de escrita condicional (*SET* apenas se a chave não existir, com expiração de 24 horas). Caso a chave já exista, a mensagem é descartada como duplicata sem reprocessar a regra de negócio; caso contrário, o processamento prossegue normalmente. A identidade física da mensagem foi escolhida como chave de deduplicação, e não o conteúdo do seu *payload*, porque mensagens de domínios como atualização de estoque ou avaliação de loja podem ter o mesmo conteúdo para dois eventos de negócio legitimamente distintos; deduplicar por conteúdo descartaria, nesses casos, eventos válidos.

Figura 8 – Mecanismo de Deduplicação

```php
final public function __invoke(string $message, ?string $topic = null, ?int $partition = null, ?int
$offset = null): void
{
    if ($topic === null || $partition === null || $offset === null) {
        $this->consume($message);
        return;
    }

    $dedupKey = sprintf('kafka:dedup:%s:%d:%d', $topic, $partition, $offset);

    if (! $this->cacheGateway->setNx($dedupKey, 1, self::DEDUP_TTL_SECONDS)) {
        $this->logger->debug('[Kafka]: duplicate message skipped', [
            'topic' => $topic,
            'partition' => $partition,
            'offset' => $offset,
        ]);

        return;
    }

    $this->consume($message);
}
```

Fonte: Elaborado pelo autor (2026).

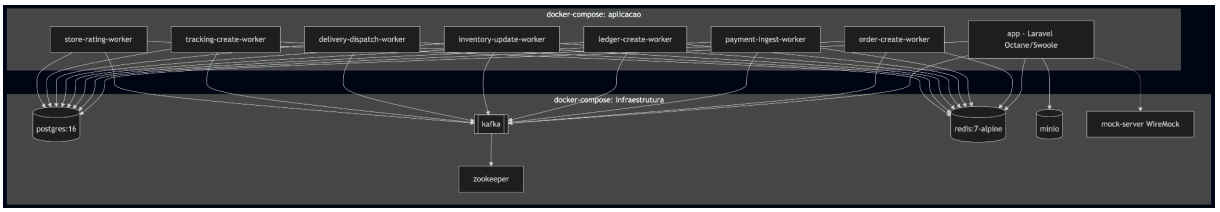
A figura 8 apresenta o mecanismo de deduplicação implementado na classe base de todos os oito consumidores Kafka da aplicação. A identidade física da mensagem (tópico, partição e *offset*) é utilizada como chave de deduplicação, em vez do conteúdo do seu *payload*, pois mensagens de domínios como atualização de estoque ou avaliação de loja podem apresentar o mesmo conteúdo para dois eventos de negócio legitimamente distintos. A centralização na classe base garante que o mecanismo se aplique uniformemente aos oito *workers* sem duplicação de código entre eles.

3.7 Infraestrutura e Ambiente

O ambiente de execução foi configurado utilizando contêineres Docker, permitindo padronização e facilidade de *deploy*. Serviços como banco de dados, cache e mensageria são executados de forma isolada, garantindo maior controle e escalabilidade.

Essa abordagem também facilita a replicação do ambiente em diferentes estágios, como desenvolvimento, testes e produção. A organização da infraestrutura baseada em contêineres é apresentada na Figura 9.

Figura 9 – Infraestrutura baseada em containers



Fonte: Elaborado pelo autor (2026).

Complementarmente aos *workers* Kafka, três rotinas agendadas (*crons*) automatizam processos que não dependem de eventos externos: o cancelamento de pedidos por tempo limite (a cada minuto), o bombeamento de tentativas de despacho de entregadores (a cada minuto) e a publicação de eventos de avaliação de lojas (a cada dia meia noite), detalhadas na Tabela 4.

Tabela 4 – Rotinas Agendadas (crons)

Comando	Regra de Negócio Principal
app:order-cancel	Localiza pedidos parados há mais de 15 minutos em status de aguardando confirmação ou aguardando pagamento, e publica evento de atualização de status para cancelamento ou falha, evitando reprocessamento por meio de um controle em cache
app:delivery-dispatch-pump	Localiza entregas aguardando entregador sem tentativa em curso e dispara o evento de despacho; adicionalmente, verifica retentativas vencidas e as republica, respeitando um limite máximo de 20 tentativas por entrega
app:store-rating-pump	Localiza lojas com avaliações recebidas no dia anterior e dispara um evento de reavaliação por loja.

4 RESULTADOS E ANÁLISE

4.1 Visão Geral da Solução

Os resultados obtidos neste trabalho demonstram a viabilidade técnica da construção de uma plataforma *backend* para *delivery* utilizando uma arquitetura modular baseada em princípios de separação de responsabilidades, processamento assíncrono e organização por domínio de negócio. A solução desenvolvida foi capaz de representar cenários reais presentes em plataformas de *marketplace* de entrega, incluindo gerenciamento de pedidos, cálculo de entrega, processamento de pagamentos, despacho de entregadores e atualização de avaliações de estabelecimentos.

A aplicação foi estruturada de forma a suportar múltiplos fluxos de execução simultaneamente, contemplando operações síncronas por meio de APIs HTTP e operações assíncronas utilizando mensageria baseada em Kafka. Essa abordagem permitiu desacoplar tarefas críticas do fluxo principal da aplicação, reduzindo impacto de operações mais custosas computacionalmente e aumentando a capacidade de escalabilidade do sistema.

Do ponto de vista operacional, a solução implementa mecanismos de gerenciamento de estado temporário utilizando Redis, empregados principalmente nos fluxos de despacho de entregas e controle de tentativas de redistribuição de pedidos. No fluxo de dispatch de entregadores, por exemplo, o sistema utiliza filas assíncronas, controle de tentativas e expiração de ofertas para evitar concorrência e múltiplas atribuições simultâneas de entregadores.

Além disso, o sistema faz uso de geolocalização baseada em índices H3 para cálculo de distância e proximidade geográfica. Esse mecanismo é utilizado principalmente no cálculo de cotação de entrega e no processo de busca de entregadores próximos ao estabelecimento, permitindo representar problemas reais relacionados à logística de *delivery*. O cálculo da cotação considera distância entre

origem e destino, multiplicadores por região, taxa de serviço e lógica de preço dinâmico baseada na relação entre oferta e demanda de entregadores.

A arquitetura implementada também demonstrou capacidade de integração com múltiplos componentes de infraestrutura, incluindo PostgreSQL para persistência relacional, Redis para cache e coordenação temporária, Kafka para mensageria assíncrona e MinIO para armazenamento de arquivos. Essa combinação de tecnologias permitiu reproduzir características normalmente encontradas em aplicações distribuídas modernas.

Dessa forma, os resultados obtidos indicam que a solução proposta foi capaz de atender aos objetivos definidos neste trabalho, servindo como uma base funcional e arquiteturalmente consistente para evolução futura da plataforma.

4.2 Arquitetura Implementada

A arquitetura implementada demonstrou ser adequada para lidar com a complexidade inerente ao domínio de plataformas de *delivery*, permitindo elevada modularização e separação de responsabilidades entre os diferentes componentes da aplicação. A solução foi desenvolvida seguindo princípios inspirados em Arquitetura Hexagonal e *Clean Architecture*, nos quais a lógica de negócio permanece desacoplada das tecnologias externas utilizadas pela aplicação.

A estrutura adotada organiza a aplicação em diferentes camadas, separando controladores HTTP, *factories* de requisição, casos de uso, regras de negócio, *gateways* e *adapters* de infraestrutura. Esse modelo permitiu que as regras centrais do domínio permanecessem independentes de detalhes específicos de implementação, como banco de dados, mecanismos de *cache*, mensageria e integrações externas.

A comunicação entre os componentes da aplicação ocorre principalmente por meio de contratos definidos por interfaces, permitindo que implementações concretas sejam substituídas sem impacto significativo sobre os casos de uso da camada de domínio. Essa abordagem reduz o acoplamento entre infraestrutura e

regras de negócio, favorecendo manutenção, evolução tecnológica e maior capacidade de adaptação da plataforma.

Outro aspecto relevante observado nos resultados foi a utilização de processamento assíncrono orientado a eventos. A aplicação utiliza *workers* consumidores de Kafka para executar operações desacopladas do fluxo HTTP principal, incluindo criação de pedidos, atualização de estoque, processamento financeiro, despacho de entregadores e atualização de avaliações de estabelecimentos.

O fluxo de atualização de avaliação de lojas demonstra esse comportamento assíncrono. Nesse processo, uma rotina agendada identifica diariamente os estabelecimentos que receberam avaliações no dia anterior e publica um evento por loja em um tópico Kafka. Em seguida, um worker especializado consome o evento e recalcula a métrica de reputação do estabelecimento por meio de uma média ponderada incremental: a média das avaliações do dia é combinada com a média histórica da loja, atribuindo-se à leva de avaliações recém-recebidas um peso três vezes maior que o de uma avaliação histórica isolada. Esse mecanismo garante que lojas com maior volume recente de avaliações tenham sua reputação atualizada de forma proporcionalmente mais responsiva, sem, no entanto, descartar o histórico acumulado de avaliações anteriores. Essa abordagem reduz o acoplamento entre componentes e evita que operações de agregação estatística impactem diretamente a experiência do usuário durante o uso da API.

A figura 10 apresenta a implementação da média ponderada incremental descrita anteriormente, na qual o peso atribuído ao lote de avaliações do dia (*NEW_RATING_WEIGHT*) é multiplicado pela quantidade de avaliações recebidas antes de ser combinado com a média histórica da loja.

Figura 10 – Implementação de Média Ponderada

```php
private const NEW_RATING_WEIGHT = 3;

private function weightedAverage(
    ?float $currentAverage,
    int $currentTotal,
    float $newAverage,
    int $newTotal,
): float {
    $weightedNewTotal = $newTotal * self::NEW_RATING_WEIGHT;
    $denominator = $currentTotal + $weightedNewTotal;

    if ($denominator == 0) {
        return $newAverage;
    }

    $currentAverage ??= $newAverage;

    return (($currentAverage * $currentTotal) + ($newAverage * $weightedNewTotal)) / $denominator;
}
```

Fonte: Elaborado pelo autor (2026).

Da mesma forma, o fluxo de dispatch de entregadores utiliza *workers* independentes responsáveis pela distribuição de pedidos para motoristas próximos, gerenciamento de tentativas, controle de *retry* e expiração de ofertas. A separação entre processamento síncrono e assíncrono mostrou-se adequada para sistemas com múltiplas operações concorrentes e requisitos elevados de escalabilidade.

Figura 11 – Laço de Oferta de Entregadores

```php
public function apply(
    string $deliveryUlid,
    array $candidateDriverUlids,
    int $offerTtlSeconds,
): ?string {
    foreach ($candidateDriverUlids as $driverUlid) {
        $driver = $this->deliveryManagementGateway->getDriver($driverUlid);

        if (! $driver || $driver->status !== DriverStatusEnum::AVAILABLE) {
            continue;
        }

        $currentOffer = $this->deliveryManagementGateway->getOffer($deliveryUlid, $driverUlid);

        if ($this->shouldSkipDriver($currentOffer)) {
            continue;
        }

        $this->deliveryManagementGateway->upsertOffer(new Offer(
            deliveryUlid: $deliveryUlid,
            driverUlid: $driverUlid,
            status: OfferStatusEnum::PENDING,
            createdAt: new \DateTimeImmutable(),
            expiresAt: (new \DateTimeImmutable())->modify("+{$offerTtlSeconds} seconds"),
        ));

        $this->deliveryManagementGateway->incrementDispatchAttempts($deliveryUlid);

        return $driverUlid;
    }

    return null;
}
```

Fonte: Elaborado pelo autor (2026).

A figura 11 apresenta o laço responsável por percorrer os entregadores candidatos em ordem de proximidade, ignorando aqueles indisponíveis ou com oferta já rejeitada, aceita ou pendente, e interrompendo a busca assim que uma oferta é registrada com sucesso.

De forma análoga, o fluxo de atualização de status de pedidos utiliza uma máquina de estados que restringe as transições permitidas a partir de cada status, impedindo, por exemplo, que um pedido cancelado seja posteriormente reaberto, ou que um pedido aguardando pagamento avance diretamente para o status de entregue sem passar pelos estados intermediários. A figura 12 apresenta um trecho representativo dessa validação.

Figura 12 – Fluxo de Atualização de Pedido

```php
return match ($current) {
    OrderStatusEnum::WAITING_PAYMENT => $this->isValidNextStatusForWaitingPayment($next),
    OrderStatusEnum::WAITING_CONFIRMATION => $this->isValidNextStatusForWaitingConfirmation($next),
    OrderStatusEnum::CONFIRMED => $this->isValidNextStatusForConfirmed($next),
    OrderStatusEnum::PREPARING => $this->isValidNextStatusForPreparing($next),
    OrderStatusEnum::READY => $this->isValidNextStatusForReady($next),
    OrderStatusEnum::DISPATCHED => $this->isValidNextStatusForDispatched($next),
    OrderStatusEnum::DELIVERED => $this->isValidNextStatusForDelivered($next),
    OrderStatusEnum::CANCELLED => $this->isValidNextStatusForCancelled($next),
    OrderStatusEnum::FAILED => $this->isValidNextStatusForFailed($next),
};

// [...] cada status possui um método correspondente que retorna a lista de
// transições permitidas a partir dele — por exemplo:

private function isValidNextStatusForWaitingPayment(OrderStatusEnum $next): bool
{
    return in_array($next, [
        OrderStatusEnum::WAITING_CONFIRMATION,
        OrderStatusEnum::CANCELLED,
        OrderStatusEnum::FAILED,
    ], true);
}

private function isValidNextStatusForDispatched(OrderStatusEnum $next): bool
{
    return in_array($next, [
        OrderStatusEnum::DELIVERED,
        OrderStatusEnum::FAILED,
    ], true);
}
```

Fonte: Elaborado pelo autor (2026).

Cada status possui um conjunto fechado de transições válidas, retornando falso para qualquer combinação não prevista. Essa estrutura, embora eleve a complexidade ciclomática da classe, é inerente à quantidade de estados válidos do domínio de pedidos, e não um indício de má prática, conforme discutido na Seção 4.3.6.

Outro resultado importante identificado durante o desenvolvimento foi a capacidade de substituição de componentes de infraestrutura com baixo impacto na arquitetura. Como experimento de validação da arquitetura adotada, foi realizada a substituição completa do mecanismo de *cache* Redis por uma implementação alternativa baseada em cache em memória.

A alteração exigiu apenas a criação de um novo *adapter* compatível com os contratos previamente definidos pelos *gateways* da aplicação, além de pequenas modificações nas injeções de dependência realizadas no *AppServiceProvider* do Laravel (LARAVEL, 2026). Toda a substituição foi implementada utilizando aproximadamente 226 linhas de código, sem necessidade de alterações nas regras de negócio, casos de uso ou fluxos centrais do domínio.

Esse resultado evidencia, de forma prática, a efetividade dos princípios de inversão de dependência e desacoplamento arquitetural utilizados no projeto. Como os módulos da camada de domínio dependem exclusivamente de abstrações, a troca da tecnologia responsável pelo gerenciamento de cache pôde ser realizada sem impacto funcional significativo sobre o restante da aplicação.

Para dimensionar esse resultado, observa-se que, em toda a base de código, apenas três classes consumidoras dependem diretamente da interface de *cache* (CacheGateway), além do próprio *adapter* substituído e do ponto único de vinculação de dependências no *AppServiceProvider*. Em uma arquitetura sem essa camada de abstração na qual casos de uso invocam diretamente uma biblioteca ou serviço de *cache* específico, cada um desses pontos de consumo precisaria ser individualmente identificado, reescrito e reteste, sem um ponto único de substituição. A concentração da mudança em um único arquivo de implementação e uma única linha de vinculação de dependência, evidenciada empiricamente pelas 226 linhas do experimento realizado, demonstra de forma prática o efeito da inversão de dependência na redução do custo de mudança arquitetural.

Sob a perspectiva de engenharia de software, esse comportamento reduz consideravelmente o custo de mudança arquitetural da plataforma. Em arquiteturas fortemente acopladas, alterações envolvendo componentes centrais de infraestrutura normalmente exigem modificações distribuídas em múltiplas camadas do sistema, aumentando risco operacional, custo de manutenção e possibilidade de regressões. Na solução desenvolvida, entretanto, a substituição ficou concentrada apenas na camada de infraestrutura, preservando integralmente a lógica de negócio existente.

Além disso, esse experimento também demonstra vantagens relacionadas à portabilidade tecnológica e redução de *vendor lock-in*. A arquitetura permite substituir tecnologias específicas como Redis, sistemas de mensageria ou mecanismos de persistência sem necessidade de reescrita significativa do domínio da aplicação. Essa característica é especialmente relevante em sistemas distribuídos modernos, nos quais mudanças de infraestrutura podem ocorrer devido a custos operacionais, requisitos de escalabilidade ou restrições de ambiente.

A estratégia adotada também contribuiu diretamente para a testabilidade da aplicação. Como os casos de uso dependem de interfaces e não de implementações concretas, foi possível executar testes unitários utilizando *mocks* dos *gateways*, isolando completamente as regras de negócio da infraestrutura externa. Essa abordagem permitiu validar cenários complexos do domínio sem necessidade de acesso real a banco de dados, cache ou serviços externos.

Outro aspecto relevante observado foi a utilização de mecanismos de observabilidade e instrumentação nos casos de uso da aplicação. Os fluxos implementados realizam rastreamento de início, conclusão e falha de operações críticas, permitindo maior controle operacional e facilitando processos de monitoramento e depuração.

Além disso, a divisão da API em diferentes contextos externos, parceiros, administrativos e webhooks permitiu segmentar responsabilidades e estabelecer limites claros entre os diferentes perfis de utilização da plataforma. A Tabela 5 apresenta, de forma resumida, os principais grupos de rotas de cada um dos três contextos de API da aplicação.

Tabela 5 – Grupo de Rotas Por Contexto de API

Prefixo	Principais Recursos
/internal/v1	Países, estados, cidades, campanhas, promoções, usuários.
/external/v1	Autenticação, endereços, métodos de pagamento, carrinho, checkout, pedidos, avaliações, cotação de entrega.
/partner/v1	Empresas, lojas, catálogos, categorias, grupos de modificadores, produtos, promoções, pedidos, entregas, entregadores, veículos.

Portanto, os resultados demonstram que a arquitetura adotada foi capaz de fornecer organização estrutural, flexibilidade tecnológica, elevada capacidade de manutenção e facilidade de evolução, características fundamentais em sistemas distribuídos orientados a domínio e aplicações modernas de alta complexidade operacional.

4.3 Validação e Testes

A validação da aplicação foi realizada utilizando uma estratégia híbrida composta por testes unitários e testes de integração, permitindo validar tanto as regras isoladas do domínio quanto o comportamento completo da plataforma em cenários próximos ao ambiente real de execução.

O desenvolvimento contou com o auxílio de ferramentas de inteligência artificial baseadas em modelos de linguagem, especificamente o GitHub Copilot, utilizado em dois contextos: como *autocomplete* de código durante a implementação geral da aplicação, e de forma mais direta na geração de casos de teste unitários e de integração, seguindo os padrões arquiteturais já estabelecidos no projeto. Todo o código gerado com auxílio de IA foi revisado, validado e, quando necessário, ajustado manualmente pelo autor antes de sua incorporação ao projeto.

Ao todo, foram implementados 972 testes automatizados, totalizando 2500 *assertions* distribuídos entre diferentes módulos da aplicação. Desse total, 658 testes correspondem a testes unitários, responsáveis por validar casos de uso e regras de negócio de forma isolada por meio da utilização de *mocks* dos *gateways* e dependências externas. Os demais testes concentram-se em testes de integração responsáveis pela validação das rotas HTTP, workers assíncronos e rotinas agendadas (crons).

Essa abordagem permitiu validar tanto a consistência interna das regras de negócio quanto a comunicação entre os diferentes componentes da arquitetura, incluindo banco de dados, *cache* Redis, mensageria Kafka e fluxos assíncronos.

4.3.1 Distribuição dos Tipos de Testes

A Tabela 6 apresenta a distribuição geral dos testes implementados no projeto.

Tabela 6 – Distribuição Geral dos Testes Implementados

Tipo de Teste	Quantidade de testes	Assertions
Testes Unitários	658	1611
Testes de Integração	314	880
Total	975	2503

Fonte: Elaborado pelo autor (2026).

Os testes unitários foram desenvolvidos principalmente para validar os casos de uso presentes na camada *core* da aplicação. Nesses testes, os gateways responsáveis por acesso a banco de dados, *cache*, APIs externas e mensageria foram substituídos por *mocks*, permitindo validar exclusivamente a lógica de negócio implementada nos casos de uso e regras do domínio.

Essa estratégia possibilitou validar cenários complexos relacionados a promoções, cálculo de entrega, atualização de pedidos, pagamentos, gerenciamento de estoque e despacho de entregadores sem dependência direta da infraestrutura externa.

Já os testes de integração foram utilizados para validar o comportamento completo da aplicação, incluindo execução de rotas HTTP, persistência em banco de dados, consumo de mensagens Kafka e execução de comandos agendados. Essa abordagem permitiu verificar a correta integração entre *controllers*, *factories*, casos de uso, adaptadores e mecanismos de infraestrutura.

4.3.2 Escopo dos Testes de Integração

Os testes de integração contemplaram diferentes partes da arquitetura da aplicação. A Tabela 7 apresenta os principais componentes validados.

Tabela 7 – Componentes cobertos pelos testes de integração

Componente	Objetivo da validação
Rotas HTTP	Cadastro e consulta de usuários, endereços, pedidos, entregas e avaliações
Workers Kafka	Processamento de criação de pedidos, atualização de status, pagamentos, estoque, despacho de entregadores e avaliações
Crons/Scheduler	Cancelamento automático de pedidos, processamento de avaliações e disparo de tentativas de despacho
Banco de Dados	Persistência, atualização e recuperação de entidades do domínio
Fluxos Geográficos	Cálculo de distância utilizando H3 e busca por proximidade
Processamento de Pagamentos	Recebimento e atualização de eventos de pagamento
Fluxo de Delivery	Seleção de entregadores, controle de ofertas, retries e atualização de status

Fonte: Elaborado pelo autor (2026).

Os testes relacionados aos *workers* foram especialmente importantes devido à natureza assíncrona da aplicação. Foram executados cenários envolvendo publicação e consumo de mensagens Kafka relacionadas à criação de pedidos, atualização de estoque, processamento de pagamentos, geração de lançamentos financeiros e despacho de entregadores.

No fluxo de criação de pedidos, os testes verificaram a correta recepção dos eventos publicados, a persistência das informações no banco de dados e a execução das regras de negócio associadas ao pedido. Nos testes de atualização de estoque, foram validados os mecanismos de redução e atualização das quantidades disponíveis após o processamento dos pedidos. Já nos fluxos de pagamento, foram simulados eventos de confirmação e falha, verificando a atualização adequada dos estados financeiros e dos pedidos relacionados.

No fluxo de *dispatch* de entregas, os testes validaram mecanismos de *retry* automático, controle de expiração de ofertas, múltiplas tentativas de distribuição e prevenção de concorrência entre entregadores. Foram avaliados cenários em que um entregador aceita a oferta, rejeita a oferta ou não responde dentro do tempo configurado, exigindo o encaminhamento automático para novos candidatos. Também foram testadas situações em que não existem entregadores disponíveis na região de atendimento.

Já nos fluxos de cotação de entrega, foram testados cálculos envolvendo distância geográfica baseada em H3, multiplicadores regionais, taxa de serviço e lógica de preço dinâmico baseada na relação entre oferta e demanda de entregadores. Os testes verificaram a consistência dos valores calculados para diferentes distâncias, regiões e níveis de disponibilidade de entregadores.

A execução dos testes permitiu identificar e corrigir inconsistências relacionadas principalmente às regras de transição de status, ao processamento de eventos assíncronos e ao controle de tentativas de despacho de entregadores. Após os ajustes realizados durante o desenvolvimento, todos os cenários previstos passaram a apresentar comportamento consistente com os requisitos definidos para a plataforma, não sendo observadas falhas críticas nos fluxos considerados essenciais para o funcionamento do sistema.

4.3.3 Cobertura de Código

Para avaliar a efetividade da estratégia de testes, foram coletadas métricas de cobertura de código utilizando relatórios automatizados da aplicação. Os resultados gerais são apresentados na Tabela 8.

Tabela 8 – Cobertura geral da aplicação

Métrica	Cobertura (%)	Elementos cobertos
Linhas	93,91%	8502/9053

Funções e Métodos	89,67%	1258/1403
Classes e Traits	85,56%	634/741

Fonte: Elaborado pelo autor (2026).

Os resultados demonstram elevada abrangência da estratégia de testes implementada. A cobertura superior a 90% em linhas indica que praticamente todos os fluxos principais da aplicação foram exercidos durante os testes automatizados. Destaca-se, ainda, que classes previamente identificadas com maior complexidade ciclomática combinada a cobertura incompleta relacionadas à validação de transições de status de entrega, à validação de dados de cadastro e atualização de usuário, e à validação de modificadores de produto no fluxo de criação de pedidos foram objeto de testes unitários adicionais elaborados especificamente para exercitar os ramos condicionais anteriormente não cobertos, elevando a cobertura dessas classes de uma faixa entre 33% e 49% para uma faixa entre 94% e 100%.

Além da análise global, o relatório de cobertura também permitiu avaliar separadamente as principais estruturas da aplicação, especialmente as pastas core e app, que representam respectivamente a camada de domínio e a camada de infraestrutura/adaptação.

4.3.4 Análise de Cobertura por Camada Arquitetural

A camada *core* concentra os casos de uso, regras de negócio e contratos do domínio, enquanto a camada *app* contém *controllers,* HTTP, *factories*, *adapters*, *Eloquent*, *consumers,* Kafka e componentes de infraestrutura.

A Tabela 9 apresenta uma análise qualitativa da cobertura entre essas camadas.

Tabela 9 – Análise de cobertura por camada arquitetural

Camada	Responsabilidade principal	Nível de cobertura observado
Core	Regras de negócio e casos de uso	Muito elevado
App	Infraestrutura e integração externa	Elevado
Workers/Consumers	Processamento assíncrono	Elevado
Adapters	Integração com banco e serviços externos	Elevado
DTOs e Collections	Transporte de dados	Moderado

Fonte: Elaborado pelo autor (2026).

Observou-se que a camada *core* apresentou os maiores níveis de cobertura, principalmente devido à forte presença de testes unitários executando casos de uso com *mocks* dos *gateways*. Essa abordagem foi importante para garantir elevada confiabilidade das regras centrais do domínio sem dependência da infraestrutura externa.

Já a camada *app* apresentou cobertura elevada principalmente por meio dos testes de integração, responsáveis por validar controladores, consumers Kafka, *factories*, *middlewares* e integração entre os componentes da aplicação.

Os componentes com menor cobertura concentram-se majoritariamente em *DTOs*, coleções e estruturas simples de resposta, que possuem baixa complexidade lógica e impacto reduzido sobre as regras centrais do sistema.

4.3.5 Análise da Distribuição das Coberturas

O gráfico da Figura 13 evidencia a distribuição percentual da cobertura entre as classes da aplicação.

A análise do gráfico demonstra que a maior parte dos componentes críticos da aplicação encontra-se concentrada nas faixas mais altas de cobertura, especialmente acima de 80%. Isso indica que os módulos responsáveis pelas

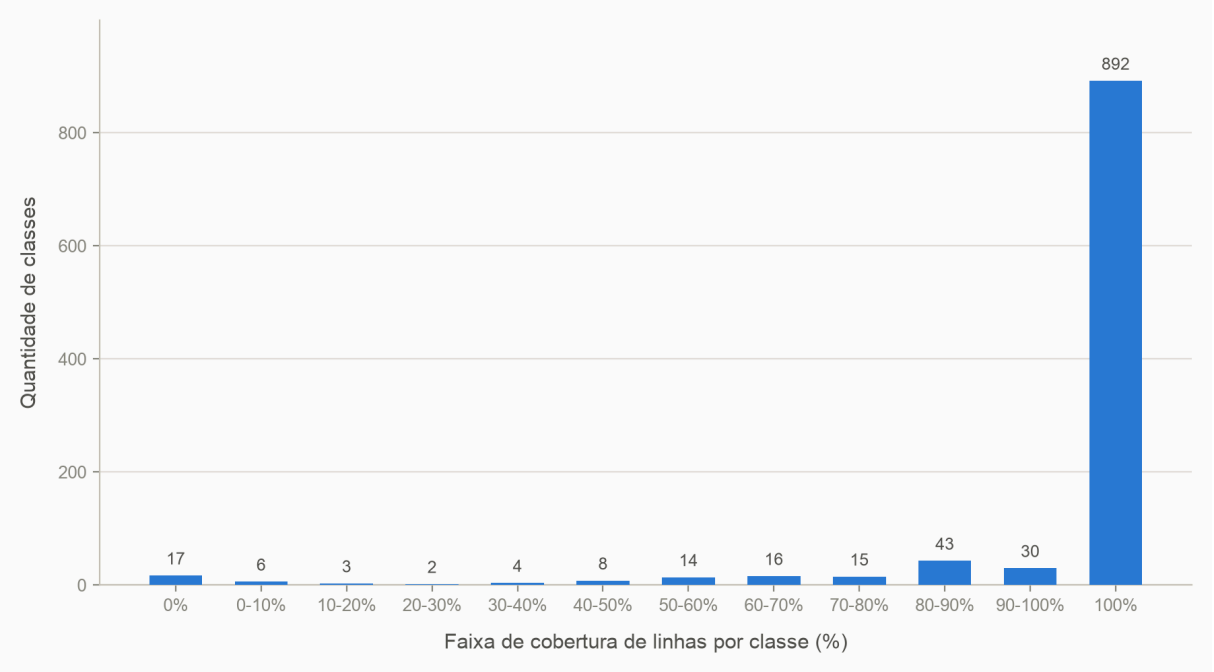
principais regras de negócio receberam maior prioridade durante o processo de validação.

Entre os módulos com elevada cobertura destacam-se componentes relacionados a: cálculo de entrega, atualização de status de pedidos, processamento de pagamentos, gerenciamento de estoque, atualização de avaliações, validação de promoções, despacho de entregadores.

Os componentes que apresentaram baixa cobertura estão majoritariamente relacionados a estruturas auxiliares, como DTOs, coleções de objetos e respostas serializadas, que possuem pouca lógica interna e baixo risco operacional.

Esse comportamento demonstra que a estratégia de testes foi orientada por criticidade de negócio, priorizando os componentes com maior impacto funcional sobre a plataforma.

Figura 13 – Porcentagem da Cobertura dos Testes



Fonte: Elaborado pelo autor (2026).

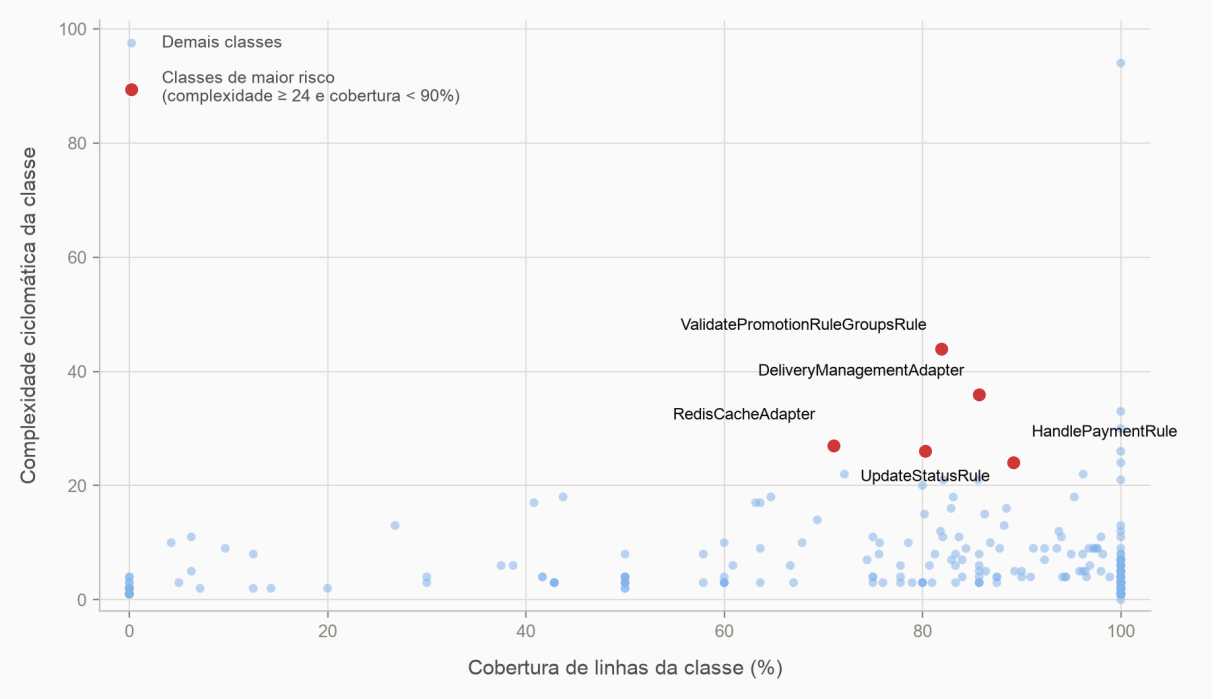
4.3.6 Análise de Complexidade e Cobertura de Testes

A Figura 14 apresenta a análise dos principais componentes da aplicação utilizando a métrica CRAP (Change Risk Anti-Patterns). Essa métrica combina a complexidade ciclomática do código com a cobertura de testes automatizados, permitindo identificar componentes que apresentam maior risco de manutenção e maior probabilidade de introdução de falhas durante futuras alterações.

O cálculo do CRAP Score considera simultaneamente a quantidade de caminhos de execução existentes em um método e o percentual de cobertura de testes associado a ele. Dessa forma, componentes complexos e pouco testados tendem a apresentar valores mais elevados, enquanto componentes complexos, mas adequadamente cobertos por testes, possuem seu risco reduzido.

Os módulos com maior complexidade concentram-se principalmente em fluxos de negócio com múltiplas validações condicionais e regras dinâmicas, como atualização de status de entregas, validação de promoções e processamento de pagamentos.

Figura 14 – Gráfico da Complexidade da aplicação



Fonte: Elaborado pelo autor (2026).

A elevada complexidade desses componentes está relacionada à quantidade de cenários de negócio suportados pela aplicação. O fluxo de atualização de status de entregas, por exemplo, precisa lidar com múltiplas transições válidas, estados intermediários, controle de *retry* e sincronização entre entregadores e pedidos.

Da mesma forma, o sistema de promoções implementa regras condicionais relacionadas a categorias, produtos, métodos de pagamento, horários específicos, pedidos mínimos e campanhas promocionais, aumentando naturalmente a complexidade estrutural dos módulos responsáveis por validação.

Verificou-se, a partir da análise dos dados de cobertura, que classes de alta complexidade nem sempre correspondiam a uma cobertura de testes proporcionalmente elevada. A título de exemplo, a regra de validação de transição de status de entregas, com complexidade ciclomática significativamente acima da média do projeto, apresentava inicialmente cobertura de aproximadamente 49%, com cinco de suas sete rotinas internas de validação não exercitadas por nenhum

teste. Esse padrão foi identificado e corrigido durante o processo de validação contínua do projeto: foram elaborados testes unitários dedicados para essa e outras três classes de alto risco (combinação de complexidade elevada e cobertura incompleta), elevando sua cobertura para a faixa entre 94% e 100%, sem alteração no comportamento das regras de negócio validadas a suíte completa de testes, após essa adição, permaneceu integralmente aprovada.

Apesar disso, os resultados demonstram que mesmo os componentes mais complexos mantiveram níveis relevantes de cobertura de testes, reduzindo significativamente o risco operacional associado à manutenção dessas regras.

4.3.6 Robustez da Solução

Os resultados obtidos durante o processo de validação demonstram que a arquitetura implementada apresentou comportamento consistente tanto em operações síncronas quanto assíncronas.

A utilização de Kafka mostrou-se adequada para o desacoplamento de tarefas críticas, enquanto o Redis apresentou bom desempenho no gerenciamento de estados temporários relacionados a dispatch de entregadores, controle de retries e sincronização de eventos concorrentes.

Além disso, os testes envolvendo geolocalização baseada em H3 demonstraram capacidade de representar cenários reais de proximidade geográfica e otimização logística, reforçando a aderência da solução ao domínio de plataformas de *delivery*.

Dessa forma, os resultados indicam que a estratégia de testes adotada foi capaz de fornecer elevado grau de confiabilidade, robustez e previsibilidade operacional para a solução desenvolvida.

5 CONSIDERAÇÕES

Os resultados obtidos ao longo do desenvolvimento permitiram alcançar as seguintes contribuições:

a) Desenvolvimento de uma plataforma backend completa para um sistema de *delivery*, contemplando fluxos de pedidos, pagamentos, logística de entrega e avaliações.
b) Aplicação prática dos princípios da Arquitetura Hexagonal, promovendo desacoplamento entre domínio e infraestrutura.
c) Implementação de processamento assíncrono baseado em eventos utilizando Apache Kafka.
d) Utilização de mecanismos de cache distribuído para otimização de desempenho e gerenciamento de estado.
e) Integração de técnicas de geolocalização baseadas em H3 para suporte à cotação e ao despacho de entregas.
f) Implementação de mecanismos de distribuição automática de entregadores com controle de tentativas e *retries*.
g) desenvolvimento de uma estratégia abrangente de validação composta por 972 testes automatizados e 2500 *assertions*.
h) Obtenção de elevados índices de cobertura de testes, alcançando 93,91% de cobertura de linhas, 89,67% de cobertura de métodos e 85,56% de cobertura de classes;
i) Demonstração da viabilidade da integração de conceitos de Domain-Driven Design, Arquitetura Orientada a Eventos, cache distribuído e geolocalização em uma única solução funcional.

O presente trabalho teve como objetivo o desenvolvimento de uma plataforma backend para um sistema de *delivery* fundamentada em princípios modernos de arquitetura de *software*, sistemas distribuídos e separação de responsabilidades. Ao longo do desenvolvimento, buscou-se não apenas implementar funcionalidades operacionais, mas estruturar a solução de forma coerente com práticas consolidadas

da engenharia de *software*, priorizando escalabilidade, flexibilidade arquitetural, desacoplamento e manutenção.

Os resultados obtidos demonstram que a arquitetura proposta foi capaz de representar adequadamente a complexidade do domínio de plataformas de *delivery*, contemplando fluxos relacionados a gerenciamento de pedidos, pagamentos, despacho de entregadores, processamento assíncrono de eventos, cálculo de entrega e atualização de avaliações de estabelecimentos.

A arquitetura adotada, baseada em um monólito modular com forte inspiração na Arquitetura Hexagonal e *Clean Architecture*, mostrou-se adequada para promover isolamento entre regras de negócio e componentes de infraestrutura. Essa abordagem permitiu que o domínio da aplicação permanecesse desacoplado de tecnologias específicas, favorecendo evolução futura e redução do impacto de mudanças arquitetônicas.

Um resultado relevante observado durante o desenvolvimento foi a validação prática da flexibilidade arquitetural da solução. Como experimento de substituição tecnológica, o mecanismo de *cache* Redis foi substituído por uma implementação baseada em cache em memória, exigindo apenas a criação de um novo *adapter* e pequenas alterações no mecanismo de injeção de dependência do Laravel. Toda a substituição foi realizada com aproximadamente 226 linhas de código, sem necessidade de alterações nos casos de uso ou regras centrais do domínio. Esse resultado evidencia a efetividade da utilização de contratos abstratos e princípios de inversão de dependência na redução de acoplamento entre domínio e infraestrutura.

A incorporação de conceitos de arquitetura orientada a eventos e processamento assíncrono também demonstrou ser adequada para lidar com operações desacopladas e fluxos concorrentes. A utilização de Kafka para comunicação assíncrona permitiu separar operações críticas do fluxo HTTP principal, reduzindo dependências diretas entre componentes e favorecendo maior capacidade de escalabilidade da aplicação.

Outro aspecto relevante foi a utilização de técnicas de geolocalização baseadas em índices H3 para suporte à lógica de despacho de entregadores e

cálculo de cotação de entrega. Essa integração demonstra preocupação em alinhar a solução técnica com problemas reais do domínio logístico de plataformas de *delivery*, aproximando o sistema de cenários encontrados em aplicações utilizadas no mercado.

Do ponto de vista acadêmico, o trabalho contribui ao demonstrar a aplicação integrada de múltiplos conceitos teóricos em um único sistema funcional, incluindo arquitetura hexagonal, mensageria assíncrona, *cache* distribuído, modelagem orientada a domínio, separação de responsabilidades e geolocalização aplicada a sistemas distribuídos.

Além da implementação arquitetural, os resultados do processo de validação também reforçam a consistência da solução proposta. Foram desenvolvidos 972 testes automatizados, totalizando 2500 *assertions*, contemplando testes unitários e testes de integração. Desse total, 658 testes unitários foram utilizados para validação isolada dos casos de uso da camada de domínio por meio de *mocks* dos *gateways* de infraestrutura, enquanto os demais 314 testes de integração validaram rotas HTTP, *workers* Kafka, rotinas agendadas e persistência de dados.

Os resultados de cobertura de código também demonstraram ampla validação dos fluxos implementados, alcançando 93,91% de cobertura de linhas, 89,67% de cobertura de métodos e 85,56% de cobertura de classes na camada de domínio. A análise do relatório de cobertura evidenciou que os módulos críticos da aplicação, especialmente aqueles relacionados a pagamentos, promoções, atualização de status e despacho de entregadores, mantiveram níveis elevados de cobertura mesmo apresentando maior complexidade estrutural; adicionalmente, classes que ainda apresentavam cobertura incompleta associada a complexidade elevada foram identificadas e corrigidas por meio da adição de testes unitários dedicados, conforme detalhado na Seção 4.3.6.

Dessa forma, os resultados apresentados neste trabalho não se referem a métricas quantitativas de desempenho da aplicação, como *throughput*, latência ou escalabilidade horizontal sob carga elevada, mas sim à análise arquitetural da

solução implementada e à validação funcional realizada por meio da execução automatizada dos testes desenvolvidos.

Como limitações da metodologia adotada, destaca-se a ausência de avaliações quantitativas aprofundadas de desempenho, como testes de carga, estresse e concorrência em ambiente distribuído. O ambiente utilizado para desenvolvimento e execução dos testes permaneceu restrito à execução local, impossibilitando medições mais avançadas relacionadas à escalabilidade horizontal da solução.

Entretanto, é importante destacar que o objetivo principal deste trabalho não foi realizar *benchmarking* de desempenho, mas sim propor, implementar e validar uma arquitetura *backend* escalável fundamentada em padrões modernos de engenharia de *software*. Dessa forma, a ausência de medições quantitativas em larga escala não compromete a validade arquitetural da proposta apresentada.

Outra limitação relevante refere-se à complexidade inerente à própria arquitetura adotada. A utilização de múltiplos componentes, como mensageria assíncrona, *cache* distribuído, *workers* independentes e processamento orientado a eventos, aumenta o nível de complexidade operacional da aplicação, exigindo maior esforço de desenvolvimento, configuração e manutenção.

Além disso, a dependência de serviços externos, como provedores de pagamento e APIs de geolocalização, pode impactar o funcionamento da aplicação em cenários de indisponibilidade. Embora a arquitetura implementada favoreça desacoplamento e flexibilidade, mecanismos avançados de resiliência como *circuit breaker*, *retry policies* distribuídas e *fallback* automatizado não foram explorados de forma aprofundada neste trabalho.

Da mesma forma, apesar da ampla cobertura de testes automatizados e da validação funcional dos fluxos da aplicação, o sistema ainda não contempla mecanismos avançados de observabilidade distribuída, como tracing distribuído, métricas centralizadas e monitoramento em tempo real de eventos assíncronos, o que representa uma oportunidade importante para evolução futura da plataforma.

Como trabalhos futuros, destacam-se:

a) Realização de testes de carga e estresse em ambiente distribuído.

b) Evolução da arquitetura para um modelo baseado em microsserviços: a organização atual em módulos de domínio fracamente acoplados, cada um com seus próprios casos de uso, *gateways* e regras, comunicando-se predominantemente por eventos assíncronos via Kafka, torna a fronteira de um eventual microsserviço relativamente clara o domínio de despacho de entregas, por exemplo, já opera de forma quase isolada, comunicando-se com o domínio de pedidos apenas por eventos e por uma única interface de *gateway*. Tal migração seria motivada principalmente por necessidades de escalonamento independente entre domínios com volumes de eventos distintos, ou pela atuação de equipes de desenvolvimento distintas por domínio cenários ainda não presentes no estágio atual do projeto, o que reforça a adequação da decisão pelo monólito modular, sem impedir essa evolução futura.

c) Implementação de observabilidade avançada e tracing distribuído.

d) Utilização de estratégias avançadas de resiliência.

e) Implementação em ambientes *cloud-native* com orquestração de containers.

f) Implementação de auto scaling para *workers* assíncronos.

Dessa forma, conclui-se que o sistema desenvolvido atende aos objetivos propostos, apresentando-se como uma solução tecnicamente viável, arquiteturalmente consistente e alinhada com práticas modernas da engenharia de software, além de constituir uma base sólida para futuras expansões, pesquisas e evoluções na área de sistemas distribuídos e plataformas de *delivery*.

REFERÊNCIAS

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software architecture in practice**. 3. ed. Boston: Addison-Wesley, 2012.

BRODSKY, Issac. **H3:** Uber's Hexagonal Hierarchical Spatial Index. Uber Engineering, 2018. Disponível em: https://www.uber.com/br/en/blog/h3/. Acesso em: 17 abr. 2026.

EVANS, Eric. **Domain-Driven Design:** Tackling Complexity in the Heart of Software. Boston: Addison-Wesley, 2003.

FIELDING, Roy Thomas. **Architectural styles and the design of network-based software architectures**. Irvine: University of California, 2000.

FOWLER, Martin. **Microservices:** a definition of this new architectural term. Disponível em: https://martinfowler.com. Acesso em: 17 abr. 2026.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Boston: Addison-Wesley, 2003.

GITHUB. **marcozsouto/foods**. Disponível em: https://github.com/marcozsouto/foods. Acesso em: 12 maio 2026.

KLEPPMANN, Martin. **Designing Data-Intensive Applications**. Sebastopol: O'Reilly Media, 2017.

LARAVEL. **Laravel Octane**. Disponível em: https://laravel.com/docs/octane. Acesso em: 12 maio 2026.

NEWMAN, Sam. **Building Microservices**. Sebastopol: O'Reilly Media, 2015.

OPENSTREETMAP FOUNDATION. **OpenStreetMap**. Disponível em: https://www.openstreetmap.org/. Acesso em: 12 maio 2026.

RICHARDSON, Chris. **Microservices Patterns**. Shelter Island: Manning Publications, 2018.

ROBUSTO, C. C. **The cosine-haversine formula**. The American Mathematical Monthly, v. 64, n. 1, p. 38–40, 1957.

SWOOLE LABS. **Swoole**: Event-driven asynchronous and concurrent networking engine for PHP. Disponível em: https://openswoole.com/. Acesso em: 12 maio 2026.

VERNON, Vaughn. **Implementing Domain-Driven Design**. Boston: Addison-Wesley, 2013.