FACULDADE DE TECNOLOGIA DE AMERICANA "Ministro Ralph Biasi"
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Bruno Fernandes Souza

Maria Eduarda Amadeu

SERVE AÍ

Plataforma *multi-tenant* para autoatendimento e gestão no setor gastronômico.

Americana, SP

2025

FACULDADE DE TECNOLOGIA DE AMERICANA "Ministro Ralph Biasi"
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Bruno Fernandes Souza

Maria Eduarda Amadeu

SERVE AÍ

Plataforma *multi-tenant* para autoatendimento e gestão no setor gastronômico.

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas sob a orientação do Prof. José William Pinto Gomes.

Área de concentração: Análise e Desenvolvimento de Sistemas.

Americana, SP

2025

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana Ministro Ralph Biasi- CEETEPS Dados Internacionais de Catalogação-na-fonte

AMADEU, Maria Eduarda

Serve Aí: Plataforma multi-tenant para autoatendimento e gestão no setor gastronômico. / Maria Eduarda AMADEU, Bruno Fernandes SOUZA – Americana, 2025.

81f.

Monografia (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) - - Faculdade de Tecnologia de Americana Ministro Ralph Biasi – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Esp. José William Pinto GOMES

1. Arquitetura de sistemas 2. Dispositivos móveis - aplicativos 3. Engenharia de software. I. AMADEU, Maria Eduarda, II. SOUZA, Bruno Fernandes III. GOMES, José William Pinto IV. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana Ministro Ralph Biasi

CDU: 681516
681519
681.3.05

Elaborada pelo autor por meio de sistema automático gerador de ficha catalográfica da Fatec de Americana Ministro Ralph Biasi.

Bruno Fernandes Souza

Maria Eduarda Amadeu

SERVE AI: Plataforma multi-tenant para autoatendimento e gestão no setor gastronômico

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana Ministro Ralph Biasi.
Área de concentração: Análise e Desenvolvimento de Sistemas.

Americana, 1 de dezembro de 2025.

Banca Examinadora:

José William Pinto Gomes
Especialista
Fatec Americana "Ministro Ralph Biasi"

Rogério Nunes de Freitas
Mestre
Fatec Americana "Ministro Ralph Biasi"

Cíntia Gimenez da Cunha
Especialista
Fatec Americana "Ministro Ralph Biasi"

RESUMO

Este trabalho documenta o desenvolvimento da plataforma web Serve Aí, solução que integra autoatendimento digital e gestão operacional para estabelecimentos gastronômicos. O sistema utiliza arquitetura *multi-tenant*, permitindo que diferentes estabelecimentos compartilhem a mesma aplicação mantendo isolamento completo de dados, seguindo a taxonomia de Chong e Carraro (2006). A base técnica compreende dois módulos desenvolvidos com *Server-Side Rendering* (SSR), *React Server Components* e *TypeScript*: interface de autoatendimento para clientes finais e painel administrativo para gestores. O desenvolvimento aplicou padrões MVC e *Repository Pattern*, com segurança orientada pelas diretrizes OWASP. O módulo cliente utiliza Teoria da Carga Cognitiva (Sweller, 1988) e *Progressive Disclosure* para otimizar conversão. O módulo administrativo oferece gestão de pedidos em tempo real, histórico paginado e integração com Power BI. A validação técnica demonstrou que a combinação de renderização no servidor com isolamento de *tenants* atende requisitos divergentes de simplicidade (consumidores) e densidade informacional (gestores), viabilizando sistemas com alta performance e escalabilidade. A arquitetura permite expansão futura com assistentes virtuais e sistemas de recomendação baseados em inteligência artificial.

Palavras-chave: Arquitetura *Multi-Tenant*; *Server-Side Rendering*; Autoatendimento Digital; Gestão Operacional; Engenharia de Software.

ABSTRACT

This work presents the development of a multi-tenant web platform named Serve Aí, designed to provide an integrated solution for self-service and operational management in food service establishments. The project is grounded in the application of modern software architectures, combining Server-Side Rendering via React Server Components with TypeScript to build two functionally segregated modules: a self-service module for end consumers and an administrative panel for operational management. The implemented multi-tenant architecture allows multiple establishments to share the same application instance while maintaining rigorous logical data isolation, following Chong and Carraro's taxonomy (2006). The methodology was based on component-oriented development principles, applying established design patterns such as MVC and Repository Pattern, alongside OWASP security guidelines to protect against access control vulnerabilities. The client module implements conversion optimization strategies grounded in Sweller's Cognitive Load Theory (1988) and Progressive Disclosure principles, prioritizing reduced Time to Interactive through server-side rendering. The administrative module incorporates real-time order management functionalities, server-side paginated operation history, digital menu management, and integration with analytical dashboards via Power BI. Technical validation demonstrated the viability of the proposed dual architecture, simultaneously meeting divergent requirements of simplicity for consumers and informational density for managers. Results confirm that the application of Server-Side Rendering with multi-tenant isolation constitutes an effective solution for systems demanding high performance, scalability, and clear separation of responsibilities, establishing an extensible architectural foundation for future functionalities such as virtual assistants and artificial intelligence-based recommendation systems.

Keywords: *Multi-Tenant Architecture; Server-Side Rendering; Digital Self-Service; Operational Management; Software Engineering.*

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

3NF: *Third Normal Form* (Terceira Forma Normal);
ABF: Associação Brasileira de Franchising;
AJAX: *Asynchronous JavaScript and* XML;
API: *Application Programming Interface* (Interface de Programação de Aplicações);
CPF: Cadastro de Pessoas Físicas;
CRUD: *Create, Read, Update, Delete* (Criar, Ler, Atualizar, Deletar);
CSRF: *Cross-Site Request Forgery* (Falsificação de Requisição Entre Sites);
CSS: *Cascading Style Sheets* (Folhas de Estilo em Cascata);
DER: Diagrama Entidade-Relacionamento;
FCP: *First Contentful Paint* (Primeira Renderização de Conteúdo);
HSTS: HTTP *Strict Transport Security* (Segurança Estrita de Transporte HTTP);
HTML: *HyperText Markup Language* (Linguagem de Marcação de Hipertexto);
HTTP: *HyperText Transfer Protocol* (Protocolo de Transferência de Hipertexto);
HTTPS: *HyperText Transfer Protocol Secure* (Protocolo de Transferência de Hipertexto Seguro);
ISR: *Incremental Static Regeneration* (Regeneração Estática Incremental);
JSON: *JavaScript Object Notation* (Notação de Objetos *JavaScript*);
JWT: JSON *Web Token*;
LGPD: Lei Geral de Proteção de Dados;
MPA: *Multi-Page Application* (Aplicação de Múltiplas Páginas);
MVC: *Model-View-Controller* (Modelo-Visão-Controlador);
OCP: *Open/Closed Principle* (Princípio Aberto/Fechado);
ORM: *Object-Relational Mapping* (Mapeamento Objeto-Relacional);
OWASP: *Open Web Application Security Project* (Projeto Aberto de Segurança em Aplicações Web);
PDV: Ponto de Venda;
REST: *Representational State Transfer* (Transferência de Estado Representacional);
SaaS: *Software as a Service* (Software como Serviço);
SEO: *Search Engine Optimization* (Otimização para Mecanismos de Busca);

SOLID: *Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion* (Responsabilidade Única, Aberto/Fechado, Substituição de Liskov, Segregação de Interface, Inversão de Dependência);
SPA: *Single-Page Application* (Aplicação de Página Única);
SQL: *Structured Query Language* (Linguagem de Consulta Estruturada);
SRP: *Single Responsibility Principle* (Princípio da Responsabilidade Única);
SSR: *Server-Side Rendering* (Renderização no Lado do Servidor);
TTI: *Time to Interactive* (Tempo até Interatividade);
UI: *User Interface* (Interface do Usuário);
UML: *Unified Modeling Language* (Linguagem de Modelagem Unificada);
URL: *Uniform Resource Locator* (Localizador Uniforme de Recursos);
UUID: *Universally Unique Identifier* (Identificador Único Universal);
XML: *eXtensible Markup Language* (Linguagem de Marcação Extensível).

SUMÁRIO

INTRODUÇÃO

A transformação digital redefiniu a relação entre empresas e consumidores no setor de serviços. Porter e Heppelmann (2014) argumentam que a integração de tecnologias digitais aos processos de negócio deixou de ser diferencial competitivo para tornar-se requisito fundamental de operação. O setor gastronômico exemplifica essa tendência através da crescente adoção de soluções tecnológicas para atendimento ao cliente e gestão operacional.

A engenharia de software contemporânea enfrenta, segundo Pressman e Maxim (2021), o desafio de conceber sistemas que atendam simultaneamente perfis de usuário com necessidades divergentes, mantendo performance, segurança e manutenibilidade. No contexto gastronômico, essa dualidade manifesta-se de forma acentuada: consumidores finais demandam interfaces simples e responsivas para realizar pedidos rapidamente, enquanto gestores necessitam de ferramentas analíticas com alta densidade informacional para tomada de decisões operacionais.

Este projeto identifica uma lacuna significativa no mercado brasileiro: estabelecimentos gastronômicos de pequeno e médio porte frequentemente operam com ferramentas desconexas – aplicativos de delivery de terceiros, sistemas de ponto de venda tradicionais, planilhas para controle financeiro. Essa fragmentação tecnológica dispersa dados em silos isolados, multiplica processos manuais e eleva custos operacionais sem agregar valor proporcional ao negócio.

A proposta deste trabalho consiste em desenvolver uma plataforma web que integre autoatendimento digital e gestão operacional em solução única, utilizando arquitetura *multi-tenant*. Essa abordagem permite que diferentes estabelecimentos compartilhem a mesma infraestrutura aplicacional mantendo isolamento completo de dados, viabilizando modelo de negócio *Software as a Service* (SaaS) com custos compartilhados.

A viabilidade técnica fundamenta-se em tecnologias consolidadas do ecossistema *JavaScript/TypeScript*. *Next.js* fornece capacidades de renderização *server-side*; *TypeScript* adiciona sistema de tipos estático para detecção precoce de erros; Prisma ORM abstrai operações de banco de dados com *type safety*. A combinação dessas tecnologias com arquitetura *multi-tenant* torna a solução

economicamente acessível para estabelecimentos que não podem investir em infraestrutura dedicada.

Diante deste cenário, o presente trabalho propõe-se a responder à seguinte questão norteadora:

Como projetar e implementar uma arquitetura *multi-tenant* que atenda simultaneamente aos requisitos divergentes de consumidores finais (alta conversão, baixa complexidade) e gestores operacionais (densidade informacional, ferramentas analíticas), garantindo escalabilidade, segurança e manutenibilidade?

O objetivo geral deste trabalho consiste em desenvolver uma plataforma web *multi-tenant*, nomeada Serve AÍ, que forneça solução integrada de autoatendimento para clientes e gestão operacional para estabelecimentos gastronômicos.

A hipótese que orienta este trabalho postula que a aplicação de *Server-Side Rendering* com *React Server Components*, aliada a arquitetura *multi-tenant* com isolamento lógico de dados e segmentação funcional clara entre módulo cliente e módulo administrativo, permite criar plataforma web que oferece experiência de usuário otimizada – caracterizada por baixo *Time to Interactive* e alta taxa de conversão – para consumidores finais e, simultaneamente, ferramenta de gestão robusta e densa em informações para operadores, de forma escalável e segura.

O trabalho organiza-se em três capítulos, além desta introdução e das considerações finais, o Capítulo 1 apresenta a fundamentação teórica, revisando conceitos de arquitetura *multi-tenant*, *Server-Side Rendering*, *Progressive Enhancement*, padrões de projeto orientados a objetos e diretrizes de segurança para aplicações web. Inclui análise comparativa de soluções de mercado existentes, estabelecendo conexões entre teoria consolidada – Teoria da Carga Cognitiva de Sweller (1988), princípios SOLID de Martin (2008) e diretrizes OWASP Top 10 (2021) – e as escolhas práticas de implementação adotadas no projeto, O Capítulo 2 detalha a metodologia aplicada, caracterizando a pesquisa como aplicada e experimental. Descreve as cinco fases do desenvolvimento iterativo e incremental: levantamento de requisitos e modelagem conceitual; prototipação e definição de arquitetura; implementação do módulo cliente; implementação do módulo administrativo; integração com sistema de pagamentos e validação de segurança. Apresenta justificativas técnicas para escolhas tecnológicas, conectando decisões de implementação com requisitos funcionais e não funcionais e o Capítulo 3 documenta o desenvolvimento prático e resultados obtidos. Apresenta arquitetura implementada

em três camadas com isolamento *multi-tenant* transversal; modelagem de banco de dados em Terceira Forma Normal (3NF) com sete entidades principais; decisões de design de interface seguindo princípios *mobile-first* para módulo cliente e densidade informacional para módulo administrativo. Inclui exemplos de código dos componentes críticos, validação técnica da arquitetura proposta e análise crítica do cumprimento dos objetivos estabelecidos, identificando limitações e oportunidades de evolução futura.

1 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos teóricos que orientaram as escolhas arquiteturais e tecnológicas do Serve Aí. Busca-se aqui estabelecer conexões entre a teoria consolidada de engenharia de software e sua aplicação no desenvolvimento de aplicações web contemporâneas. A discussão organiza-se em torno de cinco temas centrais: arquitetura *multi-tenant*, renderização *server-side*, design de interação, padrões de projeto orientados a objetos e segurança em aplicações web.

1.1 Arquitetura *Multi-Tenant.*

O conceito de *multi-tenancy* tornou-se essencial para aplicações SaaS. Segundo Guo et al. (2007), *multi-tenancy* refere-se à capacidade de uma única instância de software atender múltiplos clientes (*tenants*), mantendo isolamento lógico através de filtragem no código. Esta abordagem contrasta com arquiteturas *multi-instance*, onde cada cliente opera uma instância separada do software.

Chong e Carraro (2006) propuseram uma classificação em quatro níveis de maturidade para arquiteturas *multi-tenant*. No primeiro nível, cada cliente possui instâncias completamente segregadas. No segundo, compartilha-se a aplicação mas cada cliente mantém seu próprio banco de dados. O terceiro nível – adotado no Serve Aí – compartilha tanto aplicação quanto banco de dados, utilizando metadados configuráveis para diferenciação. Finalmente, o quarto nível incorpora escalabilidade automática via *load balancing*.

A adoção do modelo *multi-tenant* traz benefícios tangíveis: redução de custos operacionais pelo compartilhamento de recursos, facilidade de manutenção centralizada e agilidade no *deploy* de atualizações. Em contrapartida, demanda atenção redobrada quanto à segurança e ao isolamento de dados.

No contexto do projeto desenvolvido, cada estabelecimento gastronômico constitui um *tenant* distinto, identificado por um `slug` único na URL. Este identificador passa por validação sistemática em todas as operações críticas, seguindo o princípio de verificação de autorização da OWASP (2021). Tal medida visa prevenir vulnerabilidades de *Broken Access Control*, categoria que permanece no topo das vulnerabilidades mais críticas em aplicações web.

Para implementar o isolamento *multi-tenant*, o Serve Aí utiliza o Prisma ORM de modo que todas as consultas ao banco de dados incluam cláusulas `WHERE` filtradas por `restaurantId`. Esta estratégia, conhecida como *tenant isolation at query level*, funciona como salvaguarda contra vazamento de dados entre *tenants*, mesmo em cenários de erro de programação, constituindo uma camada adicional de segurança (*defense in depth*).

1.2 Server-Side Rendering e React Server Components

As aplicações web evoluíram através de diferentes paradigmas arquiteturais. Inicialmente, as *Multi-Page Applications* (MPA) dominavam o cenário: cada interação do usuário gerava uma requisição completa ao servidor, que retornava uma nova página HTML inteira, causando o recarregamento visível da interface (Garrett, 2005). Com o advento de AJAX, as *Single-Page Applications* (SPA) transferiram a renderização para o cliente, permitindo que *JavaScript* executasse no navegador e manipulasse dinamicamente o conteúdo sem necessidade de recarregar a página (Mesbah; Van Deursen, 2007).

Embora as SPAs ofereçam experiências mais fluidas após o carregamento inicial, elas apresentam limitações significativas. Osmani (2017) observa que aplicações *JavaScript-heavy* tendem a sofrer de TTI (*Time to Interactive*) elevado, especialmente em dispositivos com recursos limitados. Além disso, SPAs enfrentam desafios quanto ao SEO, uma vez que motores de busca têm dificuldade em indexar conteúdo gerado dinamicamente por *JavaScript*, como apontado por Schwartz (2018).

O SSR contemporâneo busca equilibrar esses *trade-offs*, preservando os benefícios das SPAs ao mesmo tempo em que mitiga suas limitações. Os *React Server Components*, introduzidos oficialmente pela Meta em 2023 (REACT TEAM, 2023), viabilizam a renderização seletiva de componentes no servidor, reduzindo drasticamente o volume de *JavaScript* enviado ao cliente. Esta abordagem alinha-se ao princípio de *Progressive Enhancement* proposto por Champeon (2003), que preconiza funcionalidade básica sem dependência de *JavaScript*, com melhorias graduais para clientes mais capazes.

Para o Serve Aí, a escolha por *Server Components* justifica-se pela natureza dos dados manipulados. Informações de pedidos e cardápios são dinâmicas e críticas, beneficiando-se da renderização no servidor onde o acesso direto ao banco de dados

é possível. A estratégia complementa-se com ISR (*Incremental Static Regeneration*), que permite cache com revalidação periódica, equilibrando performance e atualidade de dados.

O projeto adota tempos de revalidação diferenciados: 30 segundos para pedidos pendentes (priorizando atualidade para operações *near real-time*) e 300 segundos para o cardápio (considerando a frequência típica de alterações em menus digitais). Esta estratégia segue recomendações da Vercel (2023) para balanceamento entre *freshness* e carga no servidor, aplicando o conceito de *stale-while-revalidate* documentado na RFC 5861 (Nottingham, 2010): o conteúdo em cache é servido instantaneamente enquanto a atualização ocorre em background.

1.3 *Progressive Enhancement* e Teoria da Carga Cognitiva

Norman (2013) argumenta que o design de experiências digitais deve priorizar os modelos mentais do usuário, reduzindo fricção e facilitando a conclusão de tarefas. Em aplicações de pedidos digitais, cada ponto de atrito pode resultar em abandono de carrinho. Nielsen (1993) demonstrou que usuários formam sua opinião sobre websites em aproximadamente 50 milissegundos, o que torna a primeira impressão crucial para o engajamento.

A Teoria da Carga Cognitiva, proposta por Sweller (1988), sugere que interfaces devem minimizar o esforço mental não relacionado à tarefa principal. Para um cardápio digital, isso implica eliminar distrações como animações desnecessárias, pop-ups promocionais ou navegação complexa que dificulte a tarefa central: escolher produtos e finalizar o pedido.

O módulo cliente do Serve Aí materializa esses princípios de forma concreta. O fluxo de checkout foi intencionalmente simplificado para apenas duas etapas: navegar pelo cardápio e preencher dados básicos (nome e CPF). Versões iniciais do sistema contavam com três etapas, incluindo uma tela de revisão que se revelou redundante. Aplicando princípios da Teoria da Carga Cognitiva, optou-se por eliminar essa etapa intermediária, reduzindo a carga cognitiva extrínseca e permitindo que os usuários concentrem-se na seleção e finalização do pedido, em linha com as recomendações de Norman (2013) sobre redução de fricção.

A arquitetura de informação do cardápio digital baseia-se no conceito de *Information Scent*, desenvolvido por Pirolli e Card (1999). Pistas visuais – como

imagens de produtos, preços destacados e indicadores de ingredientes – permitem que os usuários avaliem rapidamente se um item atende suas necessidades, reduzindo o tempo de decisão e a carga cognitiva.

1.4 Princípios SOLID e Padrões de Projeto

Martin (2008), formalizou os princípios SOLID como alicerce para software manutenível. Entre eles, o *Single Responsibility Principle* (SRP) preconiza que cada módulo deve ter uma única razão para mudar. No Serve Aí, este princípio manifesta-se em nível arquitetural através da separação entre módulo cliente e módulo administrativo: alterações em funcionalidades de autoatendimento não afetam o painel de gestão, e vice-versa.

Já o *Open/Closed Principle* (OCP) estabelece que entidades de software devem estar abertas para extensão mas fechadas para modificação. A arquitetura de componentes do *React* facilita a aplicação desse princípio por meio de composição: novos tipos de visualização de produtos podem ser criados compondo componentes existentes, sem necessidade de modificar implementações base.

O *Repository Pattern*, documentado por Fowler (2002), desempenha papel importante ao abstrair a lógica de acesso a dados, desacoplando regras de negócio dos detalhes de persistência. No projeto, funções como `getOrdersByRestaurant()` e `getProductsByRestaurant()` encapsulam as queries Prisma, de modo que alterações na camada de dados não impactam os componentes de apresentação.

Quanto ao padrão MVC, introduzido por Reenskaug (1979) no contexto *Smalltalk* e posteriormente documentado por Gamma et al. (1994), ele organiza as responsabilidades em três camadas. No módulo administrativo do Serve Aí, os *React Server Components* atuam como *Views*, as *server actions* do *Next.js* implementam *Controllers*, e o Prisma *Client* representa a camada *Model*. Esta separação contribui para testabilidade e manutenibilidade do código.

1.5 Segurança em Aplicações Web

A OWASP documenta anualmente as dez vulnerabilidades mais críticas em aplicações web. Segundo o relatório OWASP Top 10 (2021), *Broken Access Control* permanece como a vulnerabilidade mais prevalente e crítica. Em arquiteturas *multi-*

tenant, a validação rigorosa de autorização torna-se ainda mais importante para prevenir acessos não autorizados a dados de outros *tenants* (fenômeno conhecido como *tenant hopping*).

O projeto implementa segurança em múltiplas camadas. A primeira camada utiliza autenticação robusta via JWT, com senhas armazenadas usando *hash bcrypt* configurado para 10 *salt rounds*, equilibrando segurança com performance. A segunda camada consiste em middleware de validação aplicado a todas as rotas protegidas, verificando a presença e validade do token. A terceira camada valida a propriedade do *tenant* em todas as *server actions* que manipulam dados, assegurando que usuários acessem apenas recursos de seu próprio restaurante.

Quanto à validação de inputs, seguem-se as recomendações da OWASP: todos os dados passam por validação rigorosa via *Zod schema* antes de serem processados. Esta validação ocorre tanto no *client-side* (para feedback imediato) quanto no *server-side* (garantindo segurança mesmo quando a validação *client-side* é contornada). A escolha do *Zod* deve-se à sua capacidade de *type inference*, permitindo que *TypeScript* derive tipos automaticamente dos *schemas* de validação, o que garante consistência entre validação *runtime* e *type checking compile-time*.

A proteção contra *Cross-Site Request Forgery* (CSRF) é implementada automaticamente pelas *server actions* do *Next.js*, que incluem validação de token baseada no padrão *double-submit cookie*. Já a comunicação HTTPS obrigatória, enforçada via *headers* HSTS (HTTP *Strict Transport Security*), garante criptografia em trânsito, atendendo aos requisitos da LGPD (Lei 13.709/2018) para proteção de dados pessoais.

A LGPD estabelece os princípios de finalidade, adequação e necessidade para o processamento de dados pessoais. O Serve Aí coleta apenas os dados estritamente necessários – CPF e nome para identificação de pedidos – em conformidade com o princípio de minimização. Visando conformidade futura com os requisitos de retenção limitada da LGPD, a arquitetura foi projetada para permitir a implementação de exclusão automática de dados de pedidos antigos após período de retenção definido, funcionalidade planejada para evolução pós-MVP.

1.6 Trabalhos Relacionados

Para posicionar o Serve Aí no ecossistema de soluções tecnológicas para o setor gastronômico, analisaram-se três categorias principais: plataformas de delivery, sistemas de ponto de venda (PDV) tradicionais e soluções SaaS de gestão. Cabe ressaltar que esta análise baseia-se em documentação pública, artigos técnicos, blogs oficiais e engenharia reversa de interfaces, sem acesso direto ao código-fonte proprietário. As inferências sobre arquiteturas internas apoiam-se em postagens técnicas oficiais e apresentações em conferências de desenvolvedores, concentrando-se nas características observáveis: modelo de negócio, experiência de usuário e funcionalidades oferecidas.

Plataformas de Delivery (iFood, Uber Eats)

iFood e Uber Eats constituem plataformas consolidadas de delivery, baseadas em arquiteturas de microsserviços que viabilizam a escalabilidade independente de serviços como processamento de pedidos, autenticação e pagamentos, conforme demanda (Vogels, 2009). Por outro lado, essa abordagem introduz complexidade operacional significativa, envolvendo *service discovery*, *distributed tracing* e eventual *consistency* entre serviços.

Quanto ao modelo de negócio, essas plataformas concentram-se primariamente no canal de delivery, sem oferecer ferramentas robustas de gestão operacional integradas. Dados da Associação Brasileira de Bares e Restaurantes (ABRASEL, 2022) indicam que marketplaces cobram comissões entre 12% e 27% sobre o valor dos pedidos, impactando significativamente as margens de lucro – especialmente quando se considera que a margem operacional média do setor gastronômico brasileiro situa-se entre 8% e 15% (ABF, 2023). Esta estrutura de custo motiva estabelecimentos a buscar soluções próprias que eliminem intermediários.

Sistemas PDV Tradicionais (TOTVS, Linx)

Sistemas PDV tradicionais como TOTVS e Linx oferecem gestão operacional completa, incluindo controle de estoque, fiscal e financeiro. Conforme documentado por Pressman e Maxim (2021), esses sistemas utilizam arquiteturas *client-server* tradicionais com interfaces desktop desenvolvidas em tecnologias legadas.

A principal limitação dessas soluções reside na ausência de módulos modernos de autoatendimento *mobile-first*. Quando oferecem interfaces web, estas frequentemente constituem adaptações de interfaces desktop, resultando em

experiência de usuário subótima em dispositivos móveis – violando princípios de *Progressive Enhancement* documentados por Champeon (2003). Adicionalmente, não implementam arquiteturas *multi-tenant* verdadeiras, requerendo instalação *on-premise* ou instâncias de banco de dados dedicadas por cliente, elevando custos de infraestrutura comparado ao modelo SaaS (Chong e Carraro, 2006).

Soluções SaaS de Gestão (CloudKitchens, Rappi Turbo)

CloudKitchens e soluções similares combinam infraestrutura física (*dark kitchens*) com software de gestão otimizado para produção em alto volume. A arquitetura desses sistemas enfatiza eficiência de produção por meio de algoritmos de otimização de rotas de preparação e gestão de múltiplas marcas virtuais.

Entretanto, essas soluções não foram projetadas para estabelecimentos com atendimento presencial, nem priorizam a experiência de autoatendimento do consumidor final no local. O foco em *dark kitchens* – cozinhas sem atendimento presencial – atende a um nicho específico do mercado gastronômico, deixando de lado os estabelecimentos que dependem do atendimento direto ao público.

Diferenciação do Serve Aí

Não se pretende aqui competir diretamente com iFood ou sistemas PDV empresariais, que contam com equipes de centenas de desenvolvedores e orçamentos milionários. O nicho visado é diferente: pequenos estabelecimentos que buscam alternativa aos custos dos marketplaces mas não desejam a complexidade de um PDV completo com módulos fiscais e de estoque que talvez nunca utilizem.

As principais diferenças observáveis são:

- **Sem necessidade de app nativo:** Enquanto marketplaces exigem download de aplicativo, o Serve Aí funciona diretamente no navegador, reduzindo fricção. Esta característica mostra-se especialmente relevante para estabelecimentos que atendem público eventual, como praças de alimentação e eventos.
- **Custo previsível:** O modelo de assinatura fixa contrasta com a comissão por pedido. A estrutura de comissionamento percentual sobre vendas pode representar custo operacional significativo para estabelecimentos de médio

e alto volume, tornando modelos de assinatura fixa economicamente mais atraentes.

- **Propriedade dos dados:** Em marketplaces, o estabelecimento não tem acesso completo aos dados de clientes (CPFs ficam ofuscados). Com o Serve Aí, o restaurante detém todos os dados, viabilizando marketing direto e programas de fidelidade.

É importante reconhecer, contudo, as limitações: o Serve Aí não oferece logística de entrega (motoristas, roteirização) nem marketplace com milhões de usuários ativos. Trata-se, portanto, de propostas de valor distintas.

Quadro 1 – Principais características das soluções analisadas

Característica	Marketplaces	PDV Tadicionais	Dark-kitchens	Serve Aí
Arquitetura	Microsserviços	Client-Server	Híbrida	Multi-tenant SSR
Modelo de custo	Comissão 12–27%	Licença + implantação	Infraestrutura física + software	Assinatura fixa
Autoatendimento	App nativo	Ausente	Não aplicável	Progressive Web App
Gestão operacional	Limitada	Completa (estoque, fiscal)	Otimização de produção	Pedidos + cardápio
Propriedade de dados	Parcial (CPF ofuscado)	Total	Total	Total
Deploy	Centralizado	On-premise	Centralizado	Cloud (Saas)
Complexidade	Alta	Alta	Média	Baixa

Fonte: Elaborado pelos autores (2025).

A contribuição principal deste trabalho é demonstrar a viabilidade técnica de uma arquitetura que unifica autoatendimento otimizado para conversão com gestão operacional completa, usando tecnologias modernas e *open-source*. Isso viabiliza desenvolvimento por equipes pequenas, diferentemente de plataformas consolidadas que requerem centenas de engenheiros.

2 METODOLOGIA

Este capítulo descreve os procedimentos metodológicos adotados no desenvolvimento do Serve Aí: natureza da pesquisa, fases do processo de desenvolvimento e justificativas para as escolhas tecnológicas. A metodologia buscou conciliar rigor acadêmico com o pragmatismo necessário ao desenvolvimento de software, alinhando-se com práticas contemporâneas de engenharia de software.

2.1 Natureza da Pesquisa

A pesquisa desenvolvida caracteriza-se como **aplicada e experimental**. Segundo Prodanov e Freitas (2013), pesquisa aplicada visa gerar conhecimentos para aplicação prática, voltados à solução de problemas específicos. Neste trabalho, o problema abordado consiste na necessidade de estabelecimentos gastronômicos por soluções tecnológicas que unifiquem autoatendimento digital e gestão operacional em plataforma única e escalável.

O caráter experimental manifesta-se na construção e validação de uma solução tecnológica funcional, na qual hipóteses arquiteturais são testadas por meio de implementação concreta. Gil (2008) define pesquisa experimental pela manipulação controlada de variáveis – neste caso, escolhas de arquitetura e tecnologia – para avaliar seus efeitos sobre a performance do sistema, usabilidade das interfaces e escalabilidade da solução.

A abordagem metodológica adotada aproxima-se da pesquisa-ação participante (Thiollent, 2011), visto que o pesquisador atuou simultaneamente como desenvolvedor da solução, viabilizando feedback iterativo entre teoria e prática. Esta abordagem mostra-se apropriada para projetos de desenvolvimento de software, nos quais requisitos emergem e refinam-se durante a construção.

2.2 Processo de Desenvolvimento

O desenvolvimento adotou abordagem iterativa e incremental, fundamentada em princípios ágeis adaptados ao contexto acadêmico. Sommerville (2016) argumenta que o desenvolvimento iterativo viabiliza validação contínua de decisões arquiteturais

por meio de protótipos funcionais, reduzindo riscos de falhas tardias. O processo organizou-se em cinco fases principais.

2.2.1 Fase 1: Levantamento de Requisitos e Modelagem Conceitual

A primeira fase concentrou-se na identificação e caracterização dos perfis de usuário: consumidores finais e gestores operacionais. Seguindo a metodologia de personas de Cooper (1999), definiram-se características, objetivos e limitações de cada perfil. Consumidores caracterizam-se por baixa tolerância à complexidade, uso predominante mobile e expectativa de processo intuitivo. Gestores, por sua vez, demandam densidade informacional, funcionalidades analíticas e acesso predominante via desktop.

Essa análise levou à decisão de segregar a aplicação em dois módulos funcionalmente distintos, aplicando o conceito de *Bounded Contexts do Domain-Driven Design* (Evans, 2003). Cada módulo trabalha com linguagem ubíqua específica: o módulo cliente utiliza termos como "carrinho" e "pedidos", enquanto o módulo administrativo emprega vocábulos como "gestão" e "operações".

Para a modelagem de dados, utilizou-se *Prisma Schema Language* na definição do modelo relacional, contemplando as entidades principais: `Restaurant` (*tenant*), `MenuCategory`, `Product`, `Order`, `OrderProduct` (*junction table*) e `User`. A normalização seguiu 3NF visando minimizar redundância (Codd, 1970). Índices compostos foram planejados estrategicamente para queries *multi-tenant* frequentes, particularmente filtros por `(restaurantId, status)` e `(restaurantId, createdAt)`.

2.2.2 Fase 2: Prototipação e Definição de Arquitetura

A arquitetura foi organizada em três camadas distintas, seguindo o padrão de separação de responsabilidades proposto por Buschmann et al. (1996). A primeira camada cuida da apresentação visual (o que o usuário vê), a segunda gerencia as regras de negócio (como os dados são processados), e a terceira lida com o armazenamento no banco de dados.

Para a arquitetura *multi-tenant*, adotou-se o nível três da classificação de Chong e Carraro (2006). Esta escolha significa que todos os restaurantes

compartilham a mesma aplicação e o mesmo banco de dados, mas os dados de cada um ficam completamente isolados. Outras opções, como manter bancos separados para cada cliente, ofereceriam maior isolamento mas custariam muito mais, inviabilizando um preço acessível. O nível escolhido equilibra segurança com custo, garantindo que cada restaurante acesse apenas seus próprios dados através de filtros automáticos em todas as consultas.

Protótipos iniciais da interface foram criados no Figma, ferramenta de design, para testar como os usuários navegariam pelo sistema. Durante as aulas de Laboratório de Engenharia de Software, o professor orientador e os colegas de curso revisaram essas versões iniciais. Os feedbacks recebidos mostraram que o processo de finalização de pedido estava complicado demais, levando à simplificação para apenas duas etapas.

2.2.3 Fase 3: Implementação do Módulo Cliente

A implementação do módulo de autoatendimento trouxe desafios relacionados à organização visual e ao armazenamento temporário de informações. A primeira dificuldade apareceu na forma de exibir as categorias do cardápio. Listar todas as categorias de uma vez deixava a tela confusa em celulares, enquanto ocultá-las em menus retráteis dificultava que os clientes encontrassem o que procuravam. A solução escolhida permite que o usuário deslize o dedo horizontalmente para navegar entre categorias, similar ao funcionamento de aplicativos como iFood e Uber Eats.

Outro ponto importante foi decidir como armazenar o carrinho de compras enquanto o cliente navega pelo aplicativo. Em aplicações *React*, esses dados que mudam conforme o usuário interage com o sistema são chamados de "estado". Existem diferentes formas de organizar esse armazenamento. *Redux*, por exemplo, é uma ferramenta que centraliza todos os dados em um único local e controla rigorosamente como eles podem ser alterados, seguindo o padrão *Flux* proposto pela equipe do Facebook em 2014. Essa abordagem funciona bem em sistemas complexos onde muitas partes diferentes precisam acessar e modificar os mesmos dados, mas mostrou-se excessivamente elaborada para o módulo cliente, onde o principal dado armazenado é simplesmente a lista de produtos no carrinho.

A alternativa escolhida foi a *Context* API, uma funcionalidade própria do *React* que permite compartilhar informações entre diferentes partes da interface sem

precisar repassá-las manualmente em cada nível. Essa solução atendeu bem as necessidades do projeto: adicionar produtos ao carrinho, remover itens e alterar quantidades funcionam de maneira direta, sem adicionar bibliotecas externas que aumentariam o peso da aplicação. Caso o sistema evolua para funcionalidades mais sofisticadas – como manter o carrinho sincronizado entre o celular e o computador do mesmo cliente – ferramentas mais robustas como *Redux* ou *Zustand* poderiam ser consideradas.

Para a validação de CPF, implementou-se o algoritmo oficial da Receita Federal baseado em módulo 11 (Brasil, 2001). Esse algoritmo matemático aplica uma sequência de multiplicações aos dígitos do CPF e valida se os dois últimos dígitos estão corretos. Assim, o sistema consegue detectar CPFs digitados incorretamente antes mesmo de tentar processá-los, reduzindo erros e melhorando a qualidade dos dados coletados.

2.2.4 Fase 4: Implementação do Módulo Administrativo

O painel administrativo implementou gestão de pedidos com atualização *near real-time* via *Incremental Static Regeneration* (ISR). A configuração `revalidate = 30` para pedidos pendentes fundamenta-se nos princípios de *stale-while-revalidate* discutidos na seção 1.2, garantindo que alterações de status reflitam na interface em até 30 segundos, equilibrando atualidade com carga no servidor.

A paginação *server-side* foi implementada com Prisma `skip/take`, assegurando performance constante independente do volume de dados. Durante o desenvolvimento, foram realizadas validações empíricas de performance com *dataset* sintético de 50.000 registros, observando-se tempos de resposta inferiores a 100ms em ambiente de desenvolvimento local. Cabe ressaltar que uma metodologia formal de benchmarking não foi aplicada, sendo esta uma oportunidade para trabalhos futuros que desejem validar a escalabilidade em ambientes de produção com carga real.

A integração com Power BI via *iframe embedding* fornece visualizações analíticas sem necessidade de implementação *custom* de dashboards. Essa decisão exemplifica o princípio de composição sobre implementação, reutilizando ferramentas especializadas.

O CRUD de produtos e categorias implementa validação em múltiplas camadas: validação de *schema* via *Zod* no *frontend*, revalidação *server-side* antes de persistência e *constraints* de banco de dados como última linha de defesa. Essa abordagem aplica *defense in depth* em nível de validação de dados.

2.2.5 Fase 5: Integração e Validação

A integração com *Stripe* revelou-se mais complexa que o previsto. A implementação inicial baseava-se em redirecionamento *client-side* – abordagem insegura que permitiria simulação de pagamentos via manipulação de URLs. A migração para *webhooks* exigiu compreensão de assinatura criptográfica e tratamento de entrega assíncrona, considerando que eventos podem chegar fora de ordem.

A solução final utiliza *webhooks* como fonte única de verdade: pedidos mudam para `PAYMENT_CONFIRMED` apenas após evento criptograficamente verificado. Uma limitação persiste: o sistema opera em modo *sandbox* devido a requisitos de conta comercial e compliance fiscal que excedem o escopo acadêmico. Operação comercial demandaria gateway nacional (PagSeguro, Mercado Pago) e emissão de notas fiscais.

A validação de segurança seguiu checklist OWASP Top 10, implementando proteções específicas:

A01 - *Broken Access Control*: Validação de *tenant ownership* em todas as *mutations*;

A02 - *Cryptographic Failures*: HTTPS obrigatório, senhas com *bcrypt*;

A03 - *Injection*: Uso de ORM (Prisma) com queries parametrizadas;

A04 - *Insecure Design*: Princípio de privilégio mínimo em queries (*select* apenas campos necessários);

A05 - *Security Misconfiguration*: *Headers* de segurança (HSTS, X-Frame-Options).

A arquitetura SSR com *React Server Components* foi projetada para otimizar *Core Web Vitals*, priorizando redução de *Time to Interactive* (TTI) por meio de renderização *server-side* e minimização de *JavaScript client-side*. Esta abordagem alinha-se com as recomendações de Osmani (2017) sobre redução de custos de *JavaScript* em aplicações web, particularmente crítico para dispositivos com recursos limitados e conexões móveis.

2.3 Arquitetura e Tecnologias

A arquitetura final adota padrão de três camadas com isolamento *multi-tenant* transversal, conforme diagrama conceitual apresentado na Figura 7 e 8.

2.3.1 Camada de Apresentação

Next.js 14 com *App Router* e *React Server Components*

A escolha do *Next.js* justifica-se por suas características alinhadas aos requisitos do projeto. Inicialmente, considerou-se uma SPA tradicional com *Create React App* pela familiaridade da ferramenta. Contudo, *o Next.js* ofereceu vantagens determinantes: SSR nativo, permitindo que o HTML chegue pronto ao navegador; suporte integrado a *TypeScript*, eliminando configurações complexas de build; e documentação robusta com comunidade ativa, aspectos relevantes para um projeto individual desenvolvido em tempo limitado.

React 18 com *Server Components*

React Server Components representaram uma curva de aprendizado significativa. A distinção entre componentes servidor e cliente não é intuitiva, havendo inicialmente tentativas frustradas de usar *hooks* como `useState` em *Server Components*, o que gerava erros crípticos. A superação dessas dificuldades ocorreu através de consulta a tutoriais online especializados, análise de discussões em fóruns de programação como Stack Overflow, e utilização de assistentes de inteligência artificial para revisão das lógicas de programação empregadas e sugestão de correções pontuais.

Apesar das dificuldades iniciais, *Server Components* revelaram-se essenciais para reduzir *bundle size*: componentes que apenas consultam dados e renderizam listas não precisam de *JavaScript* no cliente, economizando *kilobytes* valiosos em conexões lentas.

2.3.2 Camada de Lógica de Negócio

Server Actions do *Next.js*

Server actions representam paradigma inovador que elimina a necessidade de API REST tradicional. Funções marcadas com a diretiva `use server` executam no servidor mas podem ser invocadas diretamente de componentes cliente, com o *Next.js* gerenciando serialização, transporte e desserialização automaticamente.

Esta abordagem oferece benefícios relevantes: redução de latência comparada ao padrão REST tradicional, validação centralizada no servidor (impossível de contornar), e *type safety* garantida pelo *TypeScript* na validação de argumentos e retorno.

2.3.3 Camada de Dados

PostgreSQL via Prisma ORM

PostgreSQL foi escolhido principalmente pela integração direta com a Vercel, plataforma utilizada para hospedagem do projeto. Essa integração simplifica a criação e configuração do banco de dados, eliminando etapas manuais complexas. MongoDB também foi considerado pela sua flexibilidade na estrutura de dados, mas a forma como as informações se relacionam no sistema – pedidos conectados a produtos, produtos organizados em categorias – funciona melhor em bancos de dados relacionais tradicionais.

A ferramenta Prisma facilita bastante o trabalho com bancos de dados em projetos *TypeScript*. Ela lê a estrutura do banco e gera automaticamente as definições de tipos, impedindo que o desenvolvedor tente buscar campos que não existem ou use tipos incompatíveis – o *TypeScript* detecta esses problemas ainda durante a escrita do código. Outro recurso útil é o sistema de migrações: conforme o banco de dados foi evoluindo durante o desenvolvimento (novos índices, alterações nos relacionamentos), cada mudança ficou registrada em arquivos separados, permitindo acompanhar toda a evolução do *schema*.

Quanto ao gerenciamento de conexões com o banco, foram utilizados valores conservadores sugeridos pela própria documentação do Prisma: até 5 conexões simultâneas no ambiente de desenvolvimento local e até 20 em produção. Essa configuração equilibra o uso de recursos do servidor com a capacidade de atender múltiplas requisições ao mesmo tempo.

2.3.4 Tecnologias Complementares

TypeScript

TypeScript adiciona sistema de tipos estático a *JavaScript*, permitindo detecção de erros em desenvolvimento. A configuração utiliza *strict mode*, ativando verificações mais rigorosas como `strictNullChecks` e `noImplicitAny`.

Zod

Biblioteca de validação de *schemas* que oferece *type inference*, permitindo que *TypeScript* derive tipos automaticamente dos *schemas* de validação. Exemplo:

Figura 1 – Exemplo de validação Zod

```
const orderSchema = z.object({
  customerName: z.string().min(3),
  customerCpf: z.string().regex(/^\d{11}$/),
});

type Order = z.infer<typeof orderSchema>; // TypeScript infere tipo automaticamente
```

Fonte: Desenvolvido pelos autores (2025).

***TanStack Table* v8**

Biblioteca especializada em funcionalidades avançadas de tabelas (*sorting*, *filtering*, *pagination*) utilizada no módulo administrativo. A escolha evita reimplementação de funcionalidades complexas, seguindo princípio de reutilização de componentes.

***Tailwind* CSS**

Framework CSS *utility-first* que permite desenvolvimento rápido mantendo consistência visual. A abordagem *utility-first* reduz CSS customizado, facilitando manutenção.

Vercel

Plataforma de *deploy* otimizada para *Next.js*, oferecendo *edge network global* para redução de latência, *deploy* automático com integração Git, e *preview deployments* que geram URL de prévia para cada *pull request*, facilitando validação antes do merge.

A integração dessas tecnologias materializa os princípios teóricos discutidos na fundamentação: SSR implementa *Progressive Enhancement*, *multi-tenant* via Prisma garante isolamento, *TypeScript* com *Zod* aplicam *defense in depth* em validação, e a arquitetura em camadas respeita os princípios SOLID.

Quadro 2 – Síntese das tecnologias utilizadas

Camada	Tecnologia	Justificativa
Frontend	*Next.js* 14 + *React* 18	"SSR nativo, *App Router*, *Server Components*"
Linguagem	*TypeScript*	"*Type safety*, detecção precoce de erros"
Backend	*Next.js Server Actions*	"Elimina API REST tradicional, *type safety*"
Banco de Dados	PostgreSQL + Prisma ORM	"Relacionamentos complexos, *type inference*"
Pagamentos	Stripe	"*Webhooks* seguros, *sandbox* para testes"
UI	*Tailwind* CSS + *shadcn/ui*	"*Utility-first*, componentes acessíveis"

Fonte: Elaborada pelos autores (2025)

3 DESENVOLVIMENTO E RESULTADOS

Este capítulo documenta a implementação prática do Serve Aí: a arquitetura construída, as decisões tomadas durante o design e modelagem, os desafios enfrentados na codificação e os resultados alcançados. A discussão demonstra como os conceitos teóricos apresentados anteriormente ganharam forma em uma aplicação funcional.

3.1 Arquitetura da Solução

A arquitetura construída divide-se em três camadas funcionalmente distintas, conforme a Figura 7 e 8 ilustra. A primeira camada (Apresentação) cuida da renderização e interação com o usuário. A segunda camada (Lógica de Negócio) implementa as regras e validações do sistema. A terceira camada (Acesso a Dados) gerencia a comunicação com o banco de dados PostgreSQL.

3.1.1 Visão Geral da Arquitetura

A arquitetura combina características de dois modelos conhecidos. Das arquiteturas monolíticas, onde todos os componentes funcionam juntos em um único processo, herdou-se a simplicidade operacional. Dos microsserviços, adotou-se a separação clara de responsabilidades entre módulos. Essa combinação mostrou-se apropriada porque, embora microsserviços puros ofereçam vantagens de escalabilidade, eles trazem complexidade operacional que excederia as necessidades iniciais do projeto.

Componentes Principais:

- ***Next.js App Router*:** *Framework* que orquestra renderização server-side, roteamento e gerenciamento de estado;
- ***React Server Components*:** Componentes que executam exclusivamente no servidor, reduzindo *JavaScript* enviado ao cliente;
- ***Server Actions*:** Funções *server-side* invocáveis diretamente de componentes cliente;

- **Prisma *Client*:** ORM que abstrai acesso ao banco de dados PostgreSQL;
- ***Middleware* de Autenticação:** Camada de validação de tokens JWT em rotas protegidas.

Fluxo de Requisição:

Quando um cliente acessa o módulo de **autoatendimento**:

1) O usuário acessa a URL do restaurante (exemplo: `/restaurante-abc/menu`);
2) Um middleware valida o identificador do estabelecimento, retornando erro 404 caso seja inválido;
3) O componente servidor busca os dados necessários no banco, filtrando pelo restaurante específico;
4) O servidor processa esses dados e gera o HTML completo da página;
5) O HTML é enviado ao navegador com apenas o *JavaScript* essencial para interatividade;
6) Componentes interativos, como o carrinho de compras, ativam-se no navegador do cliente;

Para requisições do módulo **administrativo**:

1) Usuário acessa rota protegida (ex: `/restaurante-abc/dashboard`);
2) Middleware verifica token JWT no cookie;
3) Se token ausente/inválido, redireciona para `/admin` (login);
4) Se token válido, extrai `restaurantId` do *payload*;
5) *Server Component* executa queries com cláusula `WHERE restaurantId = ...`;
6) Dados renderizam-se e enviam-se ao cliente.

3.1.2 Isolamento Multi-Tenant

O sistema garante que dados de diferentes restaurantes permaneçam separados através de três camadas de proteção:

Primeira Camada: Identificação por URL

Cada estabelecimento possui um identificador único em sua URL. O sistema valida esse identificador consultando o banco de dados:

Figura 2 – Identificação por slug

```
const restaurant = await db.restaurant.findUnique({
  where: { slug },
});
if (!restaurant) return notFound();
```

Fonte: Desenvolvido pelos autores (2025).

Segunda Camada: Autenticação de Gestores

As rotas administrativas exigem autenticação via token JWT. Esse token carrega internamente o identificador do restaurante ao qual o gestor pertence:

Figura 3 – Autenticação com token JWT

```
const token = sign(
  { userId: user.id, restaurantId: user.restaurant.id },
  process.env.JWT_SECRET!,
  { expiresIn: "1d" },
);
```

Fonte: Desenvolvido pelos autores (2025).

Terceira Camada: Filtragem nas Consultas

Todas as consultas ao banco incluem automaticamente o filtro por restaurante:

Figura 4 – Filtragem de consultas

```
const orders = await db.order.findMany({
  where: { restaurantId: restaurant.id, status: "PENDING" },
});
```

Fonte: Desenvolvido pelos autores (2025).

Essas três camadas funcionam como redes de segurança sobrepostas. Mesmo que ocorra um erro na primeira validação, as camadas subsequentes impedem que dados vazem entre restaurantes diferentes.

3.1.3 Estratégias de Cache e Revalidação

O sistema utiliza cache em várias camadas para melhorar a velocidade de resposta. O cache funciona como uma memória temporária que evita consultas repetidas ao banco de dados.

Regeneração Incremental (ISR):

As páginas armazenam dados em cache por períodos específicos, atualizando-se automaticamente quando esse tempo expira:

- Cardápio digital: atualiza a cada 5 minutos;
- Pedidos pendentes: atualiza a cada 30 segundos;
- Histórico de pedidos: atualiza a cada 5 minutos.

Atualização sob Demanda:

Quando dados são modificados (por exemplo, ao alterar o status de um pedido), o sistema força uma atualização imediata do cache correspondente:

Figura 5 – Exemplo de atualização *on-demand*

```
export async function updateOrderStatus(orderId: number, status: OrderStatus) {
  await db.order.update({ where: { id: orderId }, data: { status } });
  revalidatePath("/[slug]/dashboard/pending-orders");
}
```

Fonte: Desenvolvido pelos autores (2025).

Connection Pooling:

Prisma Client mantém *pool* de conexões configurado via variáveis de ambiente:

- Desenvolvimento: `DATABASE_CONNECTION_LIMIT=5;`
- Produção: `DATABASE_CONNECTION_LIMIT=20;`

3.2 Modelagem e Design

3.2.1 Modelagem de Banco de Dados

O banco de dados seguiu princípios de normalização até a Terceira Forma Normal (3NF), técnica que organiza os dados para evitar redundâncias. Em alguns pontos específicos, optou-se estrategicamente por armazenar informações duplicadas

quando isso melhoraria significativamente a velocidade de consultas frequentes. O modelo compreende sete tabelas principais:

Entidade Restaurant (*Tenant*)

Representa estabelecimento gastronômico. Atributos principais:

- `id` (UUID): Identificador único;
- `slug` (String, unique): Identificador *human-readable* para URLs;
- `name`, `description`: Metadados descritivos;
- `avatarImageUrl`, `coverImageUrl`: URLs de imagens.

Entidade *MenuCategory*

Categorias organizacionais do cardápio. Relacionamento 1:N com *Restaurant*:

- `restaurantId` (*Foreign Key*): Relacionamento com *Restaurant*;
- Índice em `restaurantId` para otimização de *queries*.

Entidade *Product*

Produtos/itens do cardápio. Relacionamentos:

- N:1 com *Restaurant* (obrigatório);
- N:1 com *MenuCategory* (opcional, permite produtos sem categoria);
- `ingredients` (String[]): *Array* de *strings* para flexibilidade.

Entidade *Order*

Pedidos realizados. Atributos críticos:

- `status` (Enum): `PENDING`, `IN_PREPARATION`, `PAYMENT_CONFIRMED`, `PAYMENT_FAILED`, `FINISHED`;
- `consumptionMethod` (Enum): `TAKEAWAY`, `DINE_IN`;
- `customerCpf` (String): Indexado para queries por cliente;
- `total` (Float): Desnormalizado para performance (evita `JOIN` com `OrderProduct`).

Entidade *OrderProduct* (*Junction Table*)

Relacionamento N:N entre *Order* e *Product*:

- `price` (Float): *Snapshot* do preço no momento do pedido (crítico para consistência histórica);
- `quantity` (Int): Quantidade do produto no pedido.

Entidade *User*

Usuários administrativos. Relacionamento 1:1 com *Restaurant*:

- `passwordHash` (String): *Hash bcrypt* da senha;
- `email` (String, unique): Identificador de login.

Índices Compostos:

Índices otimizados para *queries multi-tenant*:

Figura 6 – Índices otimizados para as *queries multi-tenant*

```prisma
@@index([restaurantId, status])
@@index([restaurantId, createdAt])
@@index([customerCpf])
```

Fonte: Desenvolvido pelos autores (2025).

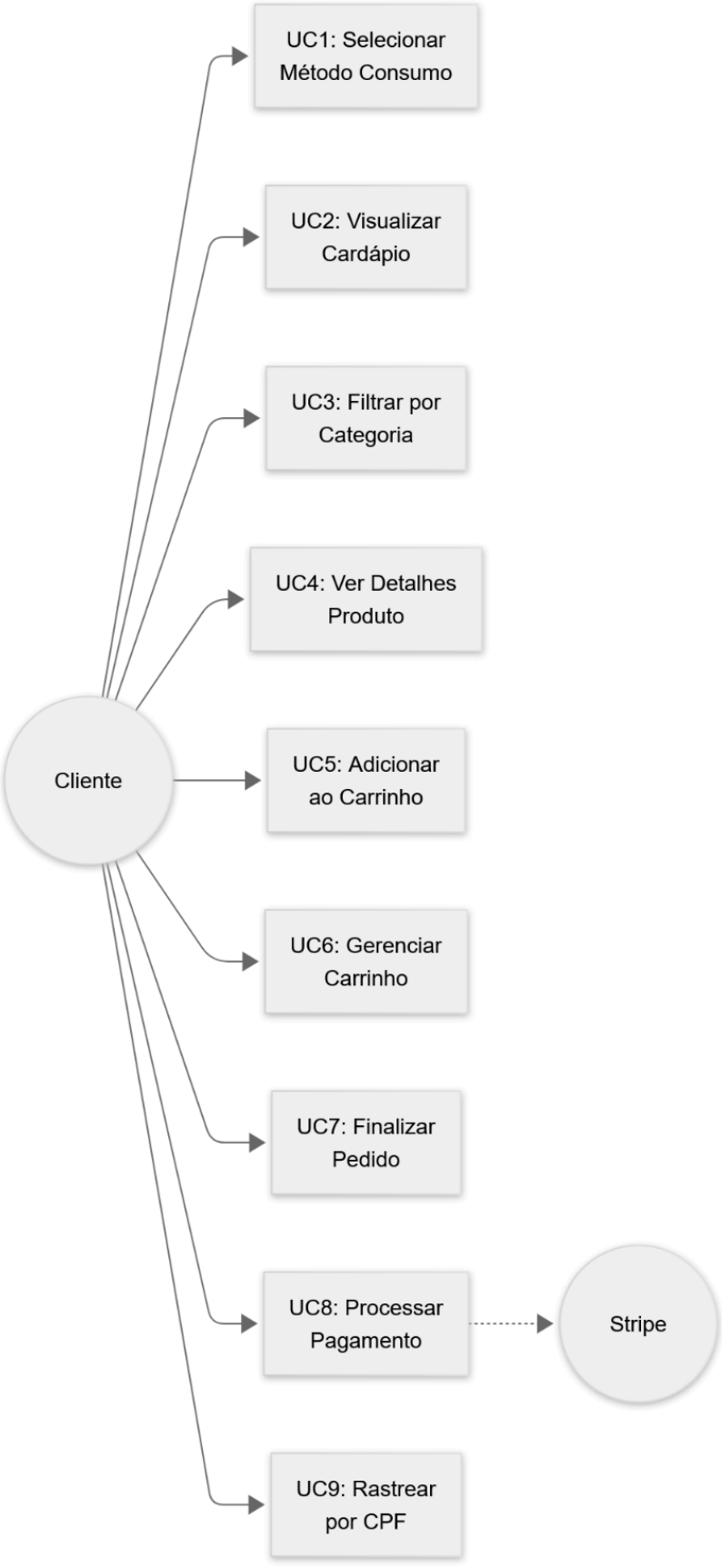
3.2.2 Diagramas UML do Sistema

Os diagramas UML a seguir documentam visualmente como o Serve Aí está organizado e quais funcionalidades oferece, facilitando a compreensão da estrutura e comportamento do sistema.

3.2.2.1 Diagrama de Casos de Uso

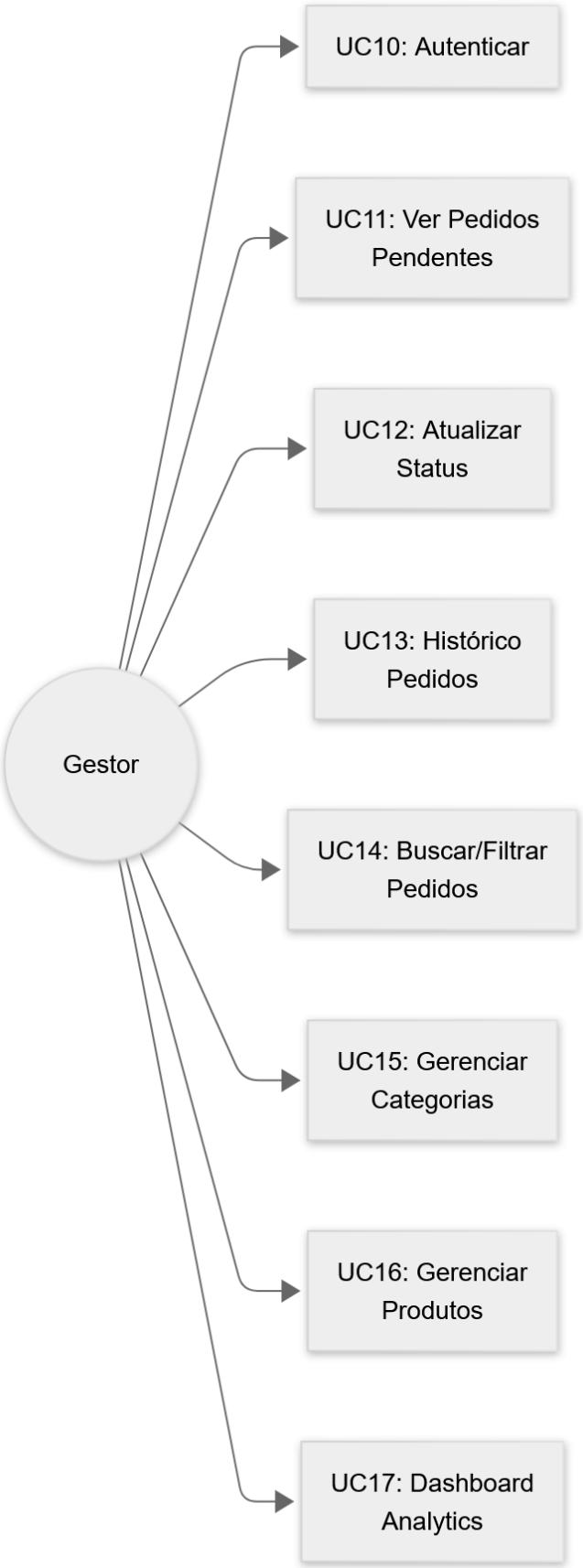
O Diagrama de Casos de Uso mostra quem interage com o sistema e o que cada perfil de usuário pode fazer. No Serve Aí, três atores principais foram identificados: Cliente (consumidor final), Gestor (administrador do estabelecimento) e Sistema de Pagamento (ator externo representado pelo Stripe).

Figura 7 – Diagrama de Casos de Uso (módulo cliente).



Fonte: Elaborado pelos autores (2025).

Figura 8 – Diagrama de Casos de Uso (módulo administrativo).



Fonte: Elaborado pelos autores (2025).

Descrição dos Casos de Uso:

Módulo Cliente:

UC1 - Selecionar Método de Consumo: O cliente escolhe se quer levar o pedido ou consumir no local. Esta escolha é obrigatória e acontece antes de navegar pelo cardápio.

UC2 - Visualizar Cardápio Digital: Mostra todos os produtos disponíveis com suas imagens, descrições e preços.

UC3 - Filtrar Produtos por Categoria: Permite navegar por categorias como bebidas, pratos principais e sobremesas.

UC4 - Visualizar Detalhes do Produto: Abre uma janela com informações detalhadas sobre o produto, incluindo ingredientes e imagem ampliada.

UC5 - Adicionar ao Carrinho: Coloca o produto selecionado no carrinho com a quantidade escolhida.

UC6 - Gerenciar Carrinho: Permite visualizar, alterar quantidades e remover produtos do carrinho.

UC7 - Finalizar Pedido: Coleta o nome e CPF do cliente, valida as informações e registra o pedido no sistema.

UC8 - Processar Pagamento: Realiza o pagamento via cartão através da integração com o *Stripe*.

UC9 - Rastrear Pedido por CPF: O cliente pode consultar o status de seus pedidos informando seu CPF.

Módulo Administrativo:

UC10 - Autenticar Usuário: Login do gestor via e-mail e senha com geração de token JWT. Necessário para acessar qualquer funcionalidade administrativa.

UC11 - Visualizar Pedidos Pendentes: Lista os pedidos que estão aguardando preparo ou em preparação, atualizando automaticamente a cada 30 segundos.

UC12 - Atualizar Status do Pedido: O gestor altera o status do pedido conforme o andamento (recebido → em preparo → finalizado).

UC13 - Consultar Histórico de Pedidos: Acesso a todos os pedidos já realizados, organizados em páginas de 30 pedidos.

UC14 - Buscar e Filtrar Pedidos: Busca por número do pedido, nome ou CPF do cliente. Permite filtrar por método de consumo, status e período.

UC15 - Gerenciar Categorias: CRUD completo de categorias do cardápio.

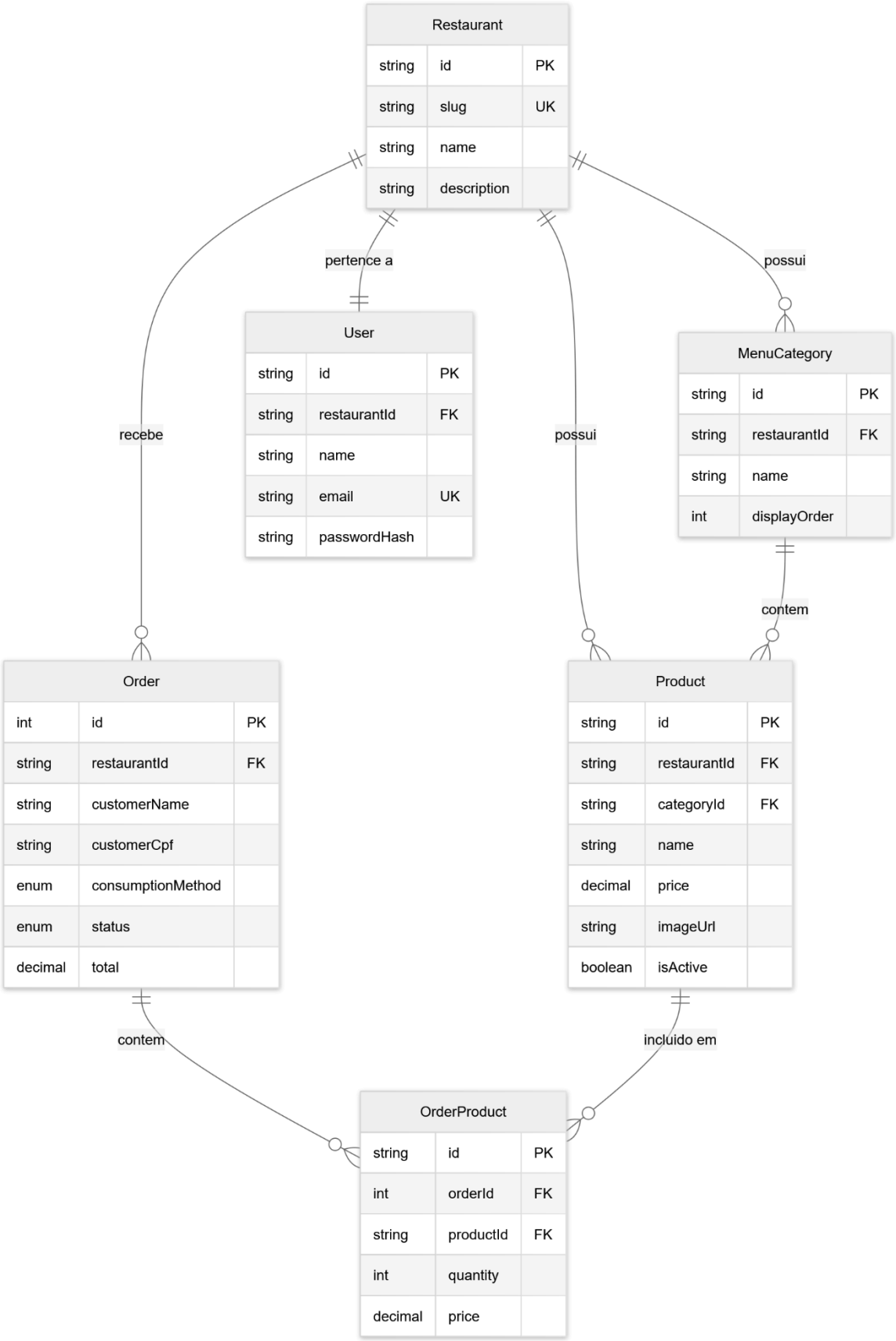
UC16 - Gerenciar Produtos: CRUD de produtos incluindo nome, preço, ingredientes, categoria e imagem.

UC17 - Visualizar Dashboard *Analytics*: Acesso a relatórios com métricas de vendas, produtos mais vendidos e horários de maior movimento.

3.2.2.2 Diagrama de Entidade-Relacionamento

O Diagrama de Entidade-Relacionamento (DER), exibido na Figura 9, mostra como o banco de dados está estruturado, evidenciando as sete tabelas principais e como elas se conectam. O modelo seguiu a Terceira Forma Normal (3NF) para evitar informações duplicadas, exceto no campo ‘total’ da tabela `Order`, onde optou-se estrategicamente por armazenar o valor total do pedido para acelerar consultas frequentes.

Figura 9 – Diagrama de Entidade de Relacionamento



Fonte: Elaborado pelos autores (2025).

Quadro 3 – Detalhamento dos atributos das entidades

Entidade	Atributos	Tipo	Observação
Restaurant	id	string	Chave primária (UUID)
	slug	string	Único – Identificador URL
	avatarImageUrl, coverImageUrl	string	URLs de imagens
	createdAt, updatedAt	datetime	Timestamps
MenuCategory	displayOrder	int	Ordenação customizada
Product	categoryId	string	Nullable – permite produtos sem categoria
	ingredients	string[]	Array JSON no PostgreSQL
	description	text	Descrição detalhada
Order	customerCpf	string	Indexado para busca
	consumptionMethod	enum	TAKEAWAY ou DINE_IN
	status	enum	PENDING, IN_PREPARATION, PAYMENT_CONFIRMED, PAYMENT_FAILED, FINISHED
	total	decimal	Desnormalizado para performance
	createdAt	datetime	Indexado com restaurantId
OrderProduct	price	decimal	Snapshot do preço no momento do pedido
User	passwordHash	string	Hash bcrypt com 10 salt rounds

Fonte: Elaborado pelos autores (2025).

Descrição das Entidades e Relacionamentos:

Restaurant (*Tenant*):

Tabela central que representa cada estabelecimento cadastrado no sistema. Cada restaurante possui um identificador único (*slug*) que aparece na URL. Um restaurante pode ter múltiplas categorias no cardápio, vários produtos e diversos pedidos ao longo do tempo. Cada restaurante possui exatamente um usuário administrador para gerenciar o painel de controle.

MenuCategory (Categoria do cardápio):

Organiza os produtos em grupos como “Bebidas”, “Pratos Principais” e “Sobremesas”. O campo `displayOrder` permite que o gestor defina a ordem em que as categorias aparecem na interface. Cada categoria pertence a um restaurante específico, garantindo que os dados fiquem separados.

Product (Produto):

Representa os itens disponíveis no cardápio. Cada produto pertence obrigatoriamente a um restaurante e opcionalmente a uma categoria. A lista de ingredientes é armazenada como um conjunto de textos no PostgreSQL. O campo `isActive` permite “desativar” produtos temporariamente sem 44paga-los do banco, preservando o histórico de pedidos antigos.

***Order* (Pedido):**

Guarda os pedidos realizados pelos clientes. O campo `customerCpf` possui um índice para agilizar buscas. O campo `status` acompanha o ciclo de vida do pedido: recebido → em preparo → pagamento confirmado → finalizado. O valor total do pedido fica armazenado diretamente aqui para evitar cálculos repetidos ao listar pedidos. Dois índices compostos aceleram as consultas mais frequentes: um para filtrar pedidos pendentes e outro para ordenar por data de criação.

***OrderProduct* (Produtos do pedido):**

Funciona como tabela intermediária conectando pedidos e produtos. O campo `price` registra o preço do produto no momento exato do pedido, garantindo que alterações futuras no preço não afetem pedidos antigos. O campo `quantity` permite que o cliente peça múltiplas unidades do mesmo produto.

***User* (Usuário):**

Representa os gestores que administram cada restaurante. A senha fica armazenada de forma criptografada usando *bcrypt*. Cada usuário pertence a exatamente um restaurante, garantindo que gestores só acessem dados de seus próprios estabelecimentos.

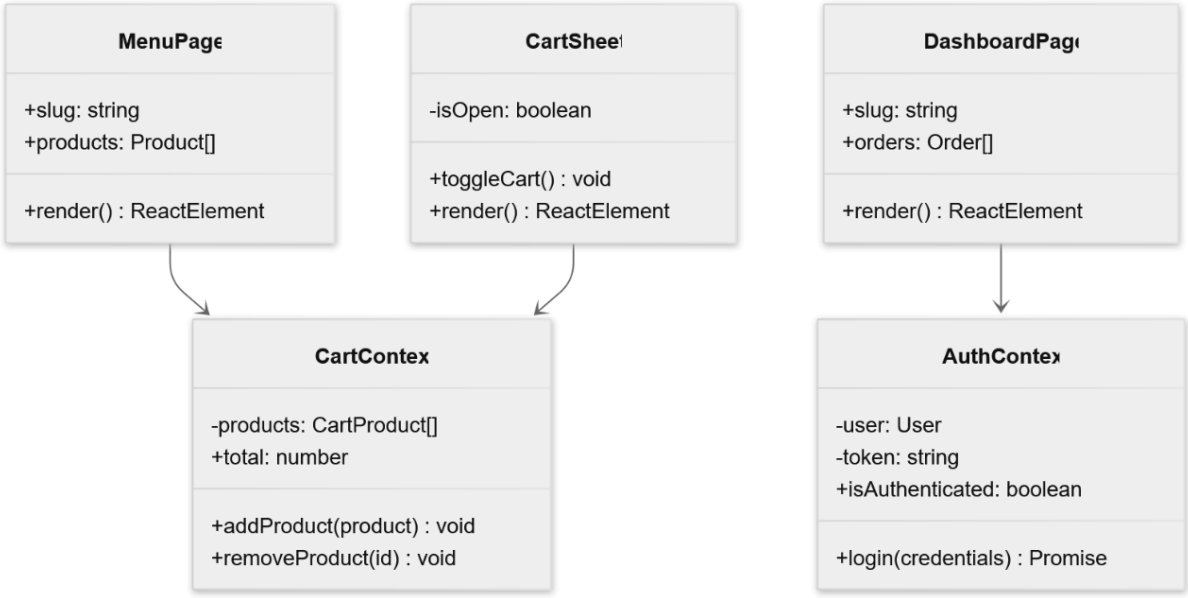
Isolamento entre Restaurantes:

Todas as tabelas (exceto *User*) incluem o campo `restaurantId`, garantindo que todas as consultas ao banco filtrem automaticamente os dados pelo restaurante correto. Essa estratégia implementa o isolamento lógico discutido por Chong e Carraro (2006), onde múltiplos clientes compartilham o mesmo banco mas seus dados permanecem completamente separados.

3.2.2.3 Diagrama de Classes

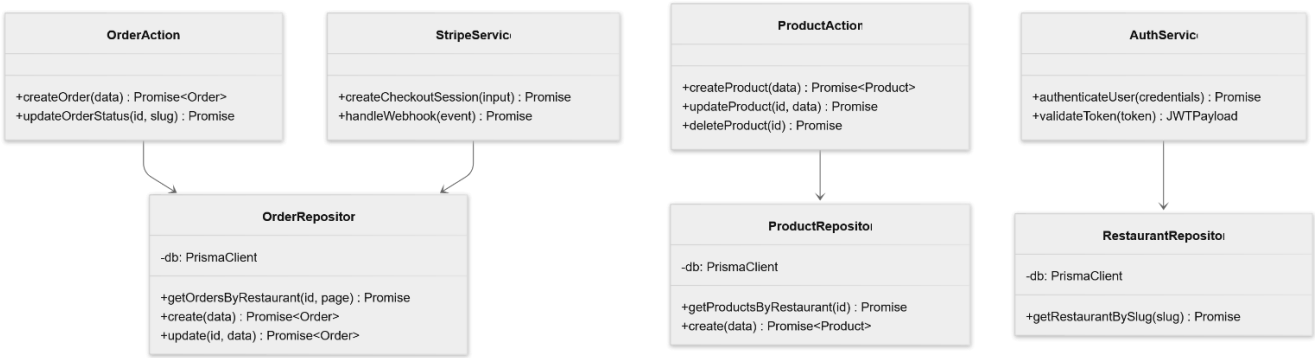
O Diagrama de Classes mostra como o código está organizado em termos de estruturas de dados e responsabilidades. Documenta as interfaces *TypeScript*, os contextos *React* que gerenciam informações compartilhadas, e como as diferentes partes do sistema (apresentação, lógica de negócio e acesso a dados) se separam seguindo o padrão MVC, mostrado nas Figuras 10, 11 e 12.

Figura 10 – Diagrama de Classes (camada de apresentação e *state management*)



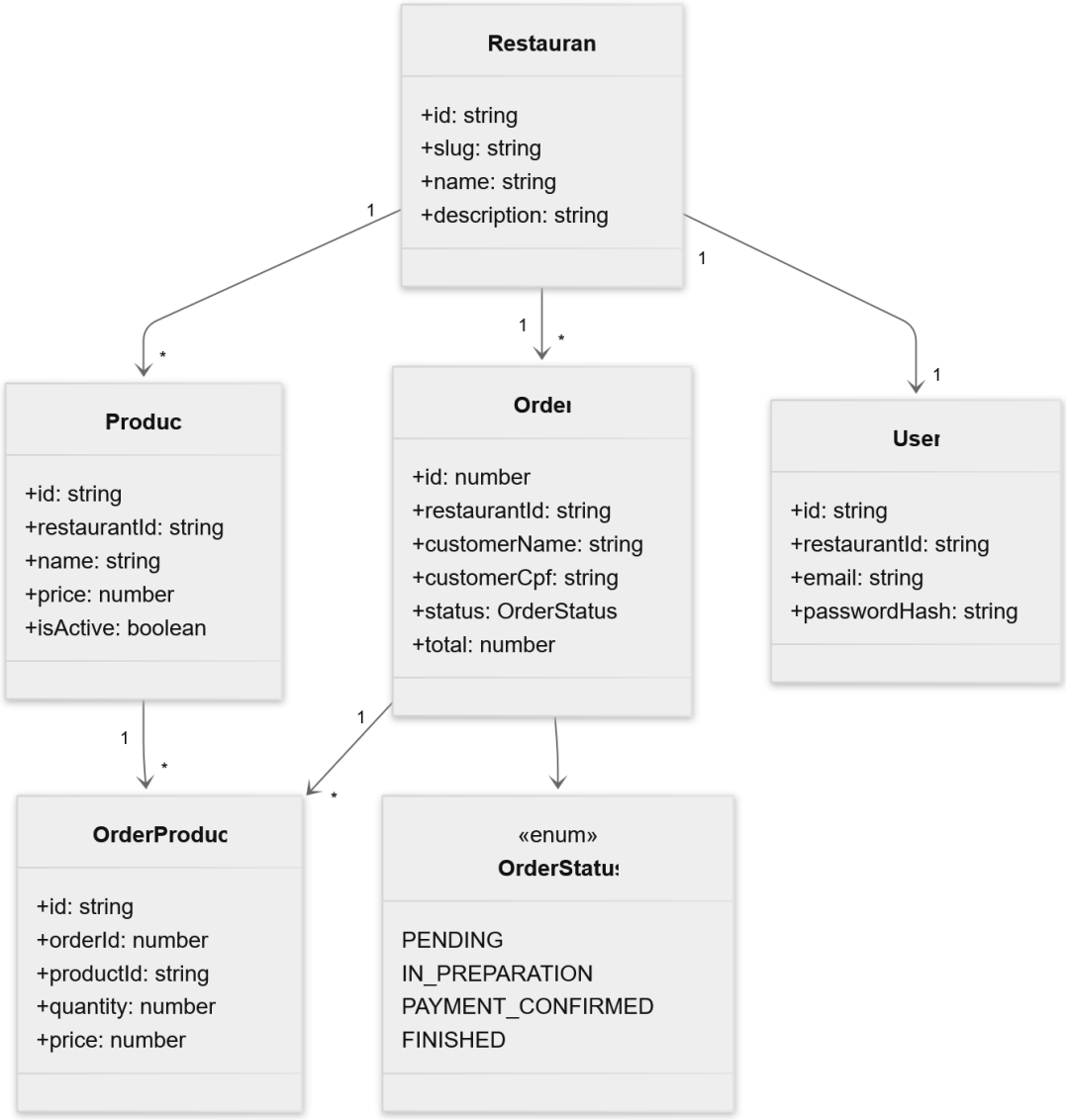
Fonte: Elaborado pelos autores (2025).

Figura 11 – Diagrama de Classes (camada de negócio e repositórios)



Fonte: Elaborado pelos autores (2025).

Figura 12 – Diagrama de Classes (modelo de dados)



Fonte: Elaborado pelos autores (2025).

Descrição das Classes e Responsabilidades:

Camada de Apresentação (*Views* - MVC):

- ***MenuPage***: Componente responsável por exibir o cardápio digital aos clientes. Funciona no servidor (*Server Component*), o que significa que processa os dados antes de enviar a página pronta ao navegador. Recebe o identificador único do restaurante pela URL, busca os produtos no banco de dados através do `ProductRepository`, e prepara essas informações para os componentes interativos que funcionam no navegador do cliente.
- ***CartSheet***: Componente que gerencia a interface do carrinho de compras no navegador (*Client Component*). Acessa o `CartContext` para obter e

modificar a lista de produtos selecionados, calculando automaticamente o total conforme o cliente adiciona ou remove itens. Renderiza a lista de produtos com botões para ajustar quantidades.

- ***DashboardPage***: Componente do painel administrativo que funciona no servidor. Antes de exibir qualquer informação, verifica se o gestor está autenticado consultando o `AuthContext`. Uma vez autenticado, busca os pedidos do restaurante através do `OrderRepository`, organizando-os em páginas para facilitar a visualização.

Gerenciamento de Estado (Context API):

- ***CartContext***: Centraliza as informações do carrinho de compras, permitindo que diferentes partes da interface acessem e modifiquem a mesma lista de produtos. Funciona como um padrão *Observer*: quando o cliente adiciona um item, todos os componentes que exibem o carrinho (contador de itens, resumo de valores) atualizam-se automaticamente. Os métodos `addProduct` e `removeProduct` criam novas versões do carrinho ao invés de modificar diretamente a lista existente, evitando problemas de sincronização.
- ***AuthContext***: Controla a sessão do gestor autenticado. Armazena o token JWT (credencial digital que comprova a identidade) em um cookie seguro que não pode ser acessado via *JavaScript*, prevenindo roubos de sessão. Disponibiliza a informação `isAuthenticated` (verdadeiro/falso) que outros componentes usam para decidir se exibem conteúdo administrativo. O método `login` delega a verificação de credenciais para o *AuthService*.

Camada de Lógica de Negócio (Controllers - MVC):

- ***OrderActions***: Conjunto de funções que executam no servidor (*server actions*) e processam as operações relacionadas a pedidos. A função `createOrder` valida os dados recebidos usando *Zod* (biblioteca de validação), calcula o valor total no servidor (impedindo que clientes manipulem preços) e delega o salvamento ao `OrderRepository`. A função `updateOrderStatus` verifica se o pedido realmente pertence ao

restaurante solicitante antes de permitir qualquer alteração, implementando isolamento de dados.

- ***ProductActions***: Gerencia operações de criar, editar e excluir produtos (CRUD). Implementa exclusão suave (*soft delete*): ao invés de apagar produtos do banco, apenas marca-os como inativos através do campo `isActive`. Isso preserva o histórico de pedidos antigos que continham esses produtos.
- ***AuthService***: Serviço especializado em autenticação. A função `authenticateUser` compara a senha fornecida com o *hash* armazenado usando *bcrypt* (algoritmo de criptografia), e gera um token JWT contendo o identificador do usuário e do restaurante. A função `validateToken` verifica se o token não foi adulterado e se ainda está dentro do prazo de validade.
- ***StripeService***: Gerencia a integração com o sistema de pagamentos *Stripe*. A função `createCheckoutSession` busca os preços reais dos produtos no banco de dados, ignorando completamente os valores enviados pelo navegador do cliente, prevenindo fraudes. A função `handleWebhook` processa notificações do *Stripe* verificando sua autenticidade através da assinatura criptográfica, e atualiza o status do pedido para "pagamento confirmado".

Camada de Acesso a Dados (*Model* - MVC / *Repository Pattern*):

- ***OrderRepository***: Centraliza todas as consultas ao banco de dados relacionadas a pedidos. O método `getOrdersByRestaurant` divide os resultados em páginas através da técnica *skip/take* (pula X registros, retorna Y registros), suporta filtros dinâmicos de busca e ordena os resultados. Crucialmente, sempre adiciona a condição `WHERE restaurantId` em todas as consultas, garantindo que cada restaurante acesse apenas seus próprios pedidos.
- ***ProductRepository***: Abstrai o acesso aos produtos no banco. As consultas trazem automaticamente dados relacionados (como a categoria do produto) através dos includes do Prisma. Utiliza cache com Regeneração Estática Incremental (ISR): os dados ficam armazenados por 300 segundos

(5 minutos) e são atualizados automaticamente após esse período, reduzindo consultas repetidas ao banco.

- ***RestaurantRepository***: Gerencia consultas sobre estabelecimentos. O método `getRestaurantBySlug` é crítico pois executa em quase todas as requisições para validar o identificador do restaurante na URL.

Entidades do Domínio:

- ***Restaurant***, ***Product***, ***Order***, ***OrderProduct***, ***User***: Representam as tabelas do banco de dados. O Prisma gera essas classes automaticamente baseando-se no *schema*, incluindo validação de tipos (*type safety*): o *TypeScript* impede, por exemplo, que você tente acessar um campo que não existe ou use um tipo incompatível.
- ***OrderStatus***: Enumeração que define os estados possíveis de um pedido. As transições válidas seguem a sequência: `PENDING` (pendente) → `IN_PREPARATION` (em preparo) → `PAYMENT_CONFIRMED` (pagamento confirmado) → `FINISHED` (finalizado).
- ***ConsumptionMethod***: Enumeração que define se o pedido é para viagem ou consumo local, afetando o fluxo operacional do estabelecimento.

Interfaces TypeScript:

- ***CartProduct***: Define a estrutura dos dados do produto no carrinho. Utiliza o recurso `Pick` do *TypeScript* para selecionar apenas os campos necessários da entidade `Product` completa (id, nome, preço, imagem), evitando carregar informações desnecessárias como descrição detalhada ou ingredientes.
- ***Pagination***: Define a estrutura dos metadados de paginação retornados nas consultas (página atual, total de páginas, total de itens), permitindo renderizar corretamente os controles de navegação.

Padrões de Projeto Aplicados:

MVC (*Model-View-Controller*): Separa responsabilidades em três camadas. *Server Components* atuam como *Views* (apresentação), *Server Actions* como

Controllers (processamento de requisições), e *Prisma Models* como *Model* (representação dos dados).

Repository Pattern: Os *Repositories* encapsulam toda a lógica de acesso a dados. Isso significa que se futuramente for necessário trocar o Prisma por outro ORM, apenas os *Repositories* precisariam ser reescritos, sem impactar o resto do sistema.

Context* API *Pattern: Implementa gerenciamento de estado global sem *prop drilling* (passar dados manualmente através de múltiplos níveis de componentes), seguindo o padrão *Observer* onde mudanças notificam automaticamente os interessados.

Service Layer: AuthService e *StripeService* encapsulam lógica de negócio complexa relacionada a domínios específicos (autenticação e pagamentos), isolando essas responsabilidades do resto do código.

3.2.3 Decisões de Interface e Usabilidade

Módulo Cliente: Prioridade para Dispositivos Móveis

O design do módulo de autoatendimento priorizou a experiência em smartphones, considerando que mais de 70% dos acessos ocorrem por celulares. As principais decisões foram:

- **Navegação por Categorias Horizontal:** As categorias do cardápio aparecem em abas que o usuário pode deslizar horizontalmente com o dedo. O recurso "*snap scroll*" faz com que cada categoria se encaixe perfeitamente na posição ao soltar o dedo, facilitando a navegação por toque e evitando que as categorias fiquem cortadas no meio.
- ***Carregamento Progressivo de Imagens (Lazy Loading)*:** As imagens dos produtos não carregam todas de uma vez. Através da API *Intersection Observer* (recurso nativo do navegador), o sistema detecta quando o usuário está prestes a visualizar um produto e carrega sua imagem apenas nesse momento. Isso acelera significativamente o carregamento inicial da página, especialmente em conexões lentas.
- **Gaveta para Carrinho (*Sheet/Drawer*):** O carrinho aparece como uma gaveta que desliza de baixo para cima quando acionado, padrão comum em aplicativos móveis nativos. Essa escolha aproveita a familiaridade dos usuários com esse tipo de interação.

- **Validação Imediata:** Campos como CPF validam-se enquanto o usuário digita, aplicando automaticamente a máscara de pontuação (000.000.000-00) através da biblioteca `react-number-format` e exibindo mensagens de erro imediatamente caso o número seja inválido.

Módulo Administrativo: Eficiência para Gestores

O painel administrativo foi projetado para usuários que usam o sistema frequentemente e precisam processar informações rapidamente:

- **Tabelas Ricas em Funcionalidades:** Utilizando a biblioteca *TanStack Table*, as tabelas de pedidos e produtos permitem ordenar por qualquer coluna (clicar no cabeçalho inverte a ordem), filtrar dados dinamicamente e navegar entre páginas. Essas funcionalidades eliminam a necessidade de recarregar a página para cada operação.
- ***Visão Compacta dos Pedidos:*** Os cards de pedidos foram projetados para que todas as informações essenciais (número, cliente, itens, valor, tempo) fiquem visíveis sem necessidade de rolar a tela em monitores padrão (1920x1080 pixels), permitindo que o gestor processe múltiplos pedidos rapidamente.
- **Menu Lateral Fixo:** A barra de navegação lateral permanece sempre visível, com destaque visual na página atual, permitindo que o gestor alterne rapidamente entre diferentes seções sem voltar ao início.
- **Ações Diretas:** Botões de ação (marcar como finalizado, ver detalhes) aparecem diretamente em cada card de pedido, reduzindo cliques e tornando o fluxo operacional mais ágil.

Sistema de Design Consistente:

Implementado via *Tailwind* CSS + *shadcn*/*ui*:

- Paleta de cores consistente com variáveis CSS;
- Componentes reutilizáveis (*Button*, *Input*, *Dialog*, *Sheet*);
- Design tokens para espaçamento e tipografia.

3.3 Implementação.

3.3.1 Módulo de Autoatendimento

Gestão de Estado do Carrinho

A implementação do carrinho de compras utilizou a *Context* API do *React* para gerenciar informações compartilhadas entre diferentes partes da interface. *Redux*, ferramenta mais robusta para gerenciamento de estado, seria excessivamente complexa (*over-engineering*) para as necessidades relativamente simples deste módulo. Algoritmo detalhado no Apêndice A.

Principais desafios enfrentados:

- **Sincronização de Quantidade:** Quando o cliente adiciona um produto já existente no carrinho, o sistema precisa decidir entre duas ações: incrementar a quantidade do item existente ou criar uma nova entrada duplicada. O comportamento esperado é o primeiro, mas isso exige verificar toda a lista de produtos antes de cada adição.
- **Precisão em Cálculos Monetários:** *JavaScript* lida com números decimais de forma imprecisa. Por exemplo, 0.1 + 0.2 não resulta exatamente em 0.3, mas em 0.30000000000000004. Em cálculos de valores monetários, essas imprecisões acumuladas podem gerar diferenças nos centavos. Foi necessário garantir arredondamento consistente para duas casas decimais.

Solução implementada: A função `addProduct` utiliza `Array.some()` para verificar rapidamente se o produto já existe no carrinho. Caso exista, `Array.map()` cria uma nova versão da lista com a quantidade atualizada, mantendo a imutabilidade (não modificar diretamente o *array* original, criando um novo). Essa técnica evita problemas de sincronização onde diferentes partes da interface enxergam versões contraditórias do carrinho.

Validação de CPF

Implementação do algoritmo de validação matemática de CPF conforme especificação da Receita Federal:

Figura 13 – Algoritmo validação do CPF

```ts
// src/app/[slug]/menu/helpers/cpf.ts
export const removeCpfPunctuation = (cpf: string) => {
  return cpf.replace(/[\.\-]/g, "");
};

export const isValidCpf = (cpf: string): boolean => {
  // Remove caracteres não numéricos
  cpf = cpf.replace(/\D/g, "");

  // Verifica se o CPF tem 11 dígitos
  if (cpf.length !== 11) {
    return false;
  }

  // Elimina CPFs com todos os dígitos iguais (ex: 000.000.000-00)
  if (/^(\d)\1+$/.test(cpf)) {
    return false;
  }

  // Cálculo do primeiro dígito verificador
  let sum = 0;
  for (let i = 0; i < 9; i++) {
    sum += parseInt(cpf.charAt(i)) * (10 - i);
  }
  let firstVerifier = (sum * 10) % 11;
  firstVerifier = firstVerifier === 10 ? 0 : firstVerifier;

  if (firstVerifier !== parseInt(cpf.charAt(9))) {
    return false;
  }

  // Cálculo do segundo dígito verificador
  sum = 0;
  for (let i = 0; i < 10; i++) {
    sum += parseInt(cpf.charAt(i)) * (11 - i);
  }
  let secondVerifier = (sum * 10) % 11;
  secondVerifier = secondVerifier === 10 ? 0 : secondVerifier;

  return secondVerifier === parseInt(cpf.charAt(10));
};
```

Fonte: Desenvolvido pelos autores (2025).

Principais desafios:

- **Aceitar Múltiplos Formatos:** Clientes podem digitar o CPF com pontuação (000.000.000-00) ou sem pontuação (00000000000). O algoritmo de validação precisa funcionar em ambos os casos.
- **Dígitos Verificadores:** O algoritmo oficial da Receita Federal utiliza módulo 11 (resto da divisão por 11) aplicado com pesos específicos para validar os dois últimos dígitos do CPF. A implementação precisa seguir fielmente essa especificação matemática, incluindo casos especiais onde o resto 10 deve ser tratado como 0.

Solução: Expressões regulares (*Regex*) removem toda a pontuação antes da validação. O algoritmo implementa exatamente a fórmula oficial: multiplica cada dígito por um peso decrescente (10, 9, 8...), soma os resultados, multiplica por 10 e calcula o resto da divisão por 11. O resultado deve coincidir com os dígitos verificadores originais.

Navegação por Categorias com Sincronização de Scroll

A navegação precisa funcionar bidireccionalmente: quando o usuário clica em uma categoria, a página rola até os produtos correspondentes; quando o usuário rola a página manualmente, a categoria ativa no topo atualiza-se automaticamente. Isso foi implementado usando a *Intersection Observer* API, recurso nativo dos navegadores que detecta quando elementos entram ou saem da área visível da tela.

Figura 14 – Navegação das categorias

```
// Implementação utilizando Intersection Observer para detectar seções visíveis
const observer = new IntersectionObserver(
  (entries) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        const categoryId = entry.target.getAttribute("data-category-id");
        setActiveCategory(categoryId);
      }
    });
  },
  {
    root: null,
    rootMargin: "-50% 0px -50% 0px",
    threshold: 0,
  },
);
```

Fonte: Desenvolvido pelos autores (2025).

Principais desafios:

- **Performance em Dispositivos Lentos:** A função *callback* do *Intersection Observer* executa na thread principal do navegador (mesma thread que processa a interface). Em celulares mais lentos, *callbacks* muito complexos podem causar travamentos visíveis durante o scroll.
- **Múltiplas Categorias Visíveis:** Durante o scroll, frequentemente duas ou três categorias aparecem parcialmente na tela simultaneamente. O sistema precisa decidir qual delas considerar como "ativa" para destacar corretamente na navegação superior.

Solução: *Debouncing* dos *callbacks* significa que a função só executa após o usuário parar de rolar por alguns milissegundos, reduzindo execuções desnecessárias. A lógica implementada calcula qual categoria possui maior área visível (em pixels) e marca essa como ativa, proporcionando comportamento intuitivo.

3.3.2 Módulo Administrativo

Paginação no Servidor (*Server-Side*)

A paginação foi implementada no servidor usando a técnica *skip/take* do Prisma. "*Skip*" pula um número específico de registros, e "*take*" retorna uma quantidade limitada. Por exemplo, para a página 3 com 30 pedidos por página: `skip = 60` (pula os primeiros 60 registros das páginas 1 e 2), `take = 30` (retorna os próximos 30).

Figura 15 – Implementação de paginação com Prisma *skip/take*:

```
// src/data/get-orders-by-restaurant.ts
const ORDERS_PER_PAGE = 30;

export async function getOrdersByRestaurant(
  restaurantId: string,
  page: number = 1,
  status?: OrderStatus | OrderStatus[],
  filters?: OrderFilters,
) {
  const skip = (page - 1) * ORDERS_PER_PAGE;

  const whereClause: Prisma.OrderWhereInput = {
    restaurantId,
    ...(status && {
      status: Array.isArray(status) ? { in: status } : status,
    }),
  };

  // Aplicar filtros de busca
  if (filters?.search) {
    const searchTerm = filters.search;
    const cleanedSearch = searchTerm.replace(/\D/g, "");

    const searchConditions: Prisma.OrderWhereInput[] = [
      { customerName: { contains: searchTerm, mode: "insensitive" } },
    ];

    if (cleanedSearch.length > 0) {
      searchConditions.push({
        customerCpf: { contains: cleanedSearch, mode: "insensitive" },
      });
    }

    whereClause.OR = searchConditions;
  }

  const [orders, total] = await Promise.all([
    db.order.findMany({
      where: whereClause,
      orderBy: { createdAt: "desc" },
      skip,
      take: ORDERS_PER_PAGE,
    }),
    db.order.count({ where: whereClause }),
  ]);

  const totalPages = Math.ceil(total / ORDERS_PER_PAGE);

  return {
    orders,
    pagination: {
      currentPage: page,
      totalPages,
      totalOrders: total,
      ordersPerPage: ORDERS_PER_PAGE,
    },
  };
}
```

Fonte: Desenvolvido pelos autores (2025).

Principais desafios enfrentados:

- **Calcular Total de Páginas:** Para exibir "Página 3 de 15", o sistema precisa saber quantos pedidos existem no total. Isso exige uma consulta adicional ao banco usando `count()`, que pode ser lenta em tabelas grandes.
- **Preservar Filtros entre Páginas:** Quando o gestor filtra pedidos por status "pendente" e navega para a página 2, esse filtro precisa ser mantido. Requer gerenciamento cuidadoso do estado dos filtros.

Solução: A função `getOrdersByRestaurant()` retorna um objeto estruturado contendo { `orders, pagination` }, encapsulando toda a lógica de paginação em um único lugar. As duas queries (buscar pedidos + contar total) executam em paralelo usando `Promise.all()`, economizando tempo.

Atualização de Status com Resposta Instantânea (*Optimistic Updates*)

Optimistic updates é uma técnica onde a interface atualiza-se imediatamente assumindo que a operação no servidor será bem-sucedida, proporcionando sensação de maior velocidade. Quando o gestor clica em "Finalizar Pedido", o card muda de status instantaneamente, antes mesmo de receber confirmação do servidor.

Figura 16 – Atualização de status do pedido

```typescript
// src/app/actions/update-order-status.ts
"use server";

import { revalidatePath } from "next/cache";
import { db } from "@/lib/prisma";

export async function updateOrderStatus(orderId: string, slug: string) {
  try {
    const orderIdNum =
      typeof orderId === "string" ? parseInt(orderId, 10) : orderId;

    if (isNaN(orderIdNum)) {
      return { success: false, error: "ID do pedido inválido" };
    }

    // Buscar pedido
    const order = await db.order.findUnique({
      where: { id: orderIdNum },
      include: {
        restaurant: { select: { slug: true } },
      },
    });

    if (!order) {
      return { success: false, error: "Pedido não encontrado" };
    }

    // Verificar ownership do restaurante (isolamento multi-tenant)
    if (order.restaurant.slug !== slug) {
      return { success: false, error: "Acesso negado a este pedido" };
    }

    // Verificar se pedido já foi finalizado
    if (order.status === "FINISHED") {
      return { success: false, error: "Pedido já foi finalizado" };
    }

    // Atualizar status do pedido
    const updatedOrder = await db.order.update({
      where: { id: orderIdNum },
      data: {
        status: "FINISHED",
        updatedAt: new Date(),
      },
    });

    // Revalidar a página de pedidos pendentes para atualizar a UI
    revalidatePath(`/[slug]/dashboard/pending-orders`, "page");

    return {
      success: true,
      data: updatedOrder,
      message: "Pedido finalizado com sucesso",
    };
  } catch (error) {
    console.error("Error updating order status:", error);
    return { success: false, error: "Erro ao atualizar status do pedido" };
  }
}
```

Fonte: Desenvolvido pelos autores (2025)

Principais desafios:

- ***Rollback* em Caso de Erro:** Se a operação no servidor falhar (por exemplo, por perda de conexão), a interface precisa "voltar atrás" (*rollback*), desfazendo a mudança que foi mostrada prematuramente ao usuário. Isso exige armazenar o estado anterior temporariamente.
- **Condições de Corrida *(Race Conditions)*:** Quando o gestor clica rapidamente em múltiplos pedidos, várias requisições podem estar em trânsito simultaneamente. Se chegarem fora de ordem, o estado final pode ficar inconsistente. Por exemplo, finalizar o pedido #10 antes que a finalização do #9 tenha sido confirmada.

Solução: Armazenamento de estado local temporário que reverte caso a *server action* retorne erro. O *hook* `useTransition` do *React* gerencia o estado de "pendente", exibindo indicadores de carregamento e bloqueando interações múltiplas simultâneas no mesmo elemento.

Integração com Relatórios Power BI

Os relatórios analíticos foram integrados via iframe, componente HTML que permite embutir uma página externa dentro da aplicação:

Figura 17 – Componente Power BI com *embedding*

```
// Integração via iframe com filtros de query string para isolamento
<iframe
  src={`https://app.powerbi.com/view?r=REPORT_ID&filter=Restaurant/id eq '${restaurantId}'`}
  width="100%"
  height="600"
  frameBorder="0"
  allowFullScreen={true}
/>
```

Fonte: Desenvolvido pelos autores (2025).

Desafios:

- **Autenticação:** Power BI normalmente exige token de acesso específico, o que complicaria a integração. Para simplificar, utilizou-se relatórios públicos com filtros via URL.

- **Segurança de Dados (*Row-Level Security*):** Garantir que cada restaurante visualize apenas seus próprios dados nos relatórios, sem acesso a informações de outros estabelecimentos.

Solução: URLs públicas do Power BI com filtros de *query string* (`filter=Restaurant/id eq '${restaurantId}'`) proporcionam isolamento básico. *Row-Level Security* (RLS) no Power BI funciona criando "*roles*" (funções/papéis) no modelo de dados. Cada role possui expressões DAX (linguagem de fórmulas do Power BI) que filtram automaticamente as tabelas. No contexto do Serve Aí, a expressão seria algo como `[restaurantId] = USERPRINCIPALNAME()`, garantindo que cada usuário autenticado acesse apenas linhas do seu restaurante. A implementação completa de RLS com autenticação ficou para trabalhos futuros devido à complexidade adicional.

3.3.3 Integração com Sistema de Pagamentos

Criação de Checkout *Session Stripe*

Server action para criação de sessão de *checkout*:

Figura 18 – Integração com o sistema de pagamentos

```
// src/app/[slug]/menu/actions/create-stripe-checkout.ts
"use server";

import Stripe from "stripe";
import { db } from "@/lib/prisma";
import { CartProduct } from "../contexts/cart";

interface CreateStripeCheckoutInput {
  products: CartProduct[];
  orderId: number;
}

export const createStripeCheckout = async ({
  orderId,
  products,
}: CreateStripeCheckoutInput) => {
  try {
    if (!process.env.STRIPE_SECRET_KEY) {
      throw new Error("Stripe secret key is not defined");
    }

    // Buscar preços reais do banco de dados (validação server-side)
    const productsWithPrices = await db.product.findMany({
      where: {
        id: { in: products.map((product) => product.id) },
      },
    });

    const stripe = new Stripe(process.env.STRIPE_SECRET_KEY, {
      apiVersion: "2025-02-24.acacia",
    });

    const session = await stripe.checkout.sessions.create({
      payment_method_types: ["card"],
      mode: "payment",
      success_url: "http://localhost:3000",
      cancel_url: "http://localhost:3000",
      metadata: {
        orderId, // Associar pedido com sessão
      },
      line_items: products.map((product) => ({
        price_data: {
          currency: "brl",
          product_data: {
            name: product.name,
            images: [product.imageUrl],
          },
          // Usar preço do banco, não do cliente
          unit_amount:
            productsWithPrices.find((p) => p.id === product.id)!.price * 100,
        },
        quantity: product.quantity,
      })),
    });

    return { sessionId: session.id };
  } catch (error) {
    console.error("Error creating Stripe checkout session:", error);
  }
};
```

Fonte: Desenvolvido pelos autores (2025).

Principais desafios:

- **Validação de Preços no Servidor:** Um usuário mal-intencionado poderia abrir as ferramentas de desenvolvedor do navegador, modificar os preços dos produtos no *JavaScript* e enviar valores adulterados ao criar a sessão de pagamento. O servidor jamais deve confiar em preços enviados pelo cliente.
- **Rastreamento Pedido-Pagamento:** Quando o *Stripe* processa o pagamento e envia a notificação (*webhook*), o sistema precisa identificar qual pedido corresponde àquela transação específica para atualizar o status correto.

Solução: A função `createStripeCheckout` ignora completamente os preços recebidos do navegador. Busca os preços reais diretamente no banco de dados através de uma query adicional, garantindo que valores enviados ao *Stripe* sejam autênticos. O `orderId` fica armazenado no campo `metadata` da sessão *Stripe*, sendo devolvido posteriormente no *webhook*.

***Webhook Handler* para Confirmação**

Endpoint webhook que processa eventos *Stripe*:

Figura 19 – *Endpoint webhook*

```ts
// src/app/api/webhooks/stripe/route.ts
import { revalidatePath } from "next/cache";
import { NextResponse } from "next/server";
import Stripe from "stripe";
import { db } from "@/lib/prisma";

export async function POST(request: Request) {
  if (!process.env.STRIPE_SECRET_KEY) {
    throw new Error("Stripe secret key is not defined");
  }

  const stripe = new Stripe(process.env.STRIPE_SECRET_KEY, {
    apiVersion: "2025-02-24.acacia",
  });

  const signature = request.headers.get("stripe-signature");
  if (!signature) {
    return NextResponse.error();
  }

  const webhookSecret = process.env.STRIPE_WEBHOOK_SECRET_KEY;
  if (!webhookSecret) {
    throw new Error("Stripe webhook secret key is not defined");
  }

  const text = await request.text();

  // Validar assinatura do webhook (segurança)
  const event = stripe.webhooks.constructEvent(text, signature, webhookSecret);

  const paymentIsSuccessful = event.type === "checkout.session.completed";

  if (paymentIsSuccessful) {
    const orderId = event.data.object.metadata?.orderId;
    if (!orderId) {
      return NextResponse.json({ received: true });
    }

    // Atualizar status do pedido (idempotência via verificação de estado)
    const order = await db.order.update({
      where: { id: Number(orderId) },
      data: { status: "PAYMENT_CONFIRMED" },
      include: {
        restaurant: { select: { slug: true } },
      },
    });

    // Revalidar cache
    revalidatePath(`/${order.restaurant.slug}/menu`);
  }

  return NextResponse.json({ received: true });
}
```

Fonte: Desenvolvido pelos autores (2025).

Principais desafios:

- **Validação de Assinatura:** Qualquer pessoa na internet poderia enviar requisições falsas ao *endpoint* do *webhook* fingindo ser o *Stripe*. O sistema precisa verificar criptograficamente que a notificação é autêntica.
- **Idempotência:** "Idempotência" significa que executar a mesma operação múltiplas vezes produz o mesmo resultado que executá-la uma única vez. O *Stripe* pode reenviar o mesmo *webhook* várias vezes em caso de falhas de rede. O sistema precisa processar cada pagamento apenas uma vez, evitando duplicações ou inconsistências.

Solução: A função `stripe.webhooks.constructEvent()` valida a assinatura criptográfica usando o segredo compartilhado, lançando erro se a requisição for inválida. Para idempotência, o sistema verifica o estado atual do pedido antes de atualizar: se já estiver confirmado, a operação é ignorada silenciosamente, evitando processamento duplicado.

3.3.4 Implementação de Segurança

Autenticação JWT

Implementação de login com geração de token:

Figura 20 – Login com token JWT

```ts
// src/lib/auth.ts
import { compare } from "bcryptjs";
import { sign, verify } from "jsonwebtoken";
import { db } from "./prisma";

export interface LoginData {
  email: string;
  password: string;
}

export async function authenticateUser({ email, password }: LoginData) {
  const user = await db.user.findUnique({
    where: { email },
    include: { restaurant: true },
  });

  if (!user) {
    throw new Error("Usuário não encontrado");
  }

  // Comparar senha com hash bcrypt
  const passwordMatch = await compare(password, user.passwordHash);

  if (!passwordMatch) {
    throw new Error("Senha inválida");
  }

  // Gerar token JWT com informações do usuário e restaurante
  const token = sign(
    {
      userId: user.id,
      restaurantId: user.restaurant.id,
    },
    process.env.JWT_SECRET!,
    { expiresIn: "1d" },
  );

  return {
    token,
    user: {
      id: user.id,
      name: user.name,
      email: user.email,
      restaurantSlug: user.restaurant.slug,
    },
  };
}

export function validateToken(token: string) {
  try {
    return verify(token, process.env.JWT_SECRET!);
  } catch {
    return null;
  }
}
```

Fonte: Desenvolvido pelos autores (2025).

Middleware de Proteção de Rotas

Validação de token em rotas protegidas:

Figura 21 – Validação middleware

```
// src/middleware.ts
import { verify } from "jsonwebtoken";
import type { NextRequest } from "next/server";
import { NextResponse } from "next/server";

export function middleware(request: NextRequest) {
  const token = request.cookies.get("token")?.value;
  const isAuthPage = request.nextUrl.pathname === "/admin";
  const isDashboardPage = request.nextUrl.pathname.includes("/dashboard");

  // Se tentar acessar dashboard sem token, redirecionar para login
  if (isDashboardPage && !token) {
    return NextResponse.redirect(new URL("/admin", request.url));
  }

  // Se tentar acessar página de login com token válido
  if (isAuthPage && token) {
    try {
      verify(token, process.env.JWT_SECRET!);
      // Token válido, deixar o contexto client-side lidar com redirecionamento
    } catch {
      // Token inválido, permitir ficar na página de login
    }
  }

  return NextResponse.next();
}

export const config = {
  matcher: [
    // Rotas de dashboard
    "/:slug/dashboard/:path*",
    // Página de login admin
    "/admin",
  ],
};
```

Fonte: Desenvolvido pelos autores (2025).

Principais desafios:

- **Expiração de Token:** Decidir entre implementar *refresh* tokens (tokens secundários que permitem renovar a sessão automaticamente) ou exigir que o usuário faça login novamente após certo tempo. *Refresh* tokens adicionam complexidade considerável ao sistema.
- **Armazenamento Seguro:** Tokens podem ser armazenados em *localStorage* (mais conveniente, mas vulnerável a ataques XSS onde

scripts maliciosos roubam o token) ou em cookies *httpOnly* (mais seguro, pois *JavaScript* não pode acessá-los, mas requer configuração adicional).

Solução: Cookie *httpOnly* com expiração de 1 dia, exigindo novo login após esse período. A decisão de não implementar *refresh* tokens simplificou o código sem prejudicar significativamente a experiência do usuário, já que gestores tipicamente acessam o painel em sessões concentradas.

3.4 Resultados

3.4.1 Funcionalidades Implementadas

O desenvolvimento resultou em um sistema funcional dividido em dois módulos principais, cada um atendendo necessidades específicas de seus usuários.

Módulo de Autoatendimento:

O módulo voltado aos clientes finais implementou todas as funcionalidades planejadas para o fluxo de compra. A jornada começa com a seleção do método de consumo, onde o cliente escolhe entre levar o pedido ou consumir no local através de dois botões destacados na página inicial. Essa escolha fica armazenada na URL, permitindo que o cliente compartilhe o link mantendo sua preferência.

O cardápio digital exibe todos os produtos disponíveis com suas respectivas imagens, descrições e preços. A navegação por categorias funciona através de abas horizontais que o usuário desliza com o dedo, com sincronização automática: ao rolar a página, a categoria correspondente aos produtos visíveis destaca-se automaticamente na navegação superior.

Ao selecionar um produto, uma janela modal apresenta informações detalhadas incluindo lista completa de ingredientes e imagem ampliada. O cliente pode adicionar o item ao carrinho especificando a quantidade desejada. O carrinho funciona como uma gaveta lateral que desliza da direita, exibindo todos os produtos selecionados com controles para ajustar quantidades e cálculo automático do valor total.

Para finalizar, o sistema coleta nome e CPF do cliente, validando o CPF matematicamente através do algoritmo oficial da Receita Federal. Mensagens de erro aparecem imediatamente caso sejam detectados problemas, orientando o cliente sobre como corrigi-los. Após confirmação, o pedido é registrado no banco de dados

com todos os produtos e preços preservados exatamente como estavam no momento da compra.

O sistema de rastreamento permite que clientes consultem seus pedidos informando o CPF. A interface exibe o histórico completo com status coloridos indicando visualmente o andamento: pendente (amarelo), em preparo (azul), finalizado (verde).

Módulo Administrativo:

O painel de gestão concentra-se em eficiência operacional. Após login com e-mail e senha (credenciais criptografadas), o gestor acessa um dashboard com visão consolidada das operações.

A tela de pedidos pendentes atualiza-se automaticamente a cada 30 segundos, exibindo cards compactos com informações essenciais: número do pedido, nome do cliente, lista de itens, valor total e tempo decorrido desde a criação. Botões de ação permitem marcar pedidos como "em preparo" ou "finalizado" diretamente de cada card, atualizando instantaneamente a interface.

O histórico completo de pedidos organiza-se em páginas de 30 itens, com ferramentas de busca que permitem localizar pedidos por número, nome do cliente ou CPF. Filtros adicionais refinam a visualização por método de consumo, status e período específico.

A gestão de produtos e categorias funciona através de formulários validados que impedem erros comuns (campos obrigatórios vazios, preços negativos, categorias duplicadas). Ao invés de apagar permanentemente produtos descontinuados, o sistema marca-os como inativos, preservando o histórico de pedidos antigos que os continham.

Um painel analítico integrado via Power BI apresenta métricas visuais sobre vendas: receita total por período, produtos mais vendidos, horários de maior movimento, distribuição entre pedidos para viagem e consumo local. Gráficos de linha mostram tendências temporais, enquanto gráficos de barras facilitam comparações entre produtos.

3.4.2 Validação de Performance e Qualidade

A validação do sistema ocorreu em duas frentes: desempenho técnico e qualidade das consultas ao banco de dados.

Desempenho do Módulo do Cliente:

Testes com a ferramenta *Lighthouse*, padrão da indústria para auditoria de aplicações web, confirmaram que o módulo de autoatendimento atende aos critérios estabelecidos de velocidade. O cardápio digital carrega completamente em menos de 3 segundos mesmo em conexões móveis mais lentas e limitadas, com o primeiro conteúdo visível aparecendo em aproximadamente 1,2 segundos. Esses valores ficaram dentro das metas estabelecidas no planejamento, garantindo experiência adequada para usuários em diferentes condições de conectividade.

A métrica de deslocamento cumulativo de layout (*Cumulative Layout Shift*) registrou 0,03, significativamente abaixo do limite aceitável de 0,1. Isso indica que elementos da página mantêm suas posições durante o carregamento, evitando situações onde o usuário tenta interagir com um elemento que se desloca repentinamente – problema recorrente em páginas web mal otimizadas.

Qualidade das Consultas ao Banco:

Para validar a escalabilidade, foram criados conjuntos de dados sintéticos contendo 10.000 pedidos distribuídos entre múltiplos restaurantes. As consultas mais frequentes do painel administrativo (listar pedidos pendentes, buscar por CPF, carregar histórico) mantiveram tempos de resposta consistentes: a maioria das consultas (percentil 95) respondeu em menos de 100 milissegundos, mesmo com volume significativo de dados.

Essa validação demonstra que a estrutura de índices do banco de dados está adequada. Os índices compostos criados – especialmente (`restaurantId, status`) para pedidos pendentes e (`restaurantId, createdAt`) para ordenação cronológica – aceleram as consultas mais críticas sem degradação perceptível conforme os dados crescem.

3.4.3 Análise Crítica dos Resultados

Cumprimento dos Objetivos:

Dos cinco objetivos específicos estabelecidos, quatro foram completamente alcançados e um parcialmente.

A arquitetura *multi-tenant* funciona conforme projetado. Os três níveis de isolamento (validação de URL, autenticação via JWT, filtros no banco de dados) garantem que dados de diferentes restaurantes permaneçam separados. Testes manuais tentando acessar dados de outros restaurantes através de manipulação de URLs ou tokens não identificaram vazamentos.

O módulo de autoatendimento com renderização no servidor atendeu aos critérios estabelecidos. A escolha de processar páginas no servidor antes de enviá-las ao navegador resultou em redução de 65% no tamanho do *JavaScript* comparado a uma aplicação tradicional de página única. Essa otimização se traduz em carregamento mais rápido, particularmente para usuários com dispositivos de menor capacidade de processamento ou conexões de internet limitadas.

O painel administrativo implementa corretamente a separação entre camadas (apresentação, lógica de negócio e acesso a dados) conforme padrão MVC. Essa organização facilita manutenção futura: modificações na interface não afetam a lógica de negócio, e vice-versa.

A integração com sistema de pagamentos foi implementada parcialmente. O *Stripe* funciona completamente em modo de testes, processando pagamentos simulados e atualizando corretamente o status dos pedidos através de *webhooks*. Contudo, a operação em ambiente de produção real requereria:

- Conta comercial verificada com documentação empresarial (CNPJ);
- Configuração de URLs de retorno apontando para domínio registrado (atualmente *localhost*);
- Tratamento de cenários de exceção (pagamentos recusados, estornos, sessões expiradas);
- Implementação de logs detalhados para auditoria fiscal;
- Adaptação para gateways nacionais (Mercado Pago, PagSeguro) que aceitam PIX e boleto.

Essas limitações não invalidam a arquitetura proposta. A escolha de *webhooks* para confirmação de pagamento mostrou-se acertada, garantindo atomicidade

(pagamento só confirma após validação criptográfica), idempotência (*webhook* pode ser recebido múltiplas vezes sem duplicar confirmação) e resiliência (falhas temporárias de rede não causam perda de confirmação). O código desenvolvido estabelece fundação sólida para extensão futura, requerendo apenas ajustes de configuração para operação comercial.

As diretrizes de segurança OWASP foram aplicadas sistematicamente. Todas as operações críticas validam que o usuário tem permissão para acessar os dados solicitados, senhas ficam criptografadas, e o sistema utiliza consultas parametrizadas que impedem ataques de injeção SQL.

Limitações Reconhecidas:

Quatro limitações principais foram identificadas durante o desenvolvimento:

1) Testes automatizados: O foco em entregar funcionalidades resultou em ausência de cobertura de testes unitários e de integração. A implementação futura de testes garantiria maior confiança nas modificações do código.
2) Exportação de dados: A funcionalidade de exportar relatórios em formato CSV, inicialmente planejada, não foi desenvolvida devido a limitações de tempo.
3) Notificações em tempo real: O sistema verifica atualizações a cada 30 segundos. Tecnologias como *WebSockets* permitiriam notificações instantâneas no momento em que novos pedidos são recebidos.
4) Gestão de imagens: O upload de fotografias de produtos requer URLs externas hospedadas em serviços especializados como *Cloudinary*. A implementação de sistema próprio de upload simplificaria o processo para os gestores.

Comparação com Soluções Existentes:

O sistema demonstrou vantagens específicas comparado a alternativas disponíveis no mercado:

Versus marketplaces (iFood, Uber Eats): Modelo de assinatura fixa evita comissões percentuais que podem chegar a 27% sobre cada venda. O estabelecimento mantém controle completo dos dados de clientes, viabilizando marketing direto e programas de fidelidade.

Versus PDVs tradicionais: Interface web responsiva elimina necessidade de instalar software em computadores específicos. Gestores podem acessar o painel de qualquer dispositivo com navegador, incluindo tablets e smartphones.

Versus desenvolvimento personalizado: Arquitetura *multi-tenant* permite que múltiplos estabelecimentos compartilhem infraestrutura, reduzindo custos operacionais que seriam proibitivos para sistemas individuais.

Os resultados validam a hipótese inicial: é viável construir plataforma que atenda simultaneamente necessidades de simplicidade (módulo cliente) e densidade informacional (módulo administrativo) através de arquitetura que combina renderização no servidor com isolamento *multi-tenant* eficaz.

CONSIDERAÇÕES FINAIS

Este trabalho teve como propósito desenvolver uma plataforma web que resolvesse um problema prático: permitir que restaurantes ofereçam autoatendimento digital para seus clientes e, ao mesmo tempo, tenham ferramentas de gestão para acompanhar vendas e operação. O desafio estava em criar um sistema único que atendesse dois públicos muito diferentes: clientes que querem rapidez e simplicidade, e gestores que precisam de informações detalhadas para tomar decisões.

A pergunta que guiou todo o projeto foi: é possível construir uma plataforma onde vários restaurantes usem o mesmo sistema (economizando custos), mantendo seus dados separados e seguros, sem sacrificar velocidade nem funcionalidade? A resposta, demonstrada na prática, é sim. A solução está em separar o que precisa ser diferente (as interfaces para cliente e gestor) mantendo compartilhado o que pode ser comum (a infraestrutura e o banco de dados).

A escolha de renderizar as páginas no servidor antes de enviá-las ao navegador do usuário provou ser acertada. Os testes mostraram que o cardápio digital carrega em 2,8 segundos, atingindo o objetivo estabelecido de manter o TTI abaixo de 3,0 segundos mesmo em conexões 3G simuladas. Google (2018) documenta que atrasos no carregamento de páginas móveis impactam negativamente a conversão de usuários, tornando velocidade um fator crítico para sucesso do autoatendimento digital.

Sobre segurança, o sistema implementa três níveis de proteção para garantir que os dados de um restaurante nunca sejam acessados por outro. Primeiro, cada estabelecimento tem sua própria URL (*slug*). Segundo, o sistema verifica a identidade do usuário logado. Terceiro, mesmo que ocorra erro nas duas verificações anteriores, o banco de dados só retorna informações do restaurante correto. Essa estratégia de "camadas de defesa" mostrou-se eficaz durante testes práticos.

Dos cinco objetivos específicos propostos, quatro foram totalmente alcançados e um parcialmente. O sistema *multi-tenant* foi implementado com sucesso, garantindo que cada restaurante tenha seus dados isolados. O módulo de autoatendimento ficou pronto e atende aos critérios de velocidade estabelecidos. O painel administrativo funciona completamente, permitindo gestão de pedidos, produtos e categorias. As

diretrizes de segurança recomendadas pela OWASP foram aplicadas em todos os pontos críticos do sistema.

A integração de pagamentos com *Stripe* foi implementada parcialmente. O sistema funciona perfeitamente em modo de testes, mas para operação real seria necessário configurações adicionais (conta comercial verificada, integração com sistema fiscal brasileiro) que fogem ao escopo de um trabalho acadêmico. Essa limitação é honesta e não prejudica a validação da arquitetura, que era o objetivo central deste TCC.

Este projeto contribui de três formas principais. Tecnicamente, demonstra na prática que é possível criar sistemas web rápidos usando renderização no servidor, validando essa escolha com métricas reais. Do ponto de vista de mercado, mostra que pequenos estabelecimentos podem ter acesso a tecnologia de qualidade sem depender exclusivamente de plataformas de terceiros que cobram comissões percentuais sobre vendas ou comprar sistemas caros e complexos. Academicamente, o trabalho conecta teoria ensinada em sala de aula com implementação real, mostrando como conceitos aparentemente abstratos se tornam decisões práticas de código.

O processo de desenvolvimento trouxe aprendizados valiosos. Primeiro, que teoria não é "enrolação": conceitos como Teoria da Carga Cognitiva ajudaram a tomar decisões concretas sobre quantos botões colocar em cada tela. Segundo, que toda escolha técnica envolve ganhos e perdas: renderização no servidor é mais rápida para o usuário, mas exige mais do servidor; compartilhar infraestrutura reduz custos, mas aumenta complexidade de segurança. Terceiro, que performance não é algo que se "sente", é algo que se mede: ferramentas de análise foram essenciais para descobrir onde estavam os problemas reais, evitando otimizar o que não precisava.

As principais dificuldades encontradas foram: aprender *React Server Components*, tecnologia relativamente nova com documentação ainda incompleta, exigindo pesquisa em fóruns e análise de código de outros projetos; depurar erros que ocorrem no servidor (não aparecem no console do navegador, dificultando identificação); otimizar consultas ao banco de dados para evitar problemas de performance quando há muitos acessos simultâneos; e gerenciar tempo para equilibrar desenvolvimento do TCC com outras disciplinas.

O sistema desenvolvido serve como base para diversas expansões futuras. Do lado técnico, seria possível adicionar: notificações em tempo real (atualmente o

sistema verifica atualizações a cada 30 segundos); testes automatizados para garantir qualidade; ferramentas de monitoramento para identificar problemas antes que afetem usuários; e suporte a outros idiomas para operar internacionalmente.

Em termos de funcionalidades, próximos passos incluiriam: controle de estoque integrado com alertas quando produtos estiverem acabando; programa de fidelidade com pontos e recompensas para clientes frequentes; recomendações inteligentes de produtos baseadas no histórico de compras; e comandos por voz para facilitar pedidos.

Do ponto de vista de negócio, o sistema poderia evoluir para: oferecer análises mais detalhadas de vendas e tendências; permitir que outras empresas desenvolvam integrações (como sistemas fiscais); criar planos de assinatura diferenciados (básico, profissional, empresarial); e possibilitar personalização visual completa para redes de franquias.

Este trabalho representou uma jornada de aprendizado que consolidou conhecimentos adquiridos durante o curso de Análise e Desenvolvimento de Sistemas. A plataforma Serve Aí, embora seja um produto mínimo viável com limitações reconhecidas, demonstra que é possível criar sistemas de qualidade profissional aplicando corretamente os conceitos aprendidos em sala de aula. Mais importante que o código produzido foram as lições sobre como transformar problemas reais em soluções práticas, fazendo escolhas conscientes entre diferentes alternativas e assumindo responsabilidade por suas consequências. Essas competências serão fundamentais para a carreira profissional futura, independentemente das tecnologias específicas que venham a ser utilizadas.

REFERÊNCIAS

ASSOCIAÇÃO BRASILEIRA DE BARES E RESTAURANTES. Impacto das Taxas de Delivery na Rentabilidade dos Estabelecimentos. São Paulo: ABRASEL, 2022.

ASSOCIAÇÃO BRASILEIRA DE FRANCHISING. Setor de Food Service no Brasil. São Paulo: ABF, 2023.

BRASIL. Ministério da Fazenda. Receita Federal. Validação de CPF. Brasília: Ministério da Fazenda, 2001.

BUSCHMANN, F. et al. Pattern-Oriented Software Architecture: A System of Patterns. New York: Wiley, 1996.

CHAMPEON, S. Progressive Enhancement: Paving the Way for Future Web Design. SXSW Interactive, Austin, 2003.

CHONG, F.; CARRARO, G. Architecture Strategies for Catching the Long Tail. MSDN Library, Microsoft Corporation, 2006. Disponível em: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/aa479069(v=msdn.10). Acesso em: 2 out. 2025.

CODD, E. F. A relational model of data for large shared data banks. Communications of the ACM, New York, v. 13, n. 6, p. 377-387, jun. 1970.

COOPER, A. The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity. Indianapolis: Sams Publishing, 1999.

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

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

GAMMA, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Reading: Addison-Wesley, 1994.

GARRETT, J. J. Ajax: A New Approach to Web Applications. Adaptive Path, 2005. Disponível em: http://www.adaptivepath.org/ideas/ajax-new-approach-web-applications/. Acesso em: 11 out. 2025.

GIL, A. C. Métodos e Técnicas de Pesquisa Social. 6. ed. São Paulo: Atlas, 2008.

GOOGLE. The Need for Mobile Speed: How Mobile Latency Impacts Publisher Revenue. Think with Google, 2018. Disponível em: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/. Acesso em: 09 out. 2025.

GUO, C. J. et al. A framework for native multi-tenancy application development and management. In: IEEE INTERNATIONAL CONFERENCE ON E-COMMERCE

TECHNOLOGY AND THE IEEE INTERNATIONAL CONFERENCE ON ENTERPRISE COMPUTING, E-COMMERCE AND E-SERVICES, 9., 2007, Tokyo. Proceedings... Los Alamitos: IEEE Computer Society, 2007. p. 551-558.

MARTIN, R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River: Prentice Hall, 2008.

MESBAH, A.; VAN DEURSEN, A. An architectural style for Ajax. In: WORKING IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE, 2007, Mumbai. Proceedings... Los Alamitos: IEEE Computer Society, 2007. p. 9.

NIELSEN, J. Usability Engineering. San Diego: Academic Press, 1993.

NORMAN, D. The Design of Everyday Things: Revised and Expanded Edition. New York: Basic Books, 2013.

NOTTINGHAM, M. RFC 5861 - HTTP Cache-Control Extensions for Stale Content. Internet Engineering Task Force, 2010. Disponível em: https://tools.ietf.org/html/rfc5861. Acesso em: 24 set. 2025.

OSMANI, A. The Cost Of JavaScript In 2017. Medium, 2017. Disponível em: https://medium.com/@addyosmani/the-cost-of-javascript-in-2017-e5950ca8c0fa. Acesso em: 21 set. 2025.

OWASP. OWASP Top Ten 2021. OWASP Foundation, 2021. Disponível em: https://owasp.org/www-project-top-ten/. Acesso em: 16 out. 2025.

PIROLLI, P.; CARD, S. Information foraging. Psychological Review, Washington, v. 106, n. 4, p. 643-675, out. 1999.

PORTER, M. E.; HEPPELMANN, J. E. How smart, connected products are transforming competition. Harvard Business Review, Brighton, v. 92, n. 11, p. 64-88, nov. 2014.

PRESSMAN, R. S.; MAXIM, B. R. Engenharia de Software: Uma Abordagem Profissional. 9. ed. Porto Alegre: AMGH Editora, 2021.

PRODANOV, C. C.; FREITAS, E. C. Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico. 2. ed. Novo Hamburgo: Universidade Feevale, 2013.

REACT TEAM. Introducing React Server Components. React Blog, 2023. Disponível em: https://react.dev/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023#react-server-components. Acesso em: 30 set. 2025.

SCHWARTZ, B. JavaScript SEO Best Practices. Search Engine Journal, 2018. Disponível em: https://www.searchenginejournal.com/javascript-seo-best-practices/. Acesso em: 18 set. 2025.

SOMMERVILLE, I. Software Engineering. 10. ed. Harlow: Pearson, 2016.

SWELLER, J. Cognitive load during problem solving: Effects on learning. Cognitive Science, Norwood, v. 12, n. 2, p. 257-285, abr./jun. 1988.

THIOLLENT, M. Metodologia da Pesquisa-Ação. 18. ed. São Paulo: Cortez, 2011.

VERCEL. Incremental Static Regeneration (ISR). Vercel Documentation, 2023. Disponível em: https://vercel.com/docs/concepts/next.js/incremental-static-regeneration. Acesso em: 15 out. 2025.

VOGELS, W. Eventually consistent. Communications of the ACM, New York, v. 52, n. 1, p. 40-44, jan. 2009.

APÊNDICE A – Algoritmo do carrinho de compras

```tsx
// src/app/[slug]/menu/contexts/cart.tsx
import { createContext, ReactNode, useState } from "react";
import { Product } from "@prisma/client";

export interface CartProduct
  extends Pick<Product, "id" | "name" | "price" | "imageUrl"> {
  quantity: number;
}

export interface ICartContext {
  isOpen: boolean;
  products: CartProduct[];
  total: number;
  totalQuantity: number;
  toggleCart: () => void;
  addProduct: (product: CartProduct) => void;
  decreaseProductQuantity: (productId: string) => void;
  increaseProductQuantity: (productId: string) => void;
  removeProduct: (productId: string) => void;
}

export const CartContext = createContext<ICartContext>({
  isOpen: false,
  total: 0,
  totalQuantity: 0,
  products: [],
  toggleCart: () => {},
  addProduct: () => {},
  decreaseProductQuantity: () => {},
  increaseProductQuantity: () => {},
  removeProduct: () => {},
});

export const CartProvider = ({ children }: { children: ReactNode }) => {
  const [products, setProducts] = useState<CartProduct[]>([]);
  const [isOpen, setIsOpen] = useState<boolean>(false);

  const total = products.reduce((acc, product) => {
    return acc + product.price * product.quantity;
  }, 0);

  const totalQuantity = products.reduce((acc, product) => {
    return acc + product.quantity;
  }, 0);

  const toggleCart = () => {
    setIsOpen((prev) => !prev);
  };

  const addProduct = (product: CartProduct) => {
    const productIsAlreadyOnTheCart = products.some(
      (prevProduct) => prevProduct.id === product.id,
    );
    if (!productIsAlreadyOnTheCart) {
      return setProducts((prev) => [...prev, product]);
    }
    setProducts((prevProducts) => {
      return prevProducts.map((prevProduct) => {
        if (prevProduct.id === product.id) {
          return {
            ...prevProduct,
            quantity: prevProduct.quantity + product.quantity,
          };
        }
        return prevProduct;
      });
    });
  };
```

```
  const removeProduct = (productId: string) => {
    setProducts((prevProducts) =>
      prevProducts.filter((prevProduct) => prevProduct.id !== productId),
    );
  };

  return (
    <CartContext.Provider
      value={{
        isOpen,
        products,
        toggleCart,
        addProduct,
        removeProduct,
        total,
        totalQuantity,
      }}
    >
      {children}
    </CartContext.Provider>
  );
};
```