TÍTULO DO TRABALHO

APLICANDO DOMAIN-DRIVEN DESIGN EM UMA ARQUITETURA DE MICROSSERVIÇOS PARA AUTOMAÇÃO RESIDENCIAL

Vinicius Mendes Castro[1]; Nicolas Nojiri[2]; Humberto Patrick Lacerda Ribeiro[3]
[1, 2, 3] Universidade de Uberaba

vinimcastro@edu.uniube.br; humberto.patrick@uniube.br

Resumo

A automação residencial tem se expandido com a popularização de dispositivos inteligentes, mas muitos sistemas ainda são construídos sobre arquiteturas monolíticas que acumulam acoplamento e dificultam a manutenção, a escalabilidade e a integração de novos dispositivos. Este trabalho tem como objeto de estudo a aplicação do Domain-Driven Design (DDD) combinado a uma arquitetura de microsserviços no domínio da automação residencial. O estudo tem como objetivo desenvolver e analisar um protótipo funcional que demonstra ganhos de modularidade, escalabilidade e manutenção em relação a abordagens monolíticas tradicionais. Quanto à metodologia, o trabalho adota uma pesquisa aplicada e experimental, conduzida de forma incremental em sprints quinzenais inspirados no Scrum; o estudo decompõe o domínio em sete bounded contexts independentes , autenticação, sensores, iluminação, segurança, regras, notificações e agendamentos, e implementa cada um como um microsserviço estruturado em quatro camadas. O trabalho desenvolve o back-end em Go, com PostgreSQL, comunicação assíncrona via NATS e interface web em Angular, além de um protótipo físico baseado em ESP32. Como resultado, o estudo obtém um sistema integrado e funcional, denominado Aurora, no qual cada contexto evolui de forma autônoma e os eventos se propagam em tempo real entre os serviços. O trabalho conclui que a combinação de DDD com microsserviços constitui uma abordagem viável e eficaz para tornar sistemas de automação residencial mais modulares, escaláveis e sustentáveis, reduzindo o acoplamento entre funcionalidades e facilitando a incorporação de novos dispositivos.

Palavras-chave: IoT; Casa inteligente; Arquitetura de software; Comunicação orientada a eventos; Go

1 Introdução

O avanço das tecnologias da informação, aliado à popularização de dispositivos inteligentes, tem transformado significativamente o ambiente doméstico, possibilitando o monitoramento, controle e automação de tarefas do cotidiano. Nesse contexto, destacam-se os sistemas de automação residencial, conhecidos como smart homes, que integram sensores, atuadores, assistentes virtuais e serviços automatizados com o objetivo de proporcionar maior conforto, segurança e eficiência energética aos usuários. Com o crescimento desse mercado, surgem também desafios técnicos importantes, especialmente relacionados à integração, escalabilidade e manutenção dos sistemas que compõem esse ecossistema inteligente.

Entretanto, muitos sistemas de automação residencial ainda são desenvolvidos com base em arquiteturas monolíticas e centralizadas, nas quais funcionalidades como iluminação, segurança, climatização e monitoramento são implementadas de forma acoplada. Essa abordagem resulta em dificuldades de manutenção, baixa escalabilidade e limitações na integração de novos dispositivos e protocolos. A presença de componentes heterogêneos como sensores, lâmpadas inteligentes, alarmes, câmeras, fechaduras e hubs de diferentes fabricantes amplifica essa complexidade, tornando crítica a ausência de uma arquitetura clara e modular.

Diante desse cenário, o problema que motiva este trabalho pode ser formulado da seguinte forma: de que maneira a aplicação de Domain-Driven Design, combinada com uma arquitetura de microsserviços, pode melhorar a modularidade, manutenção e escalabilidade de sistemas de automação residencial?

Com base nessa problemática, o objetivo deste trabalho é desenvolver e analisar um protótipo de automação residencial fundamentado nos princípios do Domain-Driven Design (DDD) e na arquitetura de microsserviços, evidenciando ganhos de modularidade, escalabilidade e manutenção do sistema em relação a abordagens monolíticas tradicionais.

A realização deste projeto justifica-se pela necessidade de modernizar as abordagens arquiteturais empregadas em sistemas de automação residencial. A aplicação do DDD permite organizar o sistema em bounded contexts independentes como iluminação, segurança, sensores e notificações, atribuindo regras e responsabilidades específicas a cada módulo e reduzindo o acoplamento entre funcionalidades. A arquitetura de microsserviços, por sua vez, possibilita que cada módulo evolua de forma autônoma, aumentando a resiliência e facilitando a incorporação de novos dispositivos sem comprometer o funcionamento do sistema como um todo. Dessa forma, a solução proposta contribui para tornar os sistemas de automação residencial mais robustos, escaláveis e sustentáveis a longo prazo, alinhando-se às demandas crescentes por ambientes inteligentes confiáveis e de fácil manutenção.

2 Referencial Teórico

2.1 Automação Residencial

A automação residencial consiste na utilização de sensores, atuadores e serviços inteligentes para monitorar e controlar o ambiente doméstico de forma automática. Esses sistemas integram dispositivos heterogêneos, como lâmpadas, fechaduras, câmeras e sensores, cada um com seus próprios padrões de comunicação, o que torna complexo garantir que todos os componentes funcionem de maneira coordenada e escalável.

Siddiqui, Khendek e Toeroe (2023) apontam que sistemas de IoT, nos quais a automação residencial se insere, enfrentam desafios como heterogeneidade de dispositivos, alta conectividade e exigências não funcionais como confiabilidade e disponibilidade. Isso significa que, sem uma arquitetura adequada, a coexistência de componentes distintos gera acoplamento e instabilidade. Esse desafio é central para este trabalho, pois o protótipo proposto precisa integrar múltiplos tipos de dispositivos de forma organizada e escalável.

Rizzardi, Sicari e Coen-Porisini (2025) reforçam que em sistemas de automação residencial baseados em microsserviços surgem também preocupações relacionadas à segurança da comunicação entre os componentes distribuídos. Os autores demonstram que decisões arquiteturais influenciam diretamente atributos como tempo de resposta, uso de memória e consumo de CPU. Isso é relevante para este projeto, pois evidencia que as escolhas arquiteturais têm impacto direto tanto na organização quanto no desempenho do sistema.

2.2 Arquitetura de Microsserviços

A arquitetura de microsserviços é uma abordagem de desenvolvimento de software que divide o sistema em serviços independentes, cada um com uma única responsabilidade. Segundo Newman (2021), sistemas monolíticos apresentam limitações importantes, como alto acoplamento entre módulos, baixa escalabilidade e dificuldade de manutenção. Em outras palavras, quando todas as funcionalidades estão concentradas em um único sistema, qualquer alteração pode impactar o todo. Para o presente trabalho, isso representa exatamente o problema que se busca resolver, uma vez que sistemas tradicionais de automação residencial tendem a ser desenvolvidos de forma centralizada.

Siddiqui, Khendek e Toeroe (2023) destacam que a arquitetura de microsserviços apresenta forte potencial para sistemas de IoT, especialmente na melhoria de atributos não funcionais como confiabilidade e disponibilidade. Ou seja, além de organizar melhor o sistema, essa arquitetura contribui diretamente para a qualidade da solução em ambientes com dispositivos heterogêneos. Isso é importante para este trabalho porque permite que módulos como iluminação, segurança e sensores evoluam de forma independente, sem que a atualização de um comprometa o funcionamento dos demais.

Le, Dang e Vo (2025) acrescentam que arquiteturas de microsserviços orientadas a domínio favorecem atributos de qualidade como resiliência, desempenho e modificabilidade, tornando o sistema mais preparado para crescer com novos dispositivos

e funcionalidades ao longo do tempo. Esse aspecto reforça a adequação dessa arquitetura ao cenário de automação residencial proposto neste projeto, que prevê a inclusão progressiva de novos módulos como climatização e controle por voz.

2.3 Domain-Driven Design

O Domain-Driven Design (DDD) é uma abordagem de modelagem de software centrada na compreensão profunda do domínio do problema. Evans (2004) argumenta que a complexidade de sistemas modernos deve ser tratada por meio de conceitos como entidades, agregados, value objects e bounded contexts, que delimitam áreas específicas do conhecimento e evitam ambiguidades na modelagem. Em outras palavras, o DDD organiza o sistema em torno do problema real que ele resolve, e não apenas em torno de detalhes técnicos. Isso é relevante para este trabalho, pois permite modelar o sistema de automação residencial com clareza, separando responsabilidades entre domínios como iluminação, segurança e sensores.

Özkan, Babur e Van den Brand (2025) confirmam, por meio de uma revisão sistemática da literatura, que o DDD tem sido amplamente adotado como mecanismo de decomposição em sistemas baseados em microsserviços, contribuindo para a estruturação do domínio e a melhoria da manutenibilidade. Ou seja, a escolha do DDD neste projeto está alinhada com as tendências acadêmicas mais recentes da área. Os autores também apontam que um dos principais desafios do DDD é a dificuldade de adoção e a necessidade de conhecimento especializado, o que reforça a importância de documentar explicitamente as decisões de modelagem tomadas neste trabalho.

Vernon (2013) complementa que os princípios do DDD tornam o sistema mais claro, previsível e sustentável, especialmente quando combinado com arquiteturas distribuídas como microsserviços. Hippchen et al. (2017) reforçam que os bounded contexts são utilizados para organizar e identificar microsserviços, criando uma correspondência direta entre os limites do domínio e os limites dos serviços. Para o presente projeto, essa combinação é fundamental, pois garante que cada módulo do protótipo tenha fronteiras bem definidas e regras próprias, reduzindo dependências acidentais e facilitando a evolução do sistema ao longo do tempo.

3 Metodologia e Descrição do sistema

Este trabalho caracteriza-se como uma pesquisa de natureza aplicada, com abordagem experimental, cujo objetivo foi desenvolver e analisar um protótipo funcional de automação residencial fundamentado nos princípios do Domain-Driven Design (DDD) e na arquitetura de microsserviços. O desenvolvimento do sistema Aurora foi conduzido de maneira incremental e iterativa, estruturado em sprints quinzenais inspirados nos princípios ágeis do Scrum. Optou-se pelo ciclo de duas semanas por se mostrar compatível com a natureza distribuída da solução: cada bounded context constitui uma unidade de trabalho coesa, passível de ser projetada, implementada e validada de forma independente dentro de um único sprint, antes de ser integrada ao ecossistema. Essa cadência reduziu riscos de integração e favoreceu a rastreabilidade do progresso ao longo do projeto.

A identificação dos bounded contexts partiu da análise do domínio da automação residencial, decompondo o sistema segundo as capacidades de negócio e as fronteiras naturais de responsabilidade entre elas. A partir desse mapeamento, delimitaram-se sete

contextos independentes, são eles, autenticação, sensores, iluminação, segurança, regras, notificações e agendamentos, cada qual com sua própria linguagem ubíqua, regras de negócio e modelo de dados, de modo a evitar ambiguidades e dependências acidentais entre áreas distintas do domínio. Esse recorte orientou diretamente a divisão dos microsserviços, estabelecendo a correspondência entre os limites do domínio e os limites dos serviços.

Definidas as fronteiras, cada microsserviço foi estruturado internamente em quatro camadas, domínio, aplicação, infraestrutura e inicialização, segundo o critério de isolar a lógica de negócio de qualquer detalhe técnico. A camada de domínio concentra entidades, agregados, value objects e eventos de domínio, sem dependência de bibliotecas externas; a de aplicação orquestra os casos de uso por meio de interfaces; a de infraestrutura implementa as dependências concretas, como handlers HTTP, repositórios e publishers/consumers de mensagens; e a de inicialização conecta os componentes e sobe o serviço. Essa separação tornou cada serviço testável, evolutivo e substituível sem impacto sobre os demais.

Quanto às tecnologias adotadas, o back-end foi desenvolvido integralmente na linguagem Go, com o framework Gin para exposição das APIs REST. A escolha de Go justifica-se por seu modelo de concorrência baseado em goroutines, sua baixa latência e seu alto throughput, características adequadas a um sistema que processa múltiplos eventos simultâneos. Na concepção inicial do projeto, considerou-se ainda o uso da linguagem Rust para os serviços de maior criticidade, no serviço de segurança e motor de regras, em razão de suas garantias de segurança de memória em tempo de compilação. Optou-se, contudo, por desenvolver a totalidade dos microsserviços em Go, a fim de preservar a homogeneidade tecnológica do back-end, simplificar a manutenção e concentrar o esforço de desenvolvimento em uma única linguagem. Cada microsserviço acessa o PostgreSQL como banco de dados relacional, operando sobre seu próprio conjunto de tabelas dentro de um banco compartilhado, uma decisão pragmática para um ambiente residencial embarcado que reduz overhead de infraestrutura sem comprometer a independência lógica entre os contextos. A comunicação assíncrona orientada a eventos é realizada por meio do NATS, escolhido como message broker por seu desempenho e por seu menor custo de infraestrutura em comparação a alternativas como o Apache Kafka ou até mesmo RabbitMQ. O frontend foi construído com Angular 18, Tailwind CSS e PrimeNG, comunicando-se com os serviços por meio de JWT e WebSocket. O Nginx atua como API Gateway, centralizando o roteamento das requisições, e toda a infraestrutura é containerizada e orquestrada com Docker e Docker Compose. O protótipo físico, baseado em ESP32 programado em C++ com as bibliotecas do framework Arduino, complementa a solução com hardware real para a validação dos fluxos de automação.

A comunicação entre os contextos foi definida em três modalidades, conforme a necessidade de cada interação: síncrona, via HTTP/REST, para ações imediatas entre serviços; assíncrona, via NATS PubSub, para a publicação e o consumo de eventos de domínio; e em tempo real, via WebSocket, para a entrega de atualizações ao frontend. O controle de versão foi realizado com Git, em repositório hospedado no GitHub, adotando-se branches separadas por sprint, segundo o padrão de nomenclatura AURORA-%, e integração contínua por meio de pull requests revisados, o que garantiu a rastreabilidade das mudanças e organizou a colaboração ao longo do desenvolvimento.

Por fim, a validação da solução foi conduzida por meio de testes unitários aplicados aos casos de uso de cada um dos sete microsserviços e de testes de integração que verificaram a comunicação entre os serviços. Foram ainda simulados cenários reais de automação, como o acionamento da iluminação a partir da detecção de movimento e o disparo de notificações mediante eventos de segurança, a fim de verificar o comportamento do sistema de ponta a ponta. A documentação das APIs foi gerada automaticamente com Swagger, e os comandos de build, teste e execução foram centralizados em um Makefile.

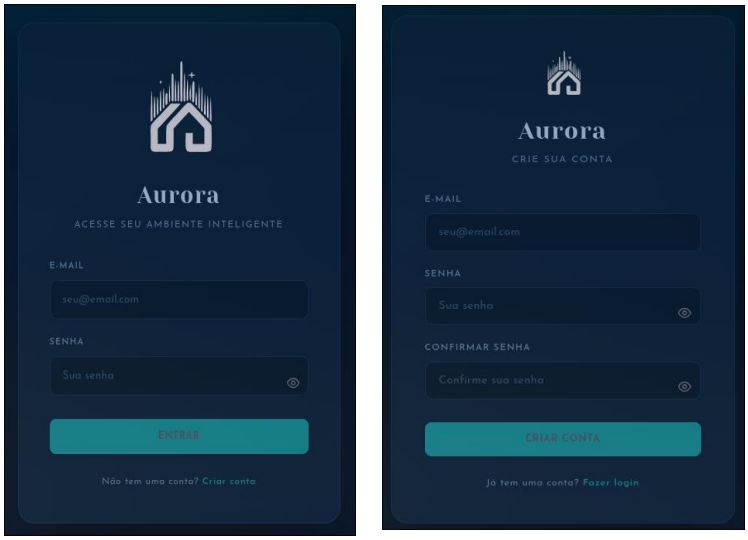
4 Resultados e discussão

Ao término do ciclo de desenvolvimento descrito neste trabalho, o sistema Aurora apresenta um conjunto completo e integrado de funcionalidades de automação residencial, distribuídas entre sete microsserviços independentes, uma interface web e um firmware embarcado para dispositivo IoT.

4.1 Funcionalidades implementadas

O auth-service é responsável pelo cadastro e pela autenticação de usuários. O cadastro exige nome, e-mail e senha, esta última armazenada com hash bcrypt antes de ser persistida no banco de dados. O login retorna um JSON Web Token (JWT) assinado com o algoritmo HS256 e validade de uma hora. Todos os endpoints protegidos do sistema validam esse token por meio de um middleware de autenticação compartilhado, garantindo que cada usuário acesse apenas os recursos de sua própria conta.

Figuras 1 e 2 – Tela de login e cadastro



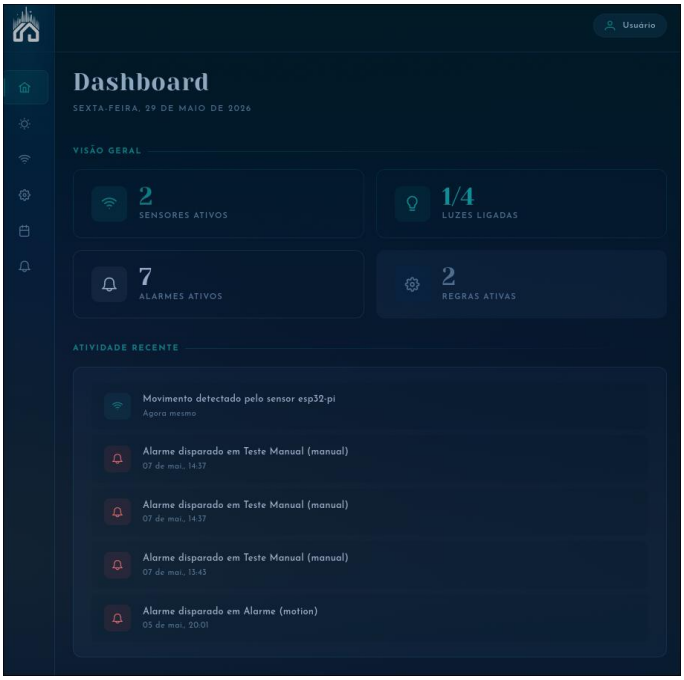
Fonte: Elaborada pelo autor (2026).

A Figura 1 apresenta a tela de autenticação e a Figura 2 a criação de conta do sistema. O usuário informa e-mail e senha para obter um token JWT, armazenado localmente e utilizado em todas as requisições subsequentes pela aplicação Angular, por meio de um interceptor HTTP que injeta automaticamente o cabeçalho de autorização.

O sensors-service centraliza o gerenciamento dos sensores cadastrados pelo usuário, com suporte a sensores de movimento (PIR) e luminosidade (LDR). O serviço expõe uma API REST para consulta de sensores e histórico de leituras e disponibiliza um endpoint WebSocket pelo qual o frontend Angular recebe atualizações em tempo real

sempre que um novo evento de sensor é registrado, sem necessidade de recarregar a página.

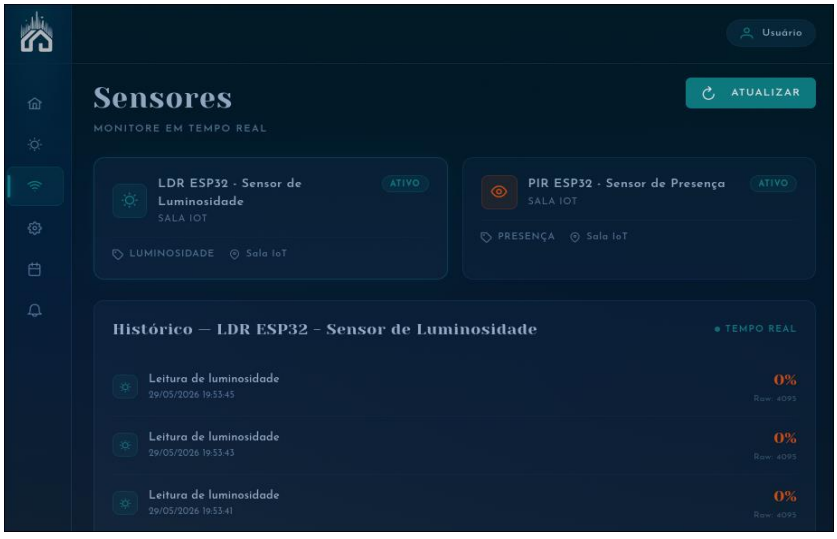
Figura 3 – Dashboard principal



Fonte: Elaborada pelo autor (2026).

A Figura 3 exibe o painel principal do sistema Aurora, que oferece ao usuário uma visão consolidada do estado atual da residência: sensores ativos, estado das lâmpadas, status do alarme e notificações recentes. O painel atualiza automaticamente as informações via WebSocket, sem necessidade de intervenção do usuário.

Figura 4 – Monitoramento de sensores em tempo real

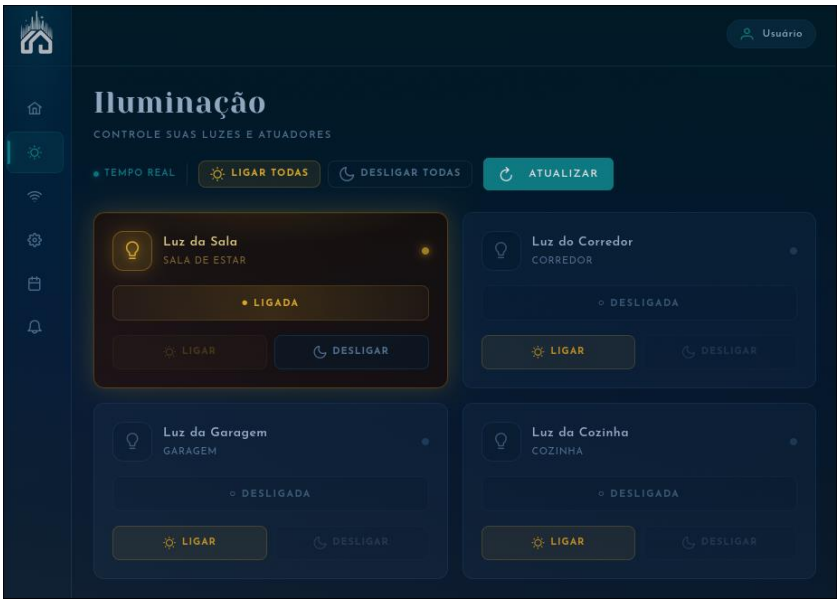


Fonte: Elaborada pelo autor (2026).

A Figura 4 apresenta a tela de sensores, exibindo as leituras mais recentes de movimento e luminosidade transmitidas em tempo real pelo dispositivo ESP32. Cada evento de detecção de movimento aparece instantaneamente na interface por meio da conexão WebSocket estabelecida com o sensors-service.

O lighting-service gerencia o estado das lâmpadas cadastradas por ambiente. O usuário pode ligar e desligar cada dispositivo individualmente pelo painel web. Assim como no serviço de sensores, o estado das lâmpadas é propagado em tempo real via WebSocket para todos os clientes conectados, garantindo que a interface reflita imediatamente qualquer alteração, independentemente de qual cliente a tenha originado.

Figura 5 – Controle de iluminação



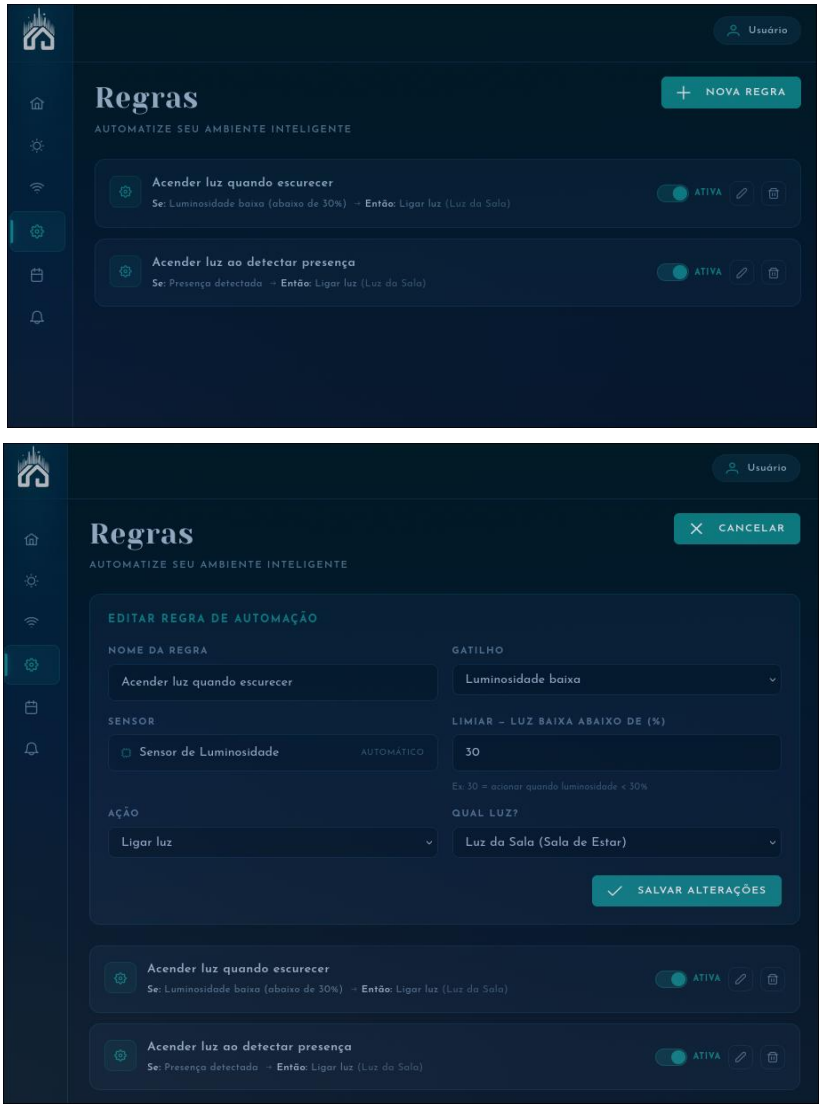
Fonte: Elaborada pelo autor (2026).

A Figura 5 mostra a interface de controle de iluminação, na qual o usuário pode ligar e desligar cada lâmpada cadastrada. O estado de cada dispositivo é sincronizado em tempo real via WebSocket para todos os clientes conectados simultaneamente.

O security-service implementa o gerenciamento do alarme residencial. O usuário pode armar e desarmar o alarme pelo painel web. Eventos de segurança, como ativação e disparo do alarme, são publicados como eventos de domínio no NATS e consumidos pelos serviços interessados, em particular o de notificações, que registra cada ocorrência no histórico do usuário.

O rules-service permite que o usuário crie regras condicionais que definem comportamentos automáticos do sistema. Cada regra é composta por uma condição, por exemplo, a detecção de movimento por um sensor específico e uma ação a ser executada, como ligar uma lâmpada ou armar o alarme. As regras são avaliadas de forma reativa: o serviço assina tópicos de eventos de domínio no NATS e, ao receber um evento, verifica quais regras do usuário possuem condição compatível e dispara as ações correspondentes via chamadas HTTP aos serviços-alvo.

Figura 6 e 7 – Motor de regras de automação

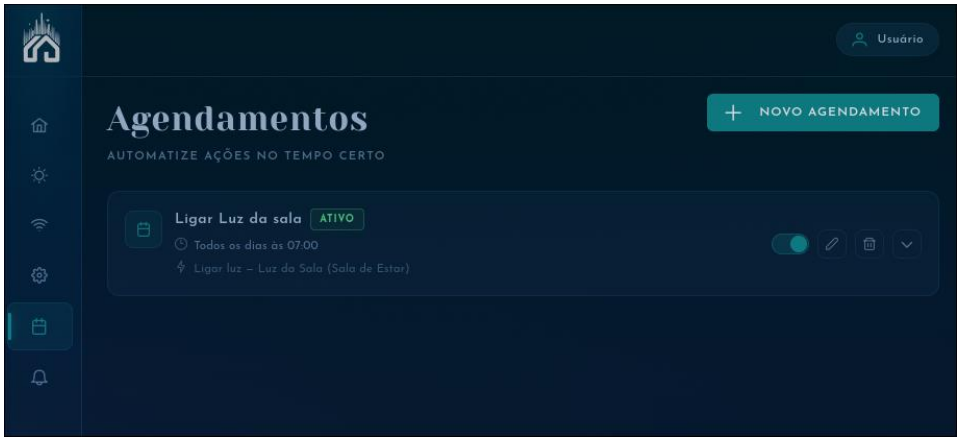


Fonte: Elaborada pelo autor (2026).

A Figura 6 e 7 apresenta a interface de criação, edição e listagem de regras de automação. O usuário define a condição (evento de sensor) e a ação a ser executada (controle de iluminação ou alarme), configurando o comportamento automático do sistema sem necessidade de intervenção manual.

O schedule-service oferece suporte a dois tipos de agendamento: recorrente (cron), baseado em expressões cron para ações que se repetem periodicamente, e pontual (one-shot), para ações programadas para um único momento futuro. Internamente, o serviço utiliza a biblioteca gocron/v2 para executar os agendamentos no horário correto e dispara chamadas HTTP para o lighting-service ou o security-service, conforme configurado pelo usuário. Isso possibilita automações como ligar as luzes externas ao pôr do sol ou armar o alarme automaticamente às 23h.

Figura 8 – Agendamentos de automação

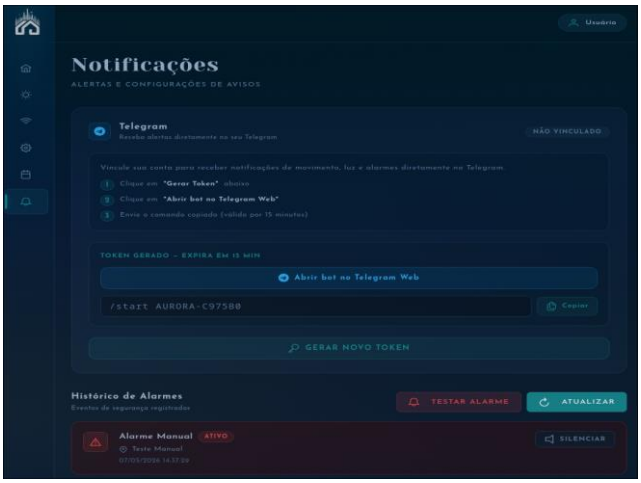


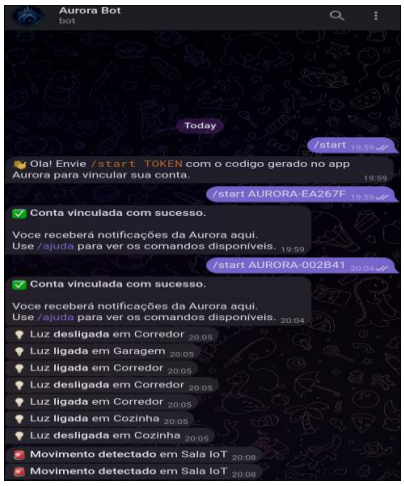
Fonte: Elaborada pelo autor (2026).

A Figura 8 exibe a tela de agendamentos, na qual o usuário cria rotinas recorrentes (cron) ou pontuais (one-shot) para acionar automaticamente dispositivos em horários pré-determinados.

O notifications-service consome eventos publicados pelos demais serviços via NATS , como detecção de movimento, mudança no estado do alarme e execução de regras e persiste um registro de cada ocorrência no banco de dados. O usuário acessa o histórico completo de notificações pelo frontend e via Telegram, permitindo acompanhar tudo o que aconteceu no ambiente residencial mesmo sem a interface aberta.

Figura 9 e 10 – Histórico de notificações





Fonte: Elaborada pelo autor (2026).

A Figura 9 e 10 mostra o histórico de notificações, que registra todos os eventos relevantes do sistema, tanto na interface web quanto via Telegram, quando sincronizado, como detecções de movimento, ativações de alarme e execuções de regras, permitindo ao usuário acompanhar o que ocorreu na residência ao longo do tempo.

O frontend foi desenvolvido como uma Single Page Application (SPA) utilizando Angular 18 com componentes standalone e lazy loading de rotas, Tailwind CSS para estilização e PrimeNG como biblioteca de componentes de interface. A aplicação é servida pelo Nginx, que também atua como API Gateway, roteando as requisições para os respectivos microsserviços. A comunicação com o backend é feita via requisições HTTP autenticadas com JWT e, para dados em tempo real, via WebSocket. O frontend conta com

telas de login e cadastro, dashboard principal, monitoramento de sensores, controle de iluminação, gerenciamento do alarme, criação e listagem de regras, gerenciamento de agendamentos e histórico de notificações.

O dispositivo físico executa um firmware desenvolvido em C++ com o framework Arduino para ESP32. O firmware conecta o hardware ao sistema Aurora via Wi-Fi 2.4 GHz e se comunica com o backend por meio de requisições HTTP autenticadas com um header X-Device-Key. O sensor PIR é lido por interrupção de hardware (ISR), que sinaliza uma flag para a task FreeRTOS dedicada no Core 0, responsável por enviar os eventos de movimento ao sensors-service. O sensor LDR é lido periodicamente pela mesma task, e o valor bruto é convertido em percentual antes do envio. No Core 1, um servidor HTTP embarcado recebe comandos dos microsserviços para ligar e desligar LEDs e o buzzer, sem bloquear as leituras dos sensores.

4.2 Funcionamento do sistema

O funcionamento do Aurora parte da autenticação do usuário no frontend Angular, que obtém um token JWT válido por uma hora. Com o token, o usuário acessa os painéis de sensores, iluminação, alarme, regras, agendamentos e notificações. Cada painel comunica-se com o microsserviço correspondente por meio do Nginx, que atua como API Gateway e roteia todas as requisições para os serviços internos.

Para eventos em tempo real, o fluxo percorre os seguintes passos: o firmware do ESP32, executando no Core 0 via FreeRTOS, detecta um evento, por exemplo, o sensor PIR identifica movimento por interrupção de hardware. A task FreeRTOS envia imediatamente uma requisição HTTP ao sensors-service, que persiste o evento no PostgreSQL e, em seguida, publica um evento de domínio sensor.motion.detected no NATS. O rules-service e o notifications-service, ambos inscritos nesse tópico, recebem o evento de forma assíncrona: o primeiro avalia se alguma regra cadastrada possui condição compatível e, em caso positivo, dispara a ação configurada via HTTP, acionando, por exemplo, uma lâmpada no lighting-service; o segundo registra uma notificação no banco de dados. Simultaneamente, o sensors-service transmite a atualização via WebSocket para o frontend Angular, que exibe o novo evento na tela em tempo real.

Para os agendamentos, o schedule-service mantém em memória os jobs configurados pelo usuário, gerenciados pela biblioteca gocron/v2. No horário determinado, o serviço dispara uma chamada HTTP ao lighting-service ou ao security-service para executar a ação agendada, como desligar todas as luzes à meia-noite ou armar o alarme às 23h. Toda a infraestrutura é containerizada com Docker e orquestrada via Docker Compose. Cada microsserviço é executado em seu próprio container e opera sobre tabelas dedicadas dentro de uma instância PostgreSQL compartilhada, preservando a independência lógica entre os contextos. O NATS opera como message broker central para a comunicação assíncrona, e o Nginx serve o frontend Angular e atua como proxy reverso para os serviços de backend.

4.3 Discussão dos resultados

A arquitetura de microsserviços com DDD demonstrou, na prática, os benefícios esperados em termos de modularidade e isolamento. A definição clara de bounded contexts impediu o acúmulo de dependências entre domínios distintos: o sensors-service não precisa conhecer a existência do rules-service para que as regras sejam avaliadas quando um sensor é ativado, bastando publicar o evento de domínio no NATS. Essa independência tornou possível desenvolver, testar e evoluir cada serviço de forma autônoma ao longo do projeto.

A comunicação assíncrona via NATS mostrou-se eficaz para propagar eventos entre contextos com baixa latência e sem acoplamento direto. Em conjunto com o WebSocket, o sistema alcançou o comportamento em tempo real esperado de um sistema de automação residencial: alterações de estado nos sensores e nas lâmpadas refletem instantaneamente na interface do usuário, sem necessidade de polling. A separação em quatro camadas dentro de cada microsserviço garantiu que a lógica de negócio permanecesse isolada de detalhes técnicos como o framework HTTP ou o banco de dados, tornando o código mais testável e coeso. A escolha de Go para o backend mostrou-se adequada, pois o modelo de concorrência baseado em goroutines tornou natural o tratamento simultâneo de múltiplas conexões WebSocket e eventos NATS sem bloqueio.

Em termos de benefícios, o sistema Aurora demonstra que é possível construir uma solução de automação residencial modular e escalável utilizando tecnologias de código aberto e infraestrutura containerizada. A adição de um novo tipo de sensor ou atuador exige apenas a criação de um novo microsserviço ou a extensão de um existente, sem impactar os demais contextos, evidenciando o benefício da escalabilidade horizontal prometido pela arquitetura de microsserviços.

4.4 Relação com o problema e o objetivo

O problema de pesquisa identificado na introdução deste trabalho aborda a dificuldade de integrar dispositivos heterogêneos em sistemas de automação residencial mantendo modularidade, manutenibilidade e escalabilidade. Sistemas construídos sobre arquiteturas monolíticas tradicionais tendem a acumular acoplamento entre componentes ao longo do tempo, tornando a adição de novas funcionalidades arriscada e a manutenção custosa. O sistema Aurora responde diretamente a esse problema: cada dispositivo e funcionalidade do ambiente residencial, sensores, lâmpadas, alarme, regras, agendamentos e notificações é representado por um contexto delimitado independente, que pode ser alterado, substituído ou expandido sem impactar os demais. A integração do hardware ESP32, por exemplo, ocorre exclusivamente por meio de chamadas HTTP ao sensors-service e ao lighting-service, sem que o firmware precise conhecer qualquer outro componente do sistema.

O objetivo central do trabalho foi desenvolver e analisar um protótipo de automação residencial baseado em Domain-Driven Design e arquitetura de microsserviços, demonstrando melhorias na modularidade, escalabilidade e manutenção do sistema em comparação com abordagens monolíticas tradicionais, sendo plenamente alcançado. O protótipo implementado conta com sete microsserviços em operação, cada um com domínio, conjunto de tabelas e ciclo de vida independentes; uma interface web que integra dados em tempo real de múltiplos serviços; e um dispositivo físico ESP32 integrado ao ecossistema. A arquitetura em quatro camadas aplicada em todos os serviços demonstra, na prática, como o DDD contribui para organizar o código em torno do problema real que o sistema resolve, e não apenas em torno de detalhes técnicos, alinhando-se às conclusões de Özkan, Babur e Van den Brand (2025), que identificaram o DDD como uma abordagem eficaz para reduzir a complexidade em sistemas distribuídos, especialmente quando aplicado em conjunto com arquitetura de microsserviços.

5 Conclusão

Este trabalho surgiu da dificuldade recorrente de integrar, organizar e manter ecossistemas de automação residencial compostos por dispositivos heterogêneos como sensores, lâmpadas, alarmes e demais atuadores, cujos sistemas, quando construídos

sobre arquiteturas monolíticas e centralizadas, tendem a acumular acoplamento ao longo do tempo, comprometendo a manutenibilidade, a escalabilidade e a inclusão de novas funcionalidades. Como resposta a esse problema, foi desenvolvido o sistema Aurora, um protótipo de automação residencial estruturado segundo os princípios do Domain-Driven Design (DDD) e organizado em uma arquitetura de microsserviços, no qual cada área do domínio é tratada como um contexto delimitado independente, com tabelas de banco de dados dedicadas e contratos de comunicação bem definidos.

O objetivo proposto foi alcançado, uma vez que o protótipo desenvolvido demonstrou, na prática, ganhos de modularidade, escalabilidade e manutenção em relação a abordagens monolíticas tradicionais. A modelagem do domínio em sete bounded contexts , autenticação, sensores, iluminação, segurança, regras, notificações e agendamentos permitiu que cada serviço fosse projetado, implementado e validado de forma autônoma, enquanto a arquitetura em quatro camadas (domínio, aplicação, infraestrutura e inicialização) manteve a lógica de negócio isolada de detalhes técnicos. Dessa forma, o sistema contribuiu para resolver o problema ao organizar o código em torno do problema real a ser tratado, e não apenas em torno de aspectos de implementação, reduzindo o acoplamento entre os módulos e respondendo diretamente à questão de pesquisa que motivou o trabalho.

Entre os principais resultados obtidos, destacam-se a implementação completa dos sete microsserviços em Go com o framework Gin, de uma interface web em Angular 18 e de um firmware embarcado para o dispositivo IoT baseado em ESP32. O sistema oferece autenticação e controle de acesso por meio de JSON Web Tokens, monitoramento de sensores em tempo real, controle individual de iluminação, gerenciamento do alarme residencial, um motor de regras de automação reativo, agendamentos recorrentes e pontuais e o registro de notificações de eventos, inclusive via Telegram. A comunicação assíncrona orientada a eventos por meio do NATS, aliada ao uso de WebSocket, permitiu que alterações de estado nos sensores e nas lâmpadas se refletissem instantaneamente na interface, sem necessidade de polling. Como benefício direto da abordagem, a adição de um novo tipo de sensor ou atuador passou a exigir apenas a criação de um novo microsserviço ou a extensão de um já existente, sem impacto sobre os demais contextos, evidenciando a escalabilidade e a organização alcançadas com a solução, construída exclusivamente sobre tecnologias de código aberto e infraestrutura containerizada com Docker.

Apesar dos resultados positivos, algumas limitações foram identificadas ao longo do desenvolvimento. A mais relevante diz respeito à complexidade operacional inerente à arquitetura escolhida: orquestrar a inicialização ordenada de sete serviços, do banco de dados PostgreSQL e do message broker NATS exigiu a elaboração de health checks e de um Makefile robusto, uma vez que inconsistências na ordem de subida resultavam em falhas de conexão que precisavam ser investigadas individualmente nos logs de cada container, dificuldade que não se apresentaria em uma abordagem monolítica. A aplicação do DDD em Go também demandou disciplina constante, pois a linguagem não impõe por sintaxe a separação entre camadas, exigindo o estabelecimento e a observância de convenções rígidas. Além disso, a integração do firmware ESP32 requereu um equilíbrio cuidadoso entre a frequência de leitura dos sensores e a capacidade da pilha de rede do dispositivo, sendo necessário distribuir as tarefas entre os dois núcleos do processador por meio do FreeRTOS.

Como trabalhos futuros, pretende-se ampliar o ecossistema com novos contextos de domínio, como climatização e gestão de energia, além de incorporar mecanismos de controle de acesso baseados em atributos para reforçar a segurança da comunicação entre os serviços distribuídos. Vislumbra-se também a realização de testes de carga mais abrangentes, capazes de mensurar com precisão a latência de propagação de eventos e a tolerância a falhas sob alta demanda; a adoção de uma camada de observabilidade com Prometheus e Grafana para monitoramento dos microsserviços em produção; e a evolução da interface para uma versão mobile. Pretende-se, ainda, conduzir a avaliação comparativa entre Go e Rust prevista na concepção inicial do projeto, reimplementando os serviços de maior criticidade, em especial o de segurança e o motor de regras, na linguagem Rust, a fim de mensurar, sob as mesmas condições de carga, eventuais ganhos de desempenho, consumo de memória e segurança de memória em relação à implementação atual em Go. Por fim, prevê-se a incorporação de assistentes de voz ao ecossistema, por meio da integração com plataformas como Alexa e Google Assistant, permitindo que o usuário acione a iluminação, consulte o estado dos sensores e arme ou desarme o alarme por comandos de voz, ampliando a acessibilidade e a naturalidade da interação com o sistema. De modo geral, conclui-se que o protótipo Aurora sustenta a hipótese central deste trabalho, demonstrando, em ambiente controlado, que a combinação de Domain-Driven Design com arquitetura de microsserviços constitui uma abordagem viável e eficaz para tornar sistemas de automação residencial mais modulares, escaláveis e sustentáveis a longo prazo.

Referências

EVANS, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley, 2004.

HIPPCHEN, Benjamin et al. Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. International Journal on Advances in Software, v. 10, n. 3-4, p. 432-445, 2017.

LE, Duc Minh; DANG, Duc-Hanh; VO, Hieu Dinh. Layered microservices architecture: A multitree-based domain-driven approach. Information and Software Technology, v. 183, art. 107720, jul. 2025.

NEWMAN, Sam. Building Microservices: Designing Fine-Grained Systems. 2. ed. Sebastopol: O'Reilly Media, 2021.

ÖZKAN, Ozan; BABUR, Önder; VAN DEN BRAND, Mark. Domain-Driven Design in software development: A systematic literature review on implementation, challenges, and effectiveness. Journal of Systems and Software, v. 230, art. 112537, dez. 2025.

RIZZARDI, Alessandra; SICARI, Sabrina; COEN-PORISINI, Alberto. Attribute-based policies through microservices in a smart home scenario. Computer Communications, v. 231, art. 108039, fev. 2025.

SIDDIQUI, Hassaan; KHENDEK, Ferhat; TOEROE, Maria. Microservices based architectures for IoT systems – State-of-the-art review. Internet of Things, v. 23, art. 100854, out. 2023.

VERNON, Vaughn. Implementing Domain-Driven Design. Upper Saddle River: Addison-Wesley, 2013.