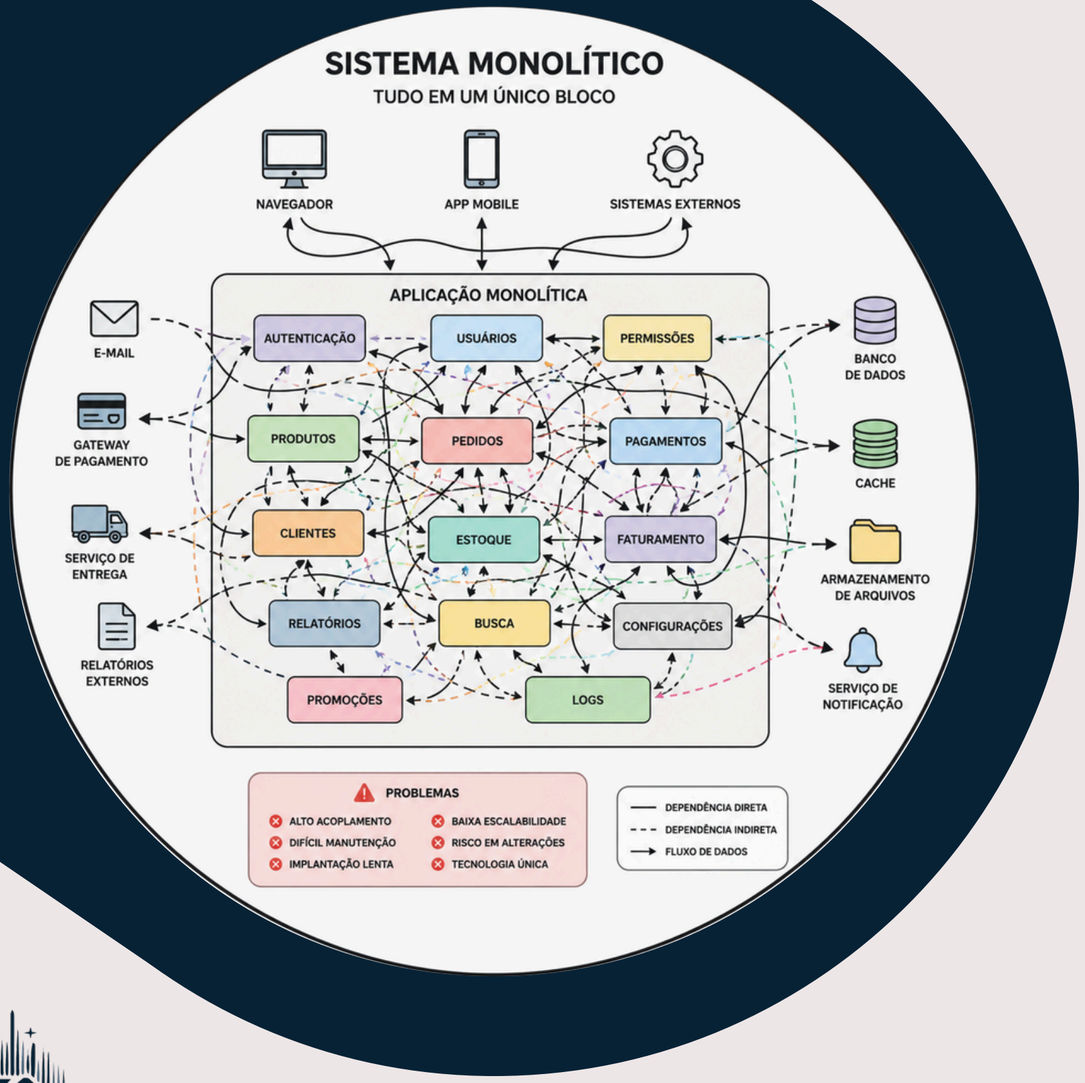
Aplicando Domain-Driven Design em Microsserviços

(Automação Residencial)

Vinicius Mendes Castro, Nicolas Nojiri,
Orientador: Humberto Patrick Lacerda Ribeiro, Universidade de Uberaba

O Problema

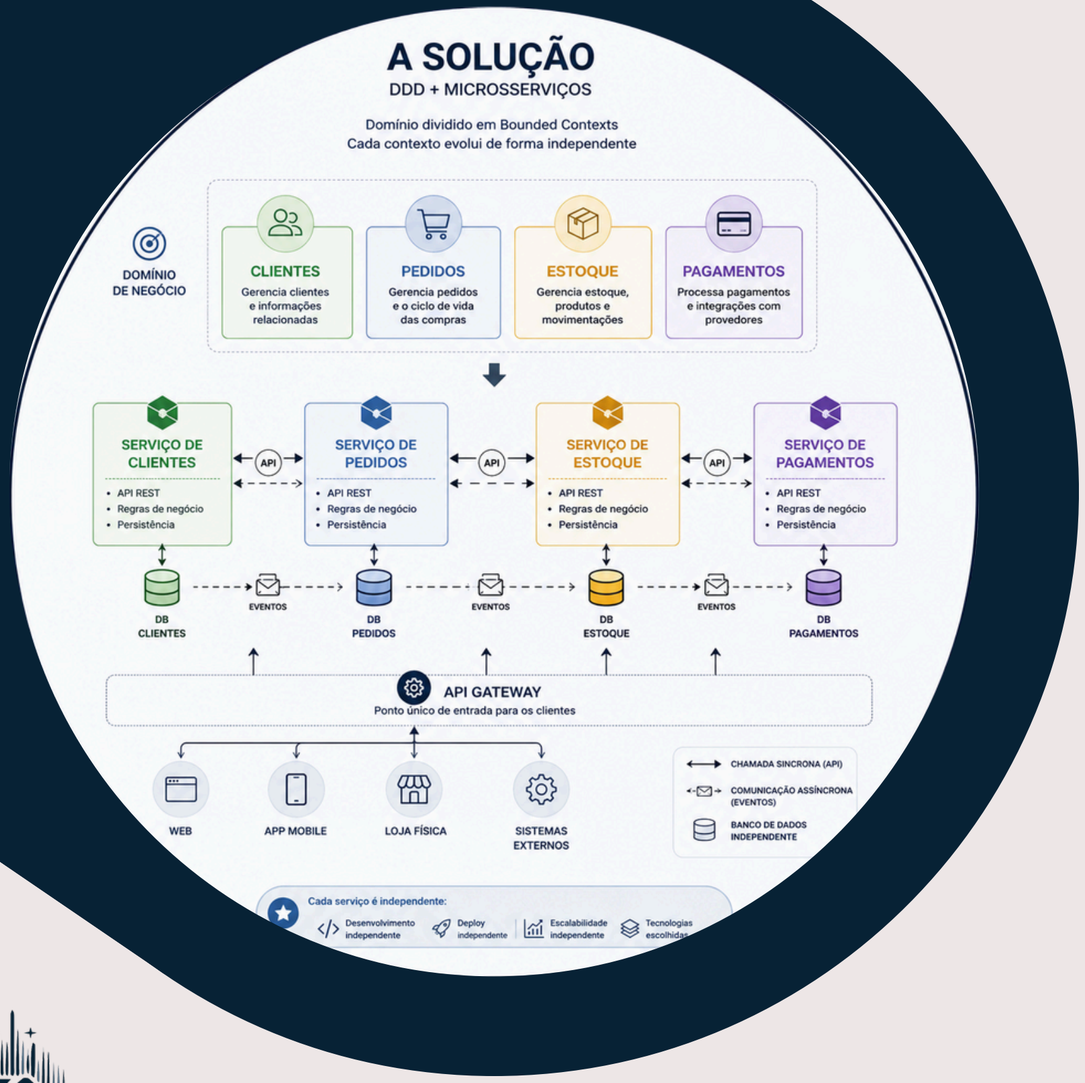


- Smart homes integram dispositivos heterogêneos (sensores, lâmpadas, alarmes, câmeras, fechaduras…)
- Muitos sistemas ainda são monolíticos e centralizados
- Consequência: alto acoplamento, baixa escalabilidade, manutenção difícil, integração de novos dispositivos limitada

Pergunta, Objetivo e Abordagem

- Como DDD + microsserviços melhoram modularidade, manutenção e escalabilidade em automação residencial?
- Protótipo funcional Aurora, evidenciando ganhos frente ao modelo monolítico nesse contexto
- Pesquisa aplicada e experimental, validada com testes e cenários reais

A Solução

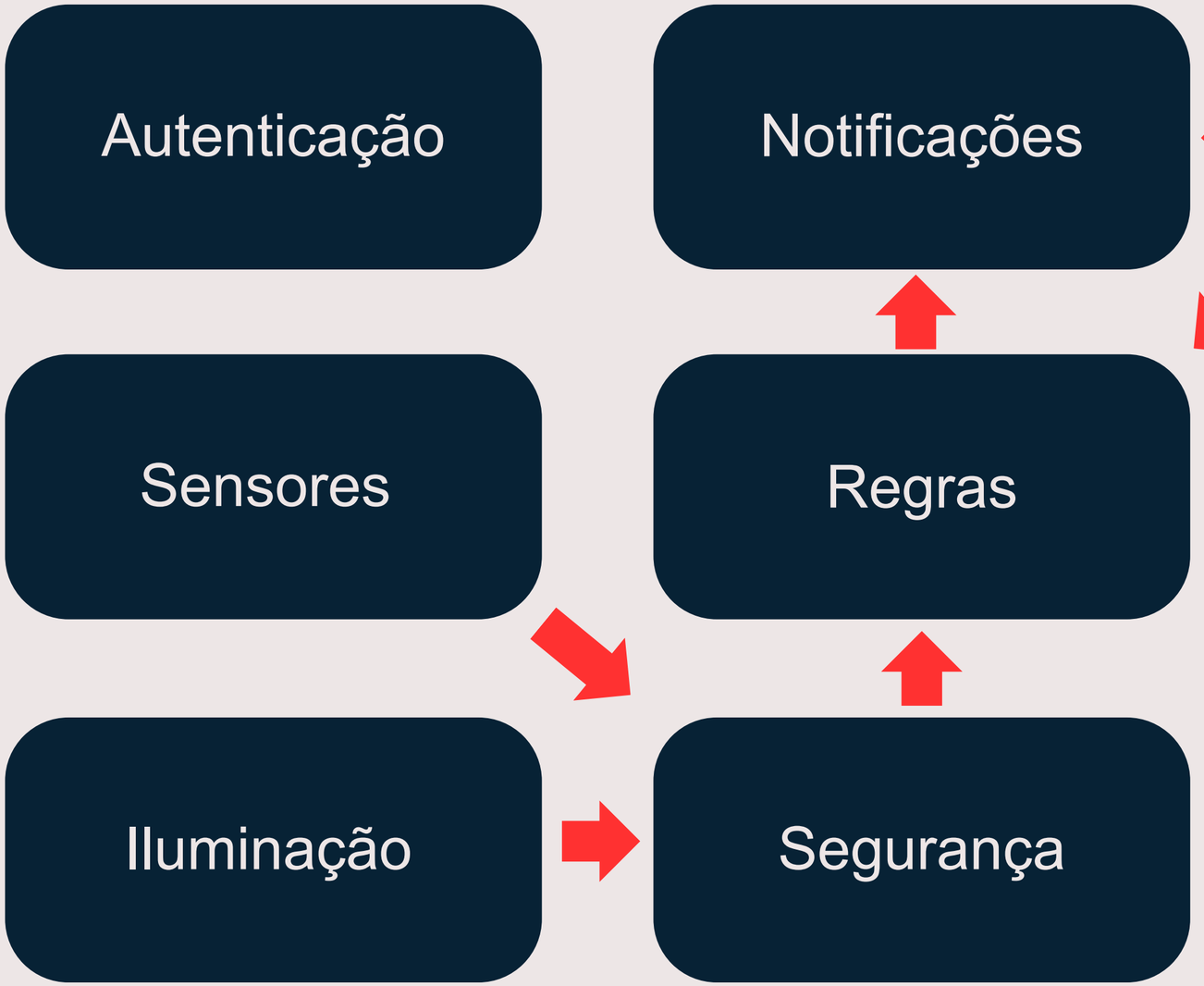


- DDD → organiza o sistema em torno do domínio, separando responsabilidades em bounded contexts
- Microsserviços → cada contexto vira um serviço independente, que evolui sozinho
- Correspondência direta: limite do domínio = limite do serviço
- Resultado esperado: menos acoplamento, mais resiliência e escalabilidade

Domain-Driven Design

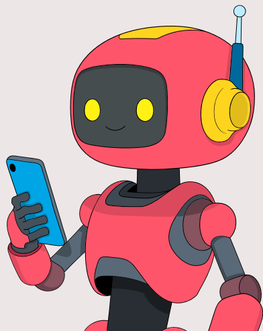
- DDD = modelagem centrada na compreensão do domínio (Evans, 2004)
- Blocos: Entidades · Agregados · Value Objects · Eventos de domínio
- Bounded Context: delimita uma área do conhecimento e evita ambiguidade
- Linguagem ubíqua: vocabulário próprio de cada contexto

Bounded Contexts



- Recorte do domínio = recorte dos microsserviços
- Cada um com:
  - Linguagem ubíqua própria
  - Regras de negócio próprias
  - Modelos de dados próprio

= NATS (Message Broker)



Arquitetura de Microsserviços

- 7 microsserviços independentes, banco compartilhado, reduzindo overhead de infraestrutura
- NATS como message broker (comunicação assíncrona)
- Nginx como API Gateway (roteamento)
- Frontend Angular 18 · ESP32 (hardware real)
- Tudo containerizado com Docker / Docker Compose

ARQUITETURA DE MICROSSERVIÇOS

7 Microsserviços Independentes • NATS (Message Broker) • Nginx (API Gateway) • Frontend Angular 18 • ESP32 (Hardware Real)

TUDO CONTAINERIZADO COM DOCKER / DOCKER COMPOSE

FRONTEND

Angular 18

HTTPS

NGINX API GATEWAY

- ✓ Roteamento de Requisições
- ✓ Load Balancing
- ✓ Autenticação / Segurança
- ✓ Rate Limiting
- ✓ CORS / Headers

ROTEAMENTO
/api/*

HARDWARE REAL

ESP32

SENSORES
- Temperatura
- Umidade
- Movimento
- Porta/Janelas
- Fumaça
- ...

ATUADORES
- Iluminação
- Sirene
- Fechaduras
- Relés
- ...

1 AUTENTICAÇÃO

Gerencia usuários, login e permissões

API REST

PostgreSQL autenticacao_db

2 SENSORES

Coleta e gerencia dados dos sensores

API REST

PostgreSQL sensores_db

3 ILUMINAÇÃO

Controle e gestão de iluminação

API REST

PostgreSQL iluminacao_db

4 SEGURANÇA

Monitoramento e gestão de segurança

API REST

PostgreSQL seguranca_db

5 REGRAS

Motor de regras e automações

API REST

PostgreSQL regras_db

6 NOTIFICAÇÕES

Envio de alertas e notificações

API REST

PostgreSQL notificacoes_db

7 AGENDAMENTOS

Agendamento de ações e tarefas

API REST

PostgreSQL agendamentos_db

NATS MESSAGE BROKER

Comunicação Assíncrona (Pub/Sub)

Publicados:
- sensors.data
- security.alert
- rules.executed
- schedules.triggered

Consumidos:
- lighting.command
- notifications.send
- users.events
- system.events

ESP32 + DISPOSITIVOS

Sensores

Atuadores

CONTAINERIZAÇÃO

Docker

Docker Compose

- ✓ Isolamento
- ✓ Portabilidade
- ✓ Escalablidade
- ✓ Fácil Implantação

TECNOLOGIAS

Angular 18 • NGINX • NATS • PostgreSQL • ESP32

Docker • Docker Compose

BENEFÍCIOS

- ✓ Serviços Independentes
- ✓ Bancos de Dados Isolados
- ✓ Comunicação Assíncrona
- ✓ Escalabilidade Horizontal
- ✓ Alta Resiliência
- ✓ Facilidade de Manutenção
- ✓ Evolução Contínua

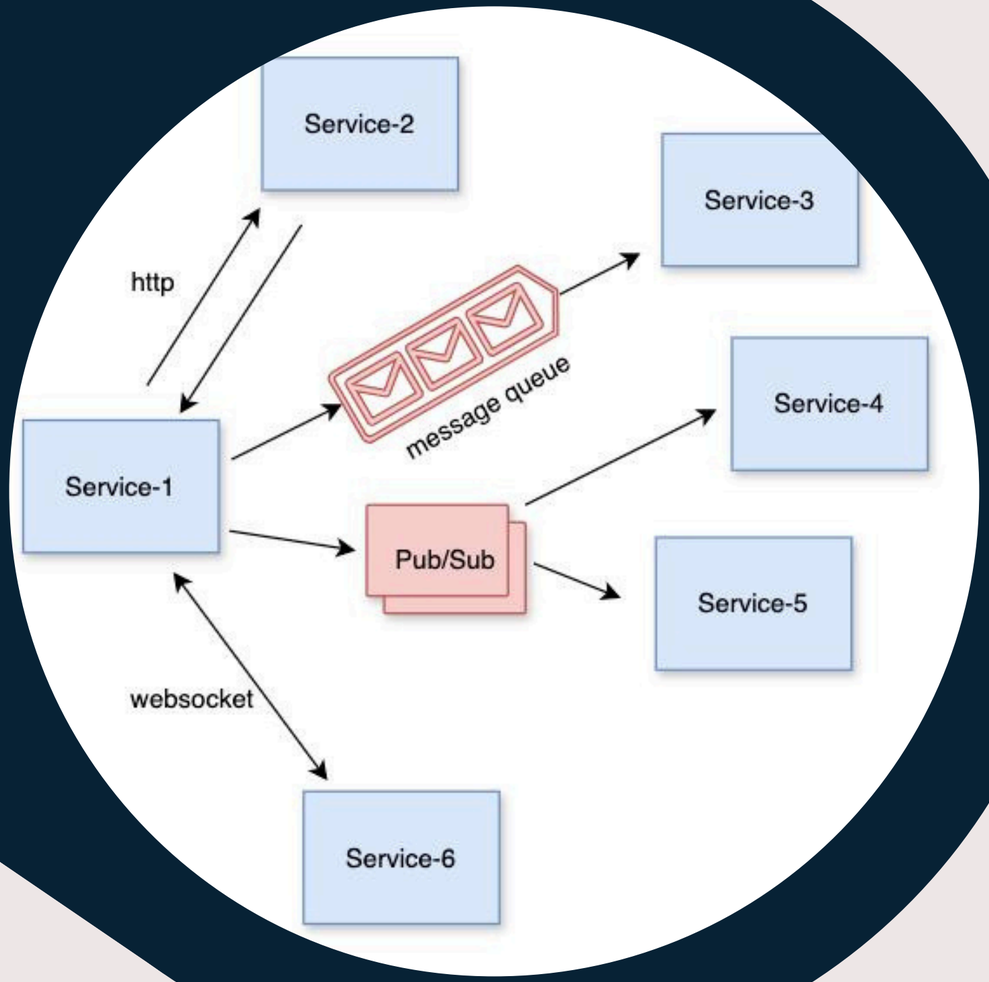
LEGENDA

Requisição HTTP (Síncrona) — Roteamento / API — Comunicação Assíncrona (NATS) — Dados Persistidos — Conexão Hardware (ESP32)

Arquitetura em Camadas

- Domínio → entidades, agregados, value objects, eventos (sem dependências externas)
- Aplicação → casos de uso, orquestrados por interfaces
- Infraestrutura → handlers HTTP, repositórios, publishers/consumers
- Inicialização → conecta os componentes e sobe o serviço
- Objetivo: isolar a lógica de negócio dos detalhes técnicos

Comunicação entre Contextos



- Síncrona — HTTP/REST: ações imediatas entre serviços
- Assíncrona — NATS PubSub: publicação/consumo de eventos de domínio
- Tempo real — WebSocket: atualizações entregues ao frontend

Fluxo em Tempo Real

- ESP32 detecta movimento (sensor PIR) → envia ao serviço de sensores
- Serviço de Sensores persiste e publica o evento de detecção no NATS
- Serviço de Regra avalia regras → aciona ação (ex.: liga lâmpada via serviço de Luzes)
- Serviço de Notificação registra a ocorrência
- Em paralelo: atualização via WebSocket chega ao painel

Resultados

- Modularidade comprovada: contextos isolados, sem dependências acidentais
- Tempo real: NATS + WebSocket → mudanças refletem na hora, sem polling
- Escalabilidade: novo dispositivo = novo serviço (ou extensão de um), sem impactar o resto
- Limitação principal: complexidade operacional (orquestrar 7 serviços + DB + NATS)

Conclusão e Trabalhos Futuros

- A combinação DDD + microsserviços se mostrou viável e eficaz para automação residencial mais modular, escalável e sustentável

- Objetivo alcançado: protótipo Aurora funcional (7 serviços + web + ESP32), 100% open source

- Futuro:
  - climatização e gestão de energia
  - observabilidade (Prometheus/Grafana)
  - versão mobile
  - comparação Go × Rust
  - assistentes de voz (Alexa/Google)