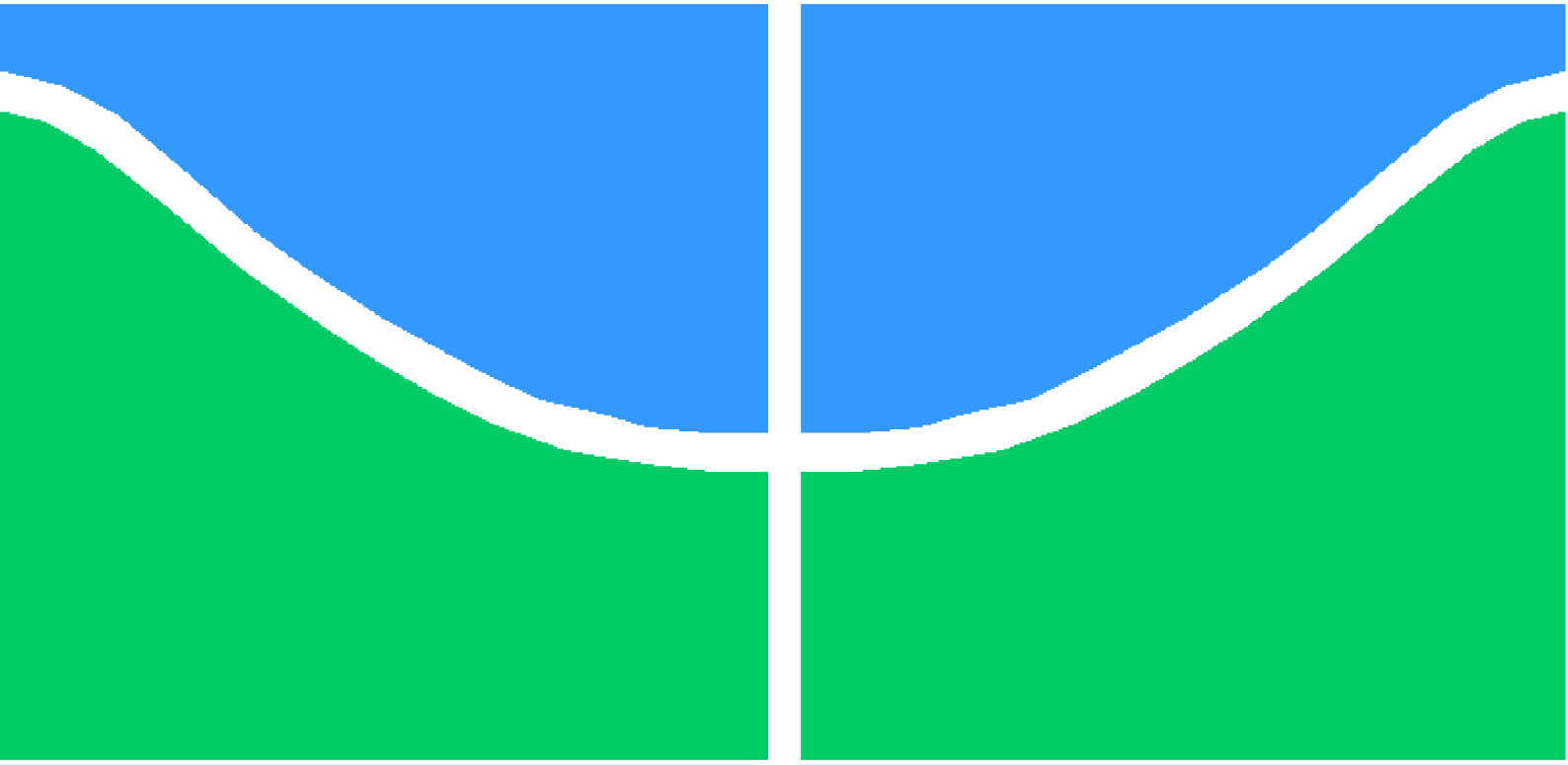


Universidade de Brasília – UnB

Faculdade de Ciências e Tecnologias em Engenharia – FCTE

Engenharia de Software

Da Análise à Implementação: Migração da Infraestrutura de IaaS para FaaS com Foco na Redução de Custos no AgroMart

Autores: Kalebe Lopes da Cunha e Murilo Schiler Lopes Santana

Orientador: Prof. Dr. André Luiz Peron Martins Lanna

Brasília, DF

2025

Kalebe Lopes da Cunha e Murilo Schiler Lopes Santana

Da Análise à Implementação: Migração da Infraestrutura de IaaS para FaaS com Foco na Redução de Custos no AgroMart

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Universidade de Brasília – UnB

Faculdade de Ciências e Tecnologias em Engenharia – FCTE

Orientador: Prof. Dr. André Luiz Peron Martins Lanna

Brasília, DF

2025

Kalebe Lopes da Cunha e Murilo Schiler Lopes Santana

Da Análise à Implementação: Migração da Infraestrutura de IaaS para FaaS com Foco na Redução de Custos no AgroMart

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Brasília, DF
2025

Resumo

O AgroMart é um aplicativo que conecta produtores rurais e consumidores por meio do modelo de Comunidades que Sustentam a Agricultura (CSAs). Essas comunidades permitem a comercialização direta de alimentos, promovendo um consumo mais sustentável e aproximando os consumidores dos produtores locais.

Na arquitetura de implantação atual, o agricultor arca com custos em dólares para manter sua CSA em funcionamento, pois o Agromart está implantado em uma arquitetura tradicional de IaaS. Como buscamos tornar essa solução acessível ao maior número possível de agricultores, não podemos impor custos a eles pela escolha do Agromart. Portanto, é essencial analisar a viabilidade de uma alternativa de implantação que seja sempre gratuita e acessível ao agricultor e, caso exista, implementá-la.

Este Trabalho de Conclusão de Curso (TCC) propõe a migração da API de dicionário e do backend STRAPI da aplicação, que atualmente operam sob o modelo de Infraestrutura como Serviço (IaaS) utilizando instâncias Amazon Elastic Compute Cloud (EC2), para uma arquitetura baseada no paradigma de computação serverless, especificamente Função como Serviço (FaaS), através da AWS Lambda.

A pesquisa abordará a viabilidade técnica e econômica dessa migração, investigando se a infraestrutura pode operar dentro dos limites do plano gratuito da AWS, reduzindo custos operacionais. Além disso, será analisada a capacidade das APIs de ultrapassar os limites da camada gratuita e as implicações de custos adicionais, bem como a necessidade de modificações na aplicação para garantir uma migração eficiente, sem impactos negativos na experiência do usuário.

A adoção de computação serverless oferece benefícios como redução de custos com infraestrutura, escalabilidade automática e menor complexidade operacional. Este trabalho documenta todas as etapas envolvidas na transição de arquiteturas tradicionais para serverless.

Palavras-chave: Serverless Computing, Função como Serviço (FaaS), AWS Lambda, Escalabilidade, Redução de Custos, Infraestrutura como Serviço (IaaS), Cold-start, Agromart, Computação em Nuvem.

Abstract

AgroMart is an application that connects rural producers and consumers through the Community-Supported Agriculture (CSA) model. These communities enable the direct commercialization of food, promoting more sustainable consumption and bringing consumers closer to local producers.

In the current deployment architecture, farmers bear costs in dollars to keep their CSA operational, as AgroMart is implemented using a traditional Infrastructure as a Service (IaaS) model. Since our goal is to make this solution accessible to as many farmers as possible, we cannot impose costs on them for choosing AgroMart. Therefore, it is essential to analyze the feasibility of a deployment alternative that is always free and accessible to farmers and, if viable, implement it.

This Final Year Project (TCC) proposes migrating the application's dictionary API and STRAPI backend, which currently operate under the IaaS model using Amazon Elastic Compute Cloud (EC2) instances, to an architecture based on the serverless computing paradigm, specifically Function as a Service (FaaS), through AWS Lambda.

The research will address the technical and economic feasibility of this migration, investigating whether the infrastructure can operate within the AWS free tier limits to reduce operational costs. Additionally, it will analyze whether the APIs may exceed the free tier limits and the implications of additional costs, as well as the necessary modifications to the application to ensure an efficient migration without negatively impacting the user experience.

Adopting serverless computing offers benefits such as cost reduction, automatic scalability, and lower operational complexity. This project documents all the steps involved in transitioning from traditional architectures to serverless.

Keywords: Serverless Computing, Function as a Service (FaaS), AWS Lambda, Scalability, Cost Reduction, Infrastructure as a Service (IaaS), Cold-start, AgroMart, Cloud Computing.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
AWS	Amazon Web Services
CMM	Capability Maturity Model
CMS	Content Management System
CSAs	Comunidades que Sustentam a Agricultura
DynamoDB	Dynamic Database
EC2	Elastic Compute Cloud
FaaS	Função como Serviço
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IaaS	Infraestrutura como Serviço
iOS	iPhone Operating System
JS	JavaScript
NoSQL	Not Only Structured Query Language
TCC	Trabalho de Conclusão de Curso
TI	Tecnologia da Informação
UnB	Universidade de Brasília
URL	Uniform Resource Locator
XP	Extreme Programming
EBS	Elastic Block Store

Sumário

1 Introdução

O conceito de Comunidade que Sustenta a Agricultura (CSA) surgiu como uma alternativa colaborativa onde consumidores e pequenos produtores de alimentos formam parcerias diretas que facilitam a comercialização de produtos frescos e sustentáveis. Neste contexto, o aplicativo Agromart funciona como uma ponte digital entre produtores e consumidores, facilitando essas conexões.

A infraestrutura em nuvem desempenha um papel essencial para aplicações como o Agromart, que depende de um sistema robusto para lidar com variações de tráfego de usuários, além de garantir a confiabilidade no atendimento às demandas de produtores e consumidores. A capacidade de uma aplicação escalar eficientemente e manter os custos sob controle são pontos cruciais para seu sucesso no mercado.

Atualmente, o backend da Agromart está hospedado na Amazon Web Services (AWS) via Elastic Compute Cloud (EC2), um modelo de infraestrutura como serviço (IaaS) no qual recursos computacionais como máquinas virtuais são gerenciados pelos usuários. Embora esta abordagem proporcione flexibilidade e controle, pode se ter muita ociosidade em seus serviços, o que pode levar a custos elevados sem uso. A migração do Agromart para um modelo de função como serviço (FaaS) via AWS Lambda oferece uma alternativa mais eficiente e escalável. Diferentemente da IaaS, o FaaS permite a execução sob demanda de funções específicas, eliminando a necessidade de provisionar servidores ociosos, escalando automaticamente conforme o volume de requisições e resultando em uma economia significativa de custos operacionais. (SILVA;CARVALHO, 2021)

A transição para a computação serverless, particularmente através do FaaS, possibilita a elasticidade necessária para que o Agromart atenda a picos de demanda sem incorrer em altos custos de infraestrutura (FERREIRA IGOR FARIAS, 2023). Esse modelo garante que o sistema seja escalável conforme a necessidade, sem que os desenvolvedores precisem gerenciar manualmente os recursos subjacentes. Isso é particularmente importante para a Agromart, que pode enfrentar variações sazonais na demanda.

Este trabalho irá analisar a migração do backend do Agromart de um ambiente IaaS para FaaS, com foco na redução de custos operacionais. Inicialmente, será feita uma análise teórica sobre ambos os modelos de computação em nuvem (IaaS e FaaS), destacando suas vantagens e desvantagens. Em seguida, será realizado um estudo experimental para explorar os limites, benefícios e precauções envolvidas na migração.

1.1 História do AgroMart

O AgroMart foi criado com o objetivo de conectar pequenos agricultores e consumidores, promovendo a produção sustentável por meio de Comunidades que Sustentam a Agricultura (CSAs). A ideia nasceu durante um hackathon na Universidade de Brasília, campus Gama, inspirado pela história de uma produtora que implementou uma barraca da honestidade para melhorar suas vendas. Inicialmente, o projeto era simples, permitindo aos agricultores divulgar pontos de venda e os consumidores acessarem informações sobre a disponibilidade de produtos.

Com o tempo, o projeto evoluiu, recebendo apoio de professores e especialistas para otimizar suas funcionalidades. A partir de pesquisas e entrevistas, foi desenvolvido um aplicativo mobile que atenderia de maneira mais adequada às necessidades dos agricultores e co-agricultores, chamado de AgroMart. A principal característica do AgroMart é sua capacidade de individualizar as CSAs, facilitando a organização da produção e o escoamento dos alimentos de forma ágil e eficiente.

A evolução do projeto não parou por aí. Foram realizadas várias iterações para melhorar sua eficiência ao longo dos anos, como a automação de processos e a migração da infraestrutura para soluções mais econômicas. Hoje, o AgroMart continua sendo aprimorado com foco em economia, escalabilidade, segurança e integração de novas funcionalidades, visando sempre facilitar o acesso a alimentos frescos e sustentáveis.

1.2 Problema

O principal problema enfrentado pelo Agromart reside no elevado custo de manutenção de sua infraestrutura atual e também no possível problema de escalabilidade visto que a máquina utilizada atualmente para servir a aplicação não possui bons poderes computacionais, ou seja, pode trazer diversos problemas para os usuários como travamentos, lentidões e até mesmo reinicializações do servidor caso haja um pico de requisições simultâneas.

A API Dicionário e o Backend do Agromart estão hospedados na Amazon Web Services (AWS) utilizando o Elastic Compute Cloud (EC2), que, apesar de oferecer flexibilidade e controle ao usuário, demanda o provisionamento contínuo de servidores, independentemente da carga de trabalho. Essa abordagem gera custos elevados desnecessários, principalmente em períodos de baixa demanda, devido à necessidade de manter a infraestrutura operacional mesmo quando o tráfego no aplicativo é reduzido ou nulo (FERREIRA IGOR FARIAS, 2023).

A instância do servidor EC2 atualmente utilizado para disponibilizar a API Dicionário e o Backend do AgroMart possui as seguintes características:

- Tipo de instância: t3.medium
- vCPUs: 2
- Memória RAM: 4 Gb
- Armazenamento: 30 Gb
- Sistema Operacional: Linux

Nome	vCPUs	Memória (GiB)	Performance de linha de base por vCPU	Créditos de CPU obtidos por hora	Largura de banda para expansão de rede (Gbps)	Largura de banda para expansão do EBS (Mbps)	Preço sob demanda por hora*	Instância reservada por 1 ano – por hora*	Instância reservada por 3 anos efetiva por hora*
t3.nano	2	0,5	5%	6	5	Até 2.085	USD 0,0052	USD 0,003	USD 0,002
t3.micro	2	1,0	10%	12	5	Até 2.085	USD 0,0104	USD 0,006	USD 0,005
t3.small	2	2,0	20%	24	5	Até 2.085	USD 0,0209	USD 0,012	USD 0,008
t3.medium	2	4,0	20%	24	5	Até 2.085	USD 0,0418	USD 0,025	USD 0,017
t3.large	2	8,0	30%	36	5	Até 2.780	USD 0,0835	USD 0,05	USD 0,036
t3.xlarge	4	16,0	40%	96	5	Até 2.780	USD 0,1670	USD 0,099	USD 0,067
t3.2xlarge	8	32,0	40%	192	5	Até 2.780	USD 0,3341	USD 0,199	USD 0,133

Figura 1 – Tabela de preços EC2

1.2.1 Cálculo do Custo Mensal de um Servidor EC2 t3.medium na AWS

Para estimar o custo mensal de um servidor **Amazon EC2 t3.medium**, considera-se a tarifa sob demanda de **0,0418 dólares por hora**, conforme divulgado pela *Amazon Web Services (AWS)*. O cálculo é realizado com base nos seguintes parâmetros:

- **Custo por hora**: 0,0418 USD/hora
- **Número total de horas no mês**:

$$24 \text{ horas/dia} \times 30 \text{ dias/mês} = 720 \text{ horas/mês} \tag{1.1}$$

- **Cotação do dólar**: 5,70 BRL/USD

Dessa forma, o custo mensal em dólares é obtido por:

$$C_{\text{mensal,USD}} = 0,0418 \times 720 = 30,096 \text{ USD} \tag{1.2}$$

Convertendo esse valor para reais, considerando a cotação de 5,70 BRL/USD:

$$C_{\text{mensal,BRL}} = 30,096 \times 5,70 = 171,55 \text{ BRL} \tag{1.3}$$

1.2.2 Resultados

O custo estimado para manter um servidor **EC2 t3.medium** em funcionamento contínuo durante um mês é de aproximadamente **R$ 171,55**. Vale ressaltar que este cálculo não inclui custos adicionais, como armazenamento *EBS*, transferência de dados ou licenciamento de software, os quais podem impactar significativamente o valor final da fatura na AWS.

Além disso, vale frisar que o gerenciamento manual de recursos limita a capacidade do sistema de escalar automaticamente de forma eficiente, o que impacta negativamente a experiência do usuário em momentos de pico de utilização.

A migração do backend para um modelo de Função como Serviço (FaaS) no AWS Lambda surge como uma possível solução viável, pois permite a execução de funções sob demanda, eliminando a necessidade de servidores ociosos e promovendo uma escalabilidade automática conforme a demanda varia (SILVA;CARVALHO, 2021). A transição para o FaaS tem o potencial de reduzir significativamente os custos operacionais do Agromart e aumentar sua capacidade de atender a cargas variáveis sem comprometer o desempenho da aplicação.

1.3 Objetivo Geral

O objetivo geral deste trabalho é avaliar a viabilidade da adoção do FaaS pelos sistemas do Agromart. Essa mudança visa migrar o backend e a api dicionário do Agromart, atualmente hospedada na Amazon Web Services (AWS) Elastic Compute Cloud (EC2), para uma arquitetura baseada em Função como Serviço (FaaS), utilizando o AWS Lambda e o framework Serverless. Essa migração visa melhorar a eficiência do uso de recursos computacionais, reduzindo os custos operacionais associados à infraestrutura dos servidores.

Sabendo que o Agromart estabelece a conexão entre agricultores e consumidores por meio das Comunidades que Sustentam a Agricultura (CSAs), os custos operacionais são financiados pelos membros associados a cada CSA. Nesse contexto, a adoção do Function as a Service (FaaS) surge como uma alternativa estratégica para otimizar a eficiência financeira do sistema. (RIBEIRO; MAGALHãES, 2023).

Com essa abordagem, espera-se que a migração para o FaaS contribua para a redução dos custos operacionais, garantindo maior viabilidade econômica para as CSAs envolvidas e promovendo um modelo mais sustentável de gestão e escalabilidade do serviço. Além disso, a arquitetura baseada em FaaS permite uma alocação mais eficiente de recursos computacionais, reduzindo o desperdício de capacidade ociosa e ajustando automaticamente a infraestrutura conforme a demanda do sistema.

Outro benefício relevante está na redução da complexidade de manutenção, uma vez que a responsabilidade pela administração dos servidores e da infraestrutura subjacente é delegada ao provedor de serviços em nuvem. Dessa forma, a equipe de desenvolvimento pode concentrar esforços na melhoria das funcionalidades do Agromart, aprimorando a experiência dos usuários e fortalecendo o ecossistema das CSAs.

1.4 Objetivos Específicos

1.4.1 Analisar a Atual Arquitetura do AgroMart

A arquitetura atual do Agromart é estruturada em três componentes principais, cada um com responsabilidades bem definidas que, quando integrados, visam fornecer um bom funcionamento do sistema. A seguir, detalharemos cada um desses componentes, analisando suas funções e interações no contexto geral do projeto.

1.4.1.1 Arquitetura Geral

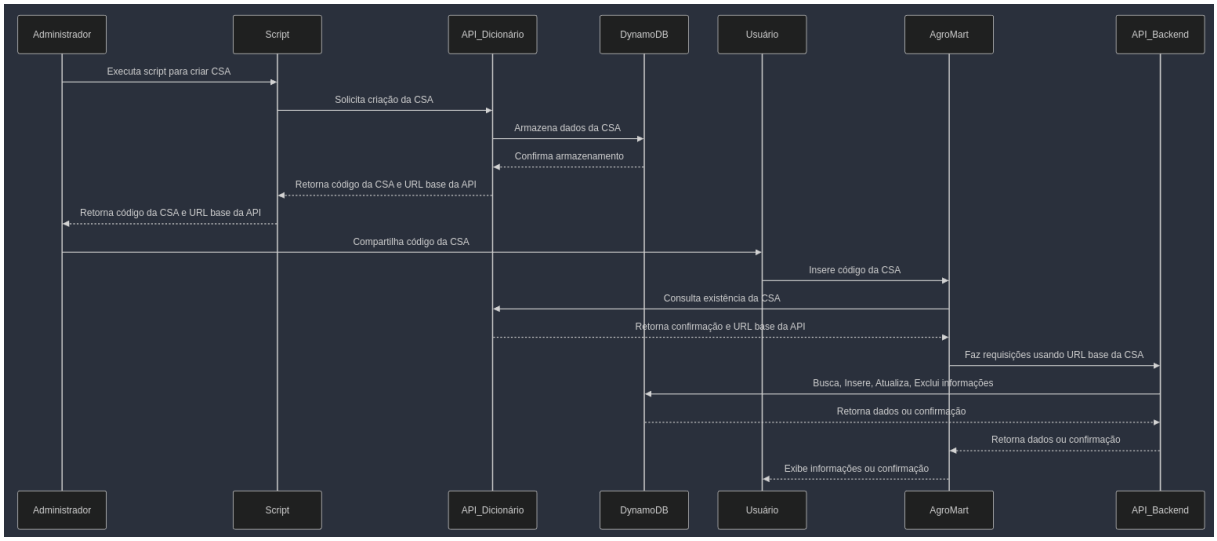


Figura 2 – Arquitetura AgroMart

1.4.1.2 API Dicionário

A primeira camada do sistema é composta pela API do Dicionário que foi desenvolvida em JavaScript. Essa API desempenha um papel fundamental no processo de cadastro e gerenciamento das URLs específicas de cada CSA participante do Agromart. Através dessa API, o sistema tem acesso às URLs únicas de cada CSA, as quais serão consultadas posteriormente pela API principal. Esse componente atua como o ponto inicial de conexão entre o Agromart e as diferentes CSAs, fornecendo um ponto central para registrar e gerenciar essas URLs.

1.4.1.3 API Principal Backend

A API principal do Backend do Agromart, também desenvolvida em JavaScript, é o núcleo do sistema. Ela é responsável por gerenciar as informações essenciais do Agromart, como dados de produtos, planos, clientes e lojas. A API principal realiza consultas dinâmicas à URL cadastrada pela API do Dicionário, permitindo que o Agromart acesse e gerencie as informações de cada CSA individualmente. Cada CSA mantém seu próprio servidor EC2, o que, embora proporcione auto- nomia, implica em custos elevados devido à necessidade de infraestrutura separada para cada unidade. Essa arquitetura, ao invés de otimizar os recursos, acaba impactando dire- tamente o orçamento das CSAs, tornando-o menos eficiente em termos de custo-benefício.

1.4.1.4 Aplicativo Mobile

O componente final da arquitetura é o aplicativo móvel, que fornece a interface com o usuário para o Agromart. Através do app, os usuários podem interagir diretamente com o sistema, realizando as operações baseadas nas informações gerenciadas pela API principal. O aplicativo mobile faz requisições para a API, obtendo os dados necessários para exibição ao usuário, utilizando a URL gerada para cada CSA específica. Esse componente é crucial para garantir a acessibilidade e a experiência do usuário, permitindo que as operações do Agromart sejam realizadas de maneira eficiente em dispositivos móveis.

1.4.2 Dockerização da API Strapi e da API Dicionário

Antes da migração para a arquitetura *Function as a Service* (FaaS), é essencial a containerização das APIs para uma compreensão detalhada do funcionamento do sistema legado *Agromart* e de suas regras de negócio. Para isso, será desenvolvido um ambiente de execução baseado em *Docker Compose*, permitindo a inicialização da aplicação em diferentes ambientes, juntamente com o banco de dados PostgreSQL, de maneira padronizada e eficiente.

1.4.3 Migração das APIs para um Ambiente Serverless

Após a análise da implementação do sistema *Agromart*, será realizada a refatoração do código-fonte para sua adaptação ao ambiente *Serverless*, utilizando o *Serverless Framework*. Um aspecto crítico dessa transição é a preservação das regras de negócio já implementadas na API Strapi, uma vez que, por se tratar de um *Content Management System* (CMS), grande parte do código encontra-se abstraída, dificultando sua compreensão e modificação direta.

1.4.4 Automação do Deploy da API Principal em FaaS

Será desenvolvido um script para automatizar o processo de *deploy* do *back-end* de uma *CSA* na AWS Lambda. Além disso, esse script será responsável por realizar uma requisição à API Dicionário, informando a URL da *CSA* criada. A configuração será gerenciada por meio de um arquivo `.env`, contendo as credenciais do proprietário da *CSA*, garantindo a segurança e a flexibilidade do processo de implantação.

1.4.5 Refatoração da Aplicação Mobile para a API FaaS

A aplicação móvel do *Agromart* possui uma forte dependência da API baseada no CMS Strapi. Com a migração para a arquitetura *Serverless*, será necessário refatorar a lógica de requisições e o tratamento de dados no aplicativo móvel, garantindo a compatibilidade com a nova API FaaS e preservando a integridade das funcionalidades existentes.

1.4.6 Análise de Serviços "Sempre Gratuito"da AWS

Será realizada uma análise comparativa das plataformas disponíveis para identificar a solução mais adequada para oferecer um serviço de custo zero aos agricultores, dentro dos limites impostos pela provedora de nuvem. O estudo incluirá a quantificação do consumo de recursos e a justificativa técnica para a escolha da plataforma mais eficiente.

1.4.7 Estratégias para Minimização de Custos no AWS Lambda

Após a conclusão da migração para *Serverless*, serão definidas estratégias de otimização para minimizar os custos operacionais da aplicação no AWS Lambda. As configurações serão ajustadas para garantir um uso eficiente dos recursos computacionais, evitando escalabilidade desnecessária durante picos de requisições e assegurando que a aplicação permaneça dentro dos limites do *AWS Free Tier*.

1.4.8 Análise de Carga e Dimensionamento API FAAS

Será realizado um mapeamento detalhado dos casos de uso da API principal do *Agromart*, permitindo a definição precisa dos fluxos de interação entre os serviços. Esse levantamento será fundamental para a validação do novo modelo arquitetural e para a garantia da continuidade das funcionalidades essenciais da plataforma.

2 Referencial Teórico

2.1 Engenharia de Software

A Engenharia de Software é uma disciplina da computação que envolve a aplicação de princípios científicos, tecnológicos e gerenciais para o desenvolvimento e manutenção de sistemas de software de alta qualidade. Essa área surgiu em resposta à chamada "crise do software", onde projetos de software eram frequentemente entregues com atrasos, acima do orçamento, ou com baixa qualidade (WAZLAWICK, 2013).

Na prática, a Engenharia de Software visa organizar o desenvolvimento de software por meio de processos estruturados que garantam a eficiência, escalabilidade e qualidade do produto final (WAZLAWICK, 2013). Esses processos incluem etapas como levantamento de requisitos, modelagem, codificação, testes e manutenção, assegurando que o produto atenda às necessidades do cliente ao longo de seu ciclo de vida.

A área é dividida em várias subdisciplinas, incluindo gerenciamento de projetos de software, manutenção de software, qualidade de software, e processos de engenharia de software, cada uma focada em um aspecto específico do ciclo de vida do desenvolvimento. A aplicação de modelos de maturidade, como o CMM (Capability Maturity Model), é uma das estratégias amplamente utilizadas para melhorar a qualidade dos processos e dos produtos de software (WAZLAWICK, 2013). Além disso, técnicas como integração contínua e testes automatizados têm ganhado destaque como práticas essenciais para garantir a entrega ágil de software (WAZLAWICK, 2013).

2.2 Manutenção Adaptativa e Perfectiva em Engenharia de Software

(SWANSON, 1976) propôs uma classificação das atividades de manutenção em quatro tipos principais: corretiva, adaptativa, perfectiva e preventiva. A manutenção adaptativa refere-se às modificações realizadas no software para garantir sua compatibilidade com mudanças no ambiente operacional, mudanças no ambiente de implantação, arquitetura de implementação , bancos de dados, hardware ou novos padrões de comunicação (SOMMERVILLE, 2015).

A manutenção perfectiva tem como objetivo melhorar o desempenho, a usabilidade e a eficiência do software sem alterar sua funcionalidade básica (SWANSON, 1976). Esse tipo de manutenção pode incluir refatoramento de código, otimização de algoritmos, aumento da escalabilidade do sistema (SOMMERVILLE, 2015).

No AgroMart, as manutenções adaptativa e perfectiva foram as mais significativas. Com o objetivo de adaptar o sistema à nova plataforma AWS Lambda e à arquitetura serverless, foi necessária uma refatoração extensa para garantir o funcionamento adequado das funcionalidades previamente definidas e implementadas. Além disso, foi preciso assegurar que a implementação estivesse em conformidade com as regras da AWS, especialmente com relação ao Always Free Tier.

A otimização do código também se fez necessária. O uso das funções precisou ser rigorosamente controlado, como no caso do getBatch do DynamoDB, para evitar o excesso de chamadas e garantir o funcionamento dentro dos limites de taxa do Always Free Tier da AWS. Também foi essencial monitorar e controlar a taxa de escrita e leitura no DynamoDB para prevenir que os limites do plano gratuito fossem ultrapassados.

2.3 Computação em Nuvem

A computação em nuvem é um modelo de fornecimento de serviços de TI em que recursos computacionais, como armazenamento, processamento e redes, são disponibilizados como serviços através da internet. Este modelo permite que as empresas e usuários acessem e utilizem esses recursos sob demanda, sem a necessidade de adquirir ou manter infraestrutura física local. A computação em nuvem, segundo (CAPPELLOZZA, 2012), tem despertado grande interesse por seu potencial de alterar significativamente os investimentos em infraestrutura de TI e promover maior flexibilidade na gestão de recursos (CAPPELLOZZA, 2012).

Entre as características principais da computação em nuvem estão a escalabilidade e o pay-per-use. A escalabilidade permite que as empresas ampliem ou reduzam seus recursos conforme a demanda, enquanto o modelo de pagamento "pay-per-use"assegura que os custos estão atrelados ao uso real dos serviços, otimizando os investimentos (NOGUEIRA, 2013). Além disso, a infraestrutura em nuvem é gerenciada por provedores de serviços, que garantem a segurança, manutenção e disponibilidade dos recursos, aliviando as empresas da responsabilidade por essas operações.

A computação em nuvem pode ser categorizada em diferentes modelos de serviço, como Infraestrutura como Serviço (IaaS), Plataforma como Serviço (PaaS) e Software como Serviço (SaaS). Cada um desses modelos oferece diferentes níveis de controle e abstração para o usuário. A IaaS fornece servidores e armazenamento, a PaaS oferece um ambiente para desenvolvimento de software, e a SaaS disponibiliza aplicativos completos através da web (CAPPELLOZZA, 2012).

A adoção da computação em nuvem oferece benefícios como redução de custos operacionais, maior flexibilidade e mobilidade no acesso a dados e sistemas, além de melhorar a eficiência no uso de recursos tecnológicos. Contudo, ela também apresenta

desafios, como preocupações com a segurança dos dados e a dependência da conectividade com a internet (CAPPELLOZZA, 2012).

2.4 Escalabilidade de Aplicações em Nuvem

A escalabilidade de aplicações em nuvem refere-se à capacidade de um sistema aumentar ou diminuir seus recursos de forma eficiente, conforme a demanda dos usuários. Isso é fundamental para garantir que uma aplicação consiga atender a variações de tráfego sem comprometer o desempenho, mantendo os custos alinhados ao uso real dos recursos.

Existem dois tipos principais de escalabilidade: vertical e horizontal. A escalabilidade vertical envolve aumentar os recursos de uma máquina única, como adicionar mais memória ou processadores, enquanto a escalabilidade horizontal adiciona novas instâncias de servidores para lidar com a demanda crescente. A computação em nuvem facilita ambos os tipos de escalabilidade, permitindo que as empresas ajustem suas infraestruturas conforme necessário sem grandes investimentos iniciais em hardware (FRANçA AUDREY TELES DOS SANTOS, 2023).

Um exemplo comum de escalabilidade horizontal pode ser visto em serviços como o AWS EC2, que permite adicionar novas instâncias automaticamente conforme a carga aumenta, ou removê-las durante períodos de baixa demanda. Isso é particularmente útil em situações como eventos sazonais, onde picos de tráfego são previsíveis, como em promoções de Black Friday ou no Exame Nacional do Ensino Médio (ENEM), onde a escalabilidade se torna crucial para evitar quedas de serviço devido à sobrecarga (FRANçA AUDREY TELES DOS SANTOS, 2023).

3 Metodologia

3.1 Scrum

O Scrum, criado por Jeff Sutherland e Ken Schwaber, é uma metodologia ágil desenvolvida como uma alternativa flexível à abordagem tradicional em cascata, que se destacava por ser linear e pouco adaptável. Com a introdução das sprints, que são ciclos curtos de desenvolvimento, e a adaptação contínua por meio de reuniões regulares, o Scrum trouxe uma mudança radical para o desenvolvimento de software (SUTHERLAND, 2014). As sprints, geralmente com duração de duas semanas, começam com uma reunião de planejamento onde a equipe decide a quantidade de trabalho que poderá ser entregue no ciclo (FERREIRA, 2017; RIBEIRO; MAGALHãES, 2023). Em alguns casos, como o projeto Agromart, a implementação completa do Scrum pode ser desnecessária, especialmente em equipes pequenas ou focadas em tarefas imprevisíveis como correção de erros e defeitos, onde a estimativa precisa é mais difícil de alcançar. No entanto, mesmo sem adotar todas as cerimônias e práticas, elementos do Scrum, como a adaptabilidade, flexibilidade e a autogestão de equipes multifuncionais, são essenciais para o sucesso de projetos que exigem ajustes rápidos diante de adversidades e mudanças de escopo (SUTHERLAND, 2014; FERREIRA, 2017; RIBEIRO; MAGALHãES, 2023).

3.2 Extreme Programming (XP)

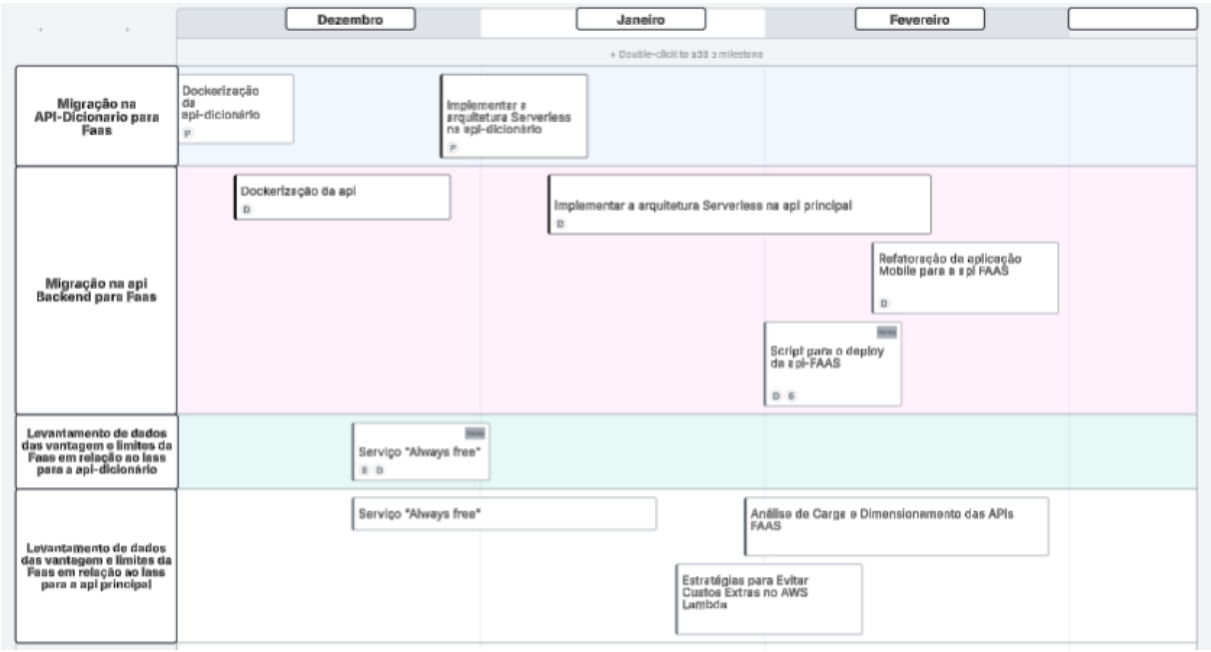
Extreme Programming (XP) é um método ágil de desenvolvimento de software iterativo e incremental, que tem como objetivo principal maximizar o valor entregue ao cliente a cada dia de trabalho da equipe. Criado por Kent Beck, o XP enfatiza a utilização de boas práticas de programação e desenvolvimento, levando essas práticas ao extremo para garantir qualidade e eficiência (PROGRAMMING, 2013). Os cinco valores fundamentais do XP — comunicação, simplicidade, coragem, respeito e feedback — orientam o comportamento da equipe de desenvolvimento e asseguram uma entrega contínua e eficiente de valor ao cliente (TELES, 2017). Entre as principais práticas adotadas no XP estão a programação em pares, desenvolvimento orientado a testes (TDD), refatoração, código coletivo, design simples e integração contínua. A interação constante com o cliente, por meio de feedbacks frequentes, e a realização de reuniões diárias (stand-up meetings) também são componentes essenciais para garantir que o desenvolvimento permaneça alinhado às necessidades do cliente e ao planejamento das entregas (BECK, 2000). A ênfase do XP na simplicidade, adaptação rápida às mudanças e ritmo sustentável torna esse método particularmente eficaz para equipes pequenas que precisam responder de forma

ágil a alterações de requisitos e garantir a alta qualidade do software entregue (BECK, 2000; TELES, 2017).

3.3 Roadmap

Um roadmap em engenharia de software é um plano estratégico que descreve a visão de desenvolvimento de um produto ou projeto ao longo do tempo. Após termos definido os objetivos específicos deste trabalho, elaboramos o roadmap para consolidarmos o planejamento das ações que serão executadas.

Figura 3 – Roadmap

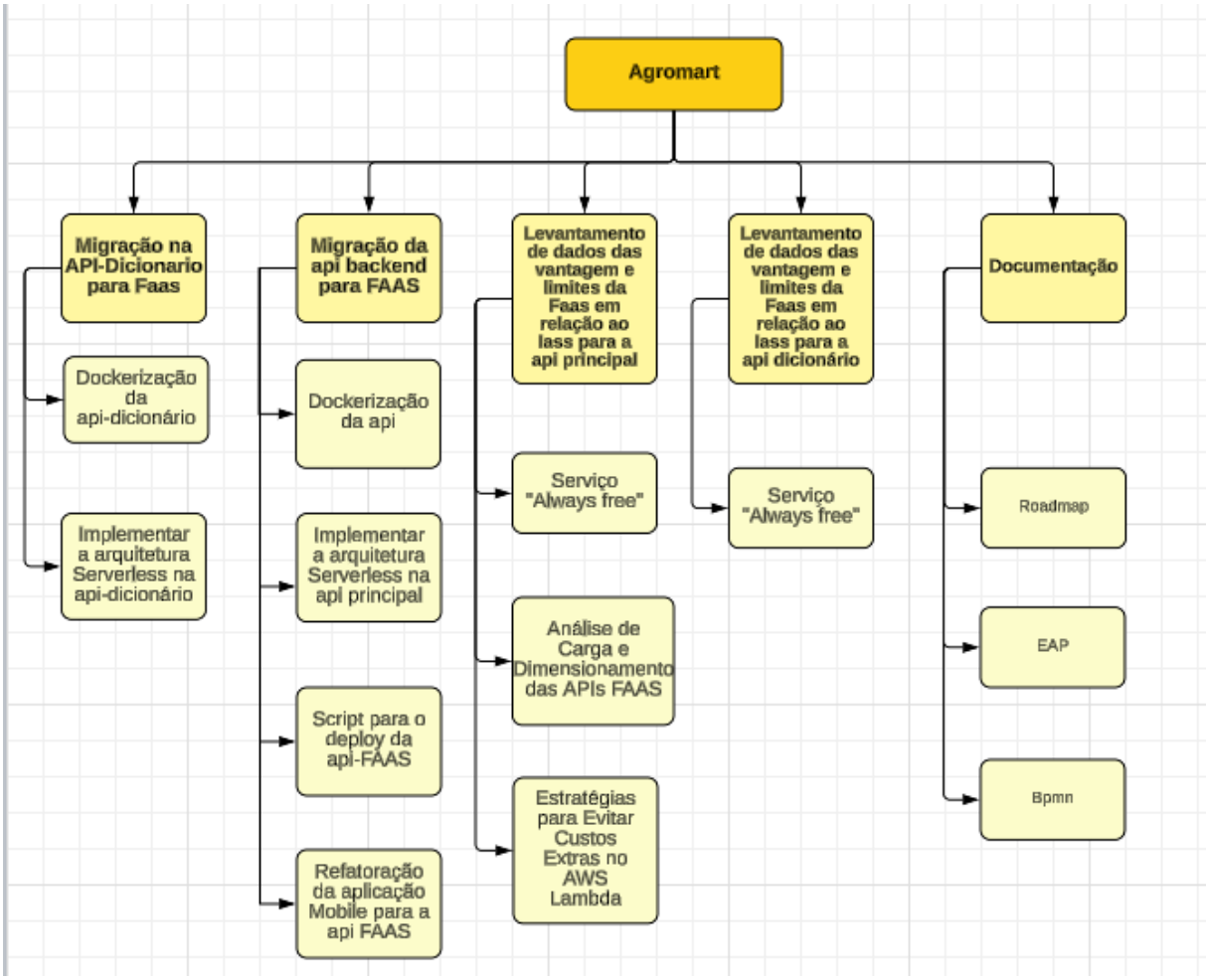


Roadmap AgroMart

3.4 Estrutura Analítica do Projeto(EAP)

É uma decomposição hierárquica das entregas do projeto. Todas as entregas que contém valor no projeto são dispostas de forma hierárquica e agrupadas em seus épicos. A seguir temos as entregas realizadas por este TCC.

Figura 4 – Estrutura Analítica do Projeto

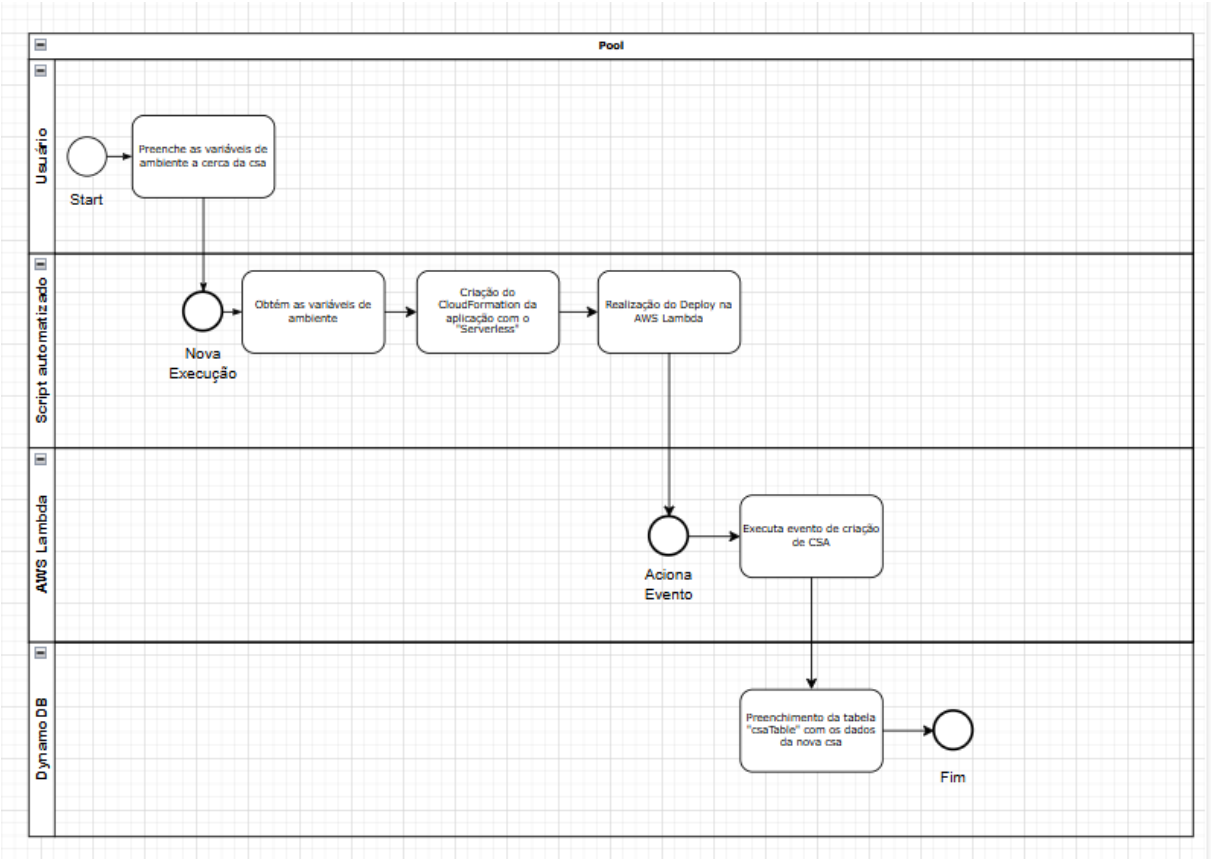


EAP AgroMart

3.5 Business Process Model and Notation

é uma notação padronizada para modelagem de processos de negócio. Ela permite representar visualmente fluxos de trabalho, tornando-os compreensíveis para analistas de negócios, desenvolvedores e stakeholders. Abaixo se encontra o diagrama BPMN do processo de criação de uma CSA como um todo:

Figura 5 – BPMN



BPMN do Script de criação de uma csa

3.6 Escolha de Ferramentas para a Migração

Para atingir o objetivo deste projeto de melhorar o custo-benefício utilizando serviços de infraestrutura oferecidos por provedores de nuvem, utilizaremos as seguintes ferramentas:

- **Javascript:**

  JavaScript é uma linguagem de programação de propósito geral, dinâmica, interpretada e amplamente utilizada no desenvolvimento de páginas e aplicativos web. Além disso, é a linguagem base do framework React Native, que é um framework desenvolvido pelo Facebook que permite criar aplicativos móveis para plataformas como iOS e Android, a qual o AgroMart foi escrito.

- **Typescript:**

  TypeScript é uma linguagem de programação de código aberto que adiciona tipagem estática e recursos avançados ao JavaScript. Ela é basicamente um superconjunto do JavaScript, o que significa que qualquer código JavaScript válido também é válido

em TypeScript, mas com a adição de tipos e outras funcionalidades que ajudam no desenvolvimento de software em larga escala.

- **Node.js:**

  É um ambiente de execução da linguagem JavaScript no lado do servidor construído sobre o motor V8 do Google Chrome. Ele permite que desenvolvedores usem JavaScript para criar aplicações no servidor. Isso significa que com Node.js, é possível usar JavaScript tanto no frontend (navegador) quanto no backend (servidor), facilitando o desenvolvimento full-stack. Um dos conceitos fundamentais do Node.js é que ele roda em uma única thread. Ao contrário de muitos servidores tradicionais que criam novas threads ou processos para lidar com cada requisição, Node.js usa um único thread principal para todas as requisições. Isso é possível porque ele lida com operações de I/O de forma assíncrona e não bloqueante.

- **Express.js:**

  Uma biblioteca para Node.js utilizada para criar as rotas HTTP que compõem os pontos de acesso da aplicação.

- **AWS DynamoDB Local:**

  Um banco de dados NoSQL altamente escalável, usado para armazenar e recuperar os dados das aplicações. A versão Local é uma versão emulada do DynamoDB usada durante o desenvolvimento local para simular as interações com o banco de dados.

- **Serverless-offline:**

  Este plugin permite a emulação do ambiente AWS Lambda em um ambiente de desenvolvimento, eliminando a necessidade de fazer deploy na AWS para testar a funcionalidade.

- **AWS API Gateway:**

  Serviço que gerencia e expõe as APIs da aplicação FaaS, permitindo que as funções Lambda sejam acessadas via HTTP.

- **DynamoDB - Admin:**

  Uma interface desenvolvida para gerenciar tabelas do Amazon DynamoDB, permitindo que administradores insiram, atualizem e excluam registros facilmente.

4 Implementações

4.1 Processo de Dockerização da API Strapi e da API Dicionário

Antes da migração para a arquitetura FaaS, foi necessário realizar a dockerização do sistema, permitindo um estudo prático e empírico das funcionalidades e regras de negócio do Agromart.

Para isso, foi criado um ambiente Docker Compose que define dois serviços principais: um responsável por instanciar a aplicação Strapi e outro para o banco de dados PostgreSQL. Essa abordagem possibilitou a validação do comportamento da API e do gerenciamento de dados antes da transição para um ambiente serverless, garantindo maior confiabilidade no processo de migração.

Código 4.1 – Service do Strapi

```
1
2  agromart_strapi_service:
3      container_name: agromart_strapi_container
4      build:
5        context: .
6        dockerfile: Dockerfile.strapi
7      env_file: .env
8      environment:
9        HOST: ${HOST}
10       PORT: ${PORT}
11       APP_KEYS: ${APP_KEYS}
12       API_TOKEN_SALT: ${API_TOKEN_SALT}
13       ADMIN_JWT_SECRET: ${ADMIN_JWT_SECRET}
14       JWT_SECRET: ${JWT_SECRET}
15       DATABASE_HOST: ${DATABASE_HOST}
16       DATABASE_PORT: ${DATABASE_PORT}
17       DATABASE_NAME: ${DATABASE_NAME}
18       DATABASE_USERNAME: ${DATABASE_USERNAME}
19       DATABASE_PASSWORD: ${DATABASE_PASSWORD}
20       EXPO_ACCESS_TOKEN: ${EXPO_ACCESS_TOKEN}
21       DATABASE_CLIENT: ${DATABASE_CLIENT}
22       DATABASE_SSL: ${DATABASE_SSL}
23     ports:
24       - "1337:1337"
25     volumes:
26       - .:/src/app
27       - agromart_node_modules:/src/app/node_modules
28     depends_on:
```

```
29        - agromart_db_service
```

Código 4.2 – Service do postgreSQL

```
1
2  agromart_db_service:
3      container_name: agromart_db_container
4      image: postgres
5      env_file: .env
6      environment:
7        POSTGRES_DB: ${DATABASE_NAME}
8        POSTGRES_USER: ${DATABASE_USERNAME}
9        POSTGRES_PASSWORD: ${DATABASE_PASSWORD}
10     ports:
11       - "5432:5432"
12     volumes:
13       - pgdata:/var/lib/postgresql/data
```

Código 4.3 – Dockerfile Strapi

```
1
2  FROM node:18-alpine
3  WORKDIR /src/app
4  COPY package.json ./
5  RUN yarn install --frozen-lockfile --verbose
6  COPY . .
7  RUN yarn build
8  CMD ["yarn","start"]
```

Código 4.4 – api dicionário Service

```
1
2  agromart_node_service:
3      container_name: dicionario_agromart_node_container
4      build:
5        context: .
6        dockerfile: Dockerfile.node
7      env_file: .env
8      environment:
9        - PORT=${PORT}
10       - NODE_ENV=${NODE_ENV}
11       - CORS_ALLOWED_ORIGIN=${CORS_ALLOWED_ORIGIN}
12       - JWT_SECRET_KEY=${JWT_SECRET_KEY}
13       - DB_DATABASE=${DB_DATABASE}
14       - DB_USERNAME=${DB_USERNAME}
15       - DB_PASSWORD=${DB_PASSWORD}
16       - DB_HOST=${DB_HOST}
17       - DB_PORT=${DB_PORT}
18       - DATABASE_URL=${DATABASE_URL}
```

```
19        - SMTP_HOST=${SMTP_HOST}
20        - SMTP_PORT=${SMTP_PORT}
21        - SMTP_USER=${SMTP_USER}
22        - SMTP_PASSWORD=${SMTP_PASSWORD}
23        - DEFAULT_MAIL_SENDER=${DEFAULT_MAIL_SENDER}
24      ports:
25        - "3000:3000"
26      volumes:
27        - .:/app
28      depends_on:
29        - agromart_db_service
```

4.2 Migração das APIs para um Ambiente Serverless

Com o objetivo de garantir a utilização totalmente gratuita do Agromart para proprietários de CSA (Community-Supported Agriculture), foi analisada a viabilidade de um serviço de hospedagem que permanecesse sem custos dentro dos limites do Free Tier da AWS. Identificou-se que a melhor solução seria a combinação do AWS Lambda, para execução das funções, e do Amazon DynamoDB, como banco de dados, desde que respeitados os limites estabelecidos pela AWS: 1 milhão de requisições mensais para aplicações FaaS e 25 GB de armazenamento no DynamoDB.

Diante dessa necessidade, foi realizada a migração da API Dicionário e do backend Strapi para uma arquitetura Serverless, utilizando o serviço AWS Lambda. Esse processo foi conduzido por meio do Framework Serverless, que facilita a implementação e o gerenciamento de aplicações FaaS.

Na nova estrutura, cada endpoint da aplicação original foi transformado em um handler, que tem a responsabilidade de processar as requisições recebidas e fornecer as respectivas respostas. A configuração principal da aplicação FaaS é definida no arquivo serverless.yml, onde são especificados os endpoints, seus respectivos handlers, a configuração das tabelas do DynamoDB, as permissões necessárias e demais parâmetros da infraestrutura.

Para possibilitar o desenvolvimento e os testes em um ambiente local, foi utilizado o plugin serverless-offline, permitindo a simulação do comportamento da aplicação dentro do ambiente AWS por meio do comando serverless offline. Já para a simulação do banco de dados localmente, foi adotado o plugin serverless-dynamodb, que não apenas possibilita a execução do DynamoDB de forma local, mas também fornece comandos auxiliares na CLI do Serverless Framework, como a verificação de tabelas específicas, consulta de dados e execução do banco de forma independente da aplicação FaaS por meio do comando serverless dynamodb start.

Além disso, a estrutura de pastas e arquivos nos projetos FaaS segue diretrizes específicas, garantindo organização e manutenção facilitada.

A estrutura de pastas e alguns arquivos específicos, para os projetos em FAAS, seguem algumas diretrizes :

- **/handlers**: diretório onde será salvo o arquivo "handler"que terá todos os métodos referentes a cada um dos endoints.
- **repository/dynamodb-client-config.js**: Arquivo que realiza configurações no DynamoDB e exporta métodos para a interação com o DynamoDB. Nele é realizado a instanciação do cliente "AWS.DynamoDB.DocumentClient(options)".
- **repository**: Diretório para os repositorios, esses que são responsáveis de se comunicar com o "dynamodb-client-config".
- **offline/migrations**: Arquivos criados mais para o entendimento futuro das tabelas do que implementação de fato. O "dynamoDB"cria as tabelas através desse arquivo. Além do entendimento de futuros mantenedores ajudou para deixar mais enxuto o arquivo serverless.yml.
- **/services**: Alguns "Handler's"se beneficiam do uso de services, para separação de responsabilidades. Este diretório é onde são armazenados os services criados.

4.2.1 Processo de migração da API dicionário

Os endpoints responsáveis pela criação e recuperação de uma ou múltiplas CSA's foram convertidos em três funções distintas, cada uma correspondendo a uma operação específica. Cada função possui um handler dedicado, responsável por processar as requisições e fornecer as respostas apropriadas.

Código 4.5 – Arquivo de rotas da aplicação original

```
1
2  import { Router } from 'express';
3
4  import * as csaController from '@/controllers/csa';
5
6  const router = Router();
7
8  router.route('/')
9    .get(csaController.getCsas)
10   .post(csaController.createCsa);
11
12   router.route('/:id')
13   .get(csaController.getcsaById)
```

```
14
15 export default router;
```

Código 4.6 – Configuração no serverless.yml para os endpoints

```
1
2  functions:
3      obter:
4          handler: handler.getCsas
5          events:
6              - http:
7                    path: /
8                    method: get
9                    cors: true
10     obterUm:
11         handler: handler.getcsaById
12         events:
13             - http:
14                   path: /{id}
15                   method: get
16                   cors: true
17     create:
18         handler: handler.createCsa
19         events:
20             - http:
21                   path: /
22                   method: post
23                   cors: true
```

4.2.2 Processo de migração da API Backend

A migração da API Strapi apresentou maior complexidade devido ao seu tamanho e às suas regras de negócio, que diferem da API Dicionário. No entanto, a abordagem utilizada foi similar: para cada entidade, foi desenvolvido um handler contendo métodos que substituem os endpoints da API original.

Dado o volume de código da aplicação FaaS, uma análise completa seria inviável e pouco produtiva. Portanto, este trabalho se concentrará nos aspectos mais relevantes da migração, destacando as diferenças em relação à API Dicionário.

- **Populando o retorno de todas as lojas**

  Por se tratar de um banco de dados NoSQL, o relacionamento entre as entidades é armazenado diretamente no documento da entidade, por meio de uma propriedade do tipo lista. Optamos por salvar apenas os IDs das entidades relacionadas,

exigindo uma busca adicional sempre que for necessário popular os dados completos da entidade referenciada.

Para otimizar esse processo, utilizamos o método dynamoDb.batchGet(params) no arquivo dynamo-client-config. Essa abordagem permite realizar uma única requisição para recuperar todos os IDs armazenados no array de relacionamentos, evitando múltiplas requisições individuais para cada item. A implementação foi realizada no arquivo LojaRepository.js, garantindo a correta recuperação e associação de dados entre Planos, Cestas e Produtos Avulsos.

Código 4.7 – Lojas Repository

```
1
2     const getLojasComRelacionamentos = async function () {
3     try {
4         const result = await getAllLojas();
5         const lojas = result.Items;
6         let idsCestas = new Set();
7         let idsPlanos = new Set();
8         let idsProdutos = new Set();
9
10        // Coletando IDs  nicos  de cestas, planos e produtos
11        lojas.forEach((loja) => {
12            (loja.cestas || []).forEach((id) => idsCestas.add(id));
13            (loja.planos || []).forEach((id) => idsPlanos.add(id));
14            (loja.produto_avulsos || []).forEach((id) =>
                  idsProdutos.add(id));
15        });
16
17        // Buscando todas as entidades relacionadas de uma s   vez
18        const [cestas, planos, produtos] = await Promise.all([
19            dynamoDbLib.batchGet("CestasTable", Array.from(
                  idsCestas)),
20            dynamoDbLib.batchGet("PlanosTable", Array.from(
                  idsPlanos)),
21            dynamoDbLib.batchGet(
22                "ProdutosAvulsosTable",
23                Array.from(idsProdutos)
24            ),
25        ]);
26
27        // Criando um mapa de ID     Objeto para facilitar
              a s s o c i a   o
28        const cestasMap = Object.fromEntries(cestas.map((c) => [c.
              id, c]));
29        const planosMap = Object.fromEntries(planos.map((p) => [p.
              id, p]));
```

```
30          const produtosMap = Object.fromEntries(produtos.map((p) =>
                [p.id, p]));
31
32          // Preenchendo os relacionamentos dentro de cada loja
33          lojas.forEach((loja) => {
34              loja.cestas = (loja.cestas || []).map(
35                  (id) => cestasMap[id] || null
36              );
37              loja.planos = (loja.planos || []).map(
38                  (id) => planosMap[id] || null
39              );
40              loja.produto_avulsos = (loja.produto_avulsos || []).map
                    (
41                  (id) => produtosMap[id] || null
42              );
43          });
44
45          return lojas;
46      } catch (error) {
47          throw new Error(
48              "Erro ao buscar lojas com relacionamentos: " + error.
                    message
49          );
50      }
51  };
```

Código 4.8 – Método getBach no arquivo dynamo-client-config

```
1
2       const batchGet = async (tableName, ids) => {
3       if (ids.length === 0) return [];
4
5       const params = {
6           RequestItems: {
7               [tableName]: {
8                   Keys: ids.map((id) => ({ id })),
9               },
10          },
11      };
12
13      try {
14          const result = await dynamoDb.batchGet(params).promise();
15          return result.Responses[tableName] || [];
16      } catch (error) {
17          throw new Error(
18              ‘Erro ao buscar m ltiplos itens em ${tableName}: ‘ +
                    error.message
19          );
```

```
20      }
21  };
```

4.3 Automação do Deploy da API Principal em FaaS

O objetivo deste script é automatizar o processo de deploy, eliminando a necessidade de digitar manualmente todos os comandos da AWS CLI e do Serverless Framework. Além disso, ele unifica e define claramente as variáveis que precisam ser preenchidas, garantindo uma execução mais eficiente e padronizada.

Ao final do processo, o script realiza um POST para a API Dicionário, enviando a URL da nova CSA implantada. Para maior acessibilidade, o script foi desenvolvido para ser executado em Windows e já inclui a instalação automática das dependências necessárias.

Código 4.9 – Script de automação do Deploy

```
1
2       # Define as vari veis que o cliente deve preencher
3   $AWS_ACCESS_KEY_ID = "AKIA2S2Y36QFEEUPKO6F"
4   $AWS_SECRET_ACCESS_KEY = "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"
5   $AWS_REGION = "us-east-1"
6   $PROJECT_DIR = $PSScriptRoot # Caminho da aplica  o Serverless
7
8   $URL_API_DICIONARIO = "https://aywcbxk6ql.execute-api.us-east-1.
        amazonaws.com/dev/"
9   $NOME_CSA = "CSA SCRIPT"
10  $NOME_RESPONSAVEL = "Murilo SCRIPT"
11  $EMAIL_CSA = "EMAIL@email.com"
12
13  # Fun   o para verificar se um comando existe
14  function CommandExists {
15      param (
16          [string]$command
17      )
18      $exists = $false
19      try {
20          if (Get-Command $command -ErrorAction Stop) {
21              $exists = $true
22          }
23      } catch {}
24      return $exists
25  }
26
27  # Instalar Node.js, se necess rio
28  if (-not (CommandExists "node")) {
```

```
29      Write-Output "Instalando Node.js..."
30      Invoke-WebRequest -Uri "https://nodejs.org/dist/v18.17.1/node-
            v18.17.1-x64.msi" -OutFile "nodejs.msi"
31      Start-Process -FilePath "nodejs.msi" -Wait -ArgumentList "/
            quiet"
32      Remove-Item "nodejs.msi"
33  }
34
35  # Instalar Serverless Framework, se necess rio
36  if (-not (CommandExists "serverless")) {
37      Write-Output "Instalando Serverless Framework..."
38      npm install -g serverless
39  }
40
41  # Instalar AWS CLI, se necess rio
42  if (-not (CommandExists "aws")) {
43      Write-Output "Instalando AWS CLI..."
44      Invoke-WebRequest -Uri "https://awscli.amazonaws.com/AWSCLIV2.
            msi" -OutFile "AWSCLIV2.msi"
45      Start-Process -FilePath "AWSCLIV2.msi" -Wait -ArgumentList "/
            quiet"
46      Remove-Item "AWSCLIV2.msi"
47  }
48
49  # Configurar credenciais da AWS
50  aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
51  aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY
52  aws configure set region $AWS_REGION
53
54  # Navegar at  o diret rio do projeto
55  Set-Location $PROJECT_DIR
56
57  # Instalar depend ncias do projeto
58  npm install
59
60  # Fazer deploy
61  serverless deploy 2>&1 | Tee-Object -Variable deployOutput
62
63  # Capturar URL Base da API
64  $apiUrls = $deployOutput | Select-String -Pattern "(https://[a-zA-
        Z0-9.-]+\.amazonaws\.com/[a-zA-Z0-9/-]+)" -AllMatches | ForEach-
        Object { $_.Matches.Value }
65  $baseApiUrl = $apiUrls | Select-String -Pattern "^https
        :\/\/[^\/]+\/dev" -AllMatches | Select-Object -First 1 | ForEach
        -Object { $_.Matches.Value }
66
```

```
67 # Requisicao POST para a api dicionario, salvando nela a URL da csa
       criada
68
69 $headers = @{
70     "Content-Type" = "application/json"
71 }
72
73 $body = @{
74     nomeCSA        = "$NOME_CSA"
75     responsavelCSA = "$NOME_RESPONSAVEL"
76     emailCSA       = "$EMAIL_CSA"
77     urlBase        = "$baseApiUrl"
78 } | ConvertTo-Json -Depth 10
79
80 Invoke-WebRequest -Uri "$URL_API_DICIONARIO" `
81     -Method Post `
82     -Headers $headers `
83     -Body $body `
84     -UseBasicParsing | Out-Null
```

4.4 Refatoração do Mobile para consumir a API em FaaS

Com a migração do backend do Agromart para uma arquitetura baseada em FaaS, foi necessário realizar ajustes no aplicativo móvel para garantir a compatibilidade com a nova API hospedada na AWS Lambda. Essas modificações foram essenciais para adequar o consumo de requisições sob demanda, otimizando a comunicação entre o app e os serviços em nuvem.

4.4.1 Adaptação das Requisições para a AWS Lambda

A principal mudança na refatoração foi a alteração dos endpoints utilizados no aplicativo. No modelo anterior, as requisições eram feitas para um servidor EC2 com URLs estáticas e pré-definidas. Com a adoção do FaaS, foi necessário modificar as chamadas para utilizar os endpoints da AWS API Gateway, que atuam como intermediários entre o aplicativo e as funções Lambda.

Os ajustes envolveram:

- Atualização das URLs das requisições para os novos endpoints gerenciados pelo API Gateway.
- Modificação do formato das requisições, garantindo compatibilidade com a execução assíncrona das funções Lambda.

- Implementação de novas regras de autenticação, caso necessário, considerando a possível integração com AWS IAM ou API Keys.

4.4.2 Atualizações no Código do Aplicativo

Para ilustrar as alterações feitas no código, a seguir são apresentados os trechos modificados. Esses diffs foram extraídos do histórico de commits no GitHub e demonstram a adaptação do código para consumir a API em FaaS corretamente.

```
43   const handleSubmit = useCallback(async (data: { CsaCode: string; }) => {
44     setLoading(true);
45
46     try {
47-      const resp = await apiDicionario.get(`csa/${data.CsaCode}`);
48-      console.log(resp.data);
49-      if (resp.data) {
50-        setChosenCsa(resp.data);
51       }
52     } catch (error) {
53       Alert.alert('CSA não encontrada!');
54       console.log(JSON.stringify(error));
55     } finally {
56       setLoading(false);
57     }
58   }, []);
```

Figura 6 – Procurar CSA - antes

```
43  const handleSubmit = useCallback(async (data: { CsaCode: string; }) => {
44    setLoading(true);
45
46+   const urlApiDicionario = 'http://10.0.2.2:4000/dev'; // URL da funcao lambda da API Dicionario
47+
48    try {
49+     const resp = await apiDicionario.get(`${urlApiDicionario}/${data.CsaCode}`);
50+     console.log(JSON.stringify(resp.data));
51+     if (resp.data?.Item) {
52+       setChosenCsa(resp.data?.Item);
53      }
54    } catch (error) {
55      Alert.alert('CSA não encontrada!');
56      console.log(JSON.stringify(error));
57    } finally {
58      setLoading(false);
59    }
60  }, []);
```

Figura 7 – Procurar CSA - depois

```
35  const handleSubmit = useCallback(
36    async (data: { city: string; number: string; complement: string; street: string; cep: string; neighborhood: string; }) => {
37      try {
38        const api = await initializeApi()
39
40        setLoading(true);
41
42        const body = {
43          cidade: data.city,
44          numero: data.number,
45          complemento: data.complement,
46          rua: data.street,
47          cep: data.cep,
48          bairro: data.neighborhood,
49-         user: user.id,
50        };
51
52        let response;
53
54-       if (user.endereco) {
55-         response = await api.put(`enderecos/${user.endereco.id}`, body);
56          Alert.alert('Deu tudo certo:', 'Endereço editado com sucesso');
57        } else {
58-         response = await api.post('/enderecos', body);
59          Alert.alert('Endereço criado com sucesso');
60        }
61
62-       await updateAddress(response.data);
63        navigation.goBack();
64      } catch (error) {
65        Alert.alert('Erro ao cadastrar endereço');
66      }
67      setLoading(false);
68    },
69    [navigation, user, updateAddress],
70  );
```

Figura 8 – Ajuste de requisição e de url Endereço - antes

```
25  const AddressForm: React.FC = () => {
35    const handleSubmit = useCallback(
36      async (data: { city: string; number: string; complement: string; street: string; cep: string; neighborhood: string; }) => {
38        const api = await initializeApi()
39
40        setLoading(true);
41
42        const body = {
43          cidade: data.city,
44          numero: data.number,
45          complemento: data.complement,
46          rua: data.street,
47          cep: data.cep,
48          bairro: data.neighborhood,
49+         userId: user.id,
50        };
51
52        let response;
53
54+       if (user.endereco?.id) {
55+         response = await api.put(`endereco/${user.endereco?.id}`, body);
56          Alert.alert('Deu tudo certo:', 'Endereço editado com sucesso');
57        } else {
58+         response = await api.post('/endereco', body);
59          Alert.alert('Endereço criado com sucesso');
60        }
61
62+       await updateAddress(response.data?.data).then(() => console.log('response.data: ' + JSON.stringify(response.data.data)));
63        navigation.goBack();
64      } catch (error) {
65        Alert.alert('Erro ao cadastrar endereço');
66      }
67      setLoading(false);
68    },
69    [navigation, user, updateAddress],
70  );
```

Figura 9 – Ajuste de requisição e de url Endereço - depois

```
102   const signIn = useCallback(
103     async ({ username, password }: SignInCredentials) => {
104       const url = await AsyncStorage.getItem('@BaseUrlChosen');
105       console.log('LOGANDO DENTRO', url);
106-      const response = await axios.post(`${url}auth/local`, {
107-        identifier: username,
108-        password,
109-      });
110
111       const { jwt: token, user } = response.data;
112
113       console.log('TOKEN', token);
114
115       await AsyncStorage.multiSet([
116         ['@Agromart:token', token],
117         ['@Agromart:user', JSON.stringify(user)],
118       ]);
119
120       setData({ token, user });
121
122       await registerDeviceInfo(user.id);
123     },
124     [],
125   );
```

Figura 10 – Ajuste de parâmetro e url Login - antes

```
const signIn = useCallback(
  async ({ username, password }: SignInCredentials) => {
    const url = await AsyncStorage.getItem('@BaseUrlChosen');
    console.log('LOGANDO DENTRO', url);
    const credenciais = {
      email: username,
      senha: password
    }

    const response = await axios.post(`${url}/auth`, credenciais);

    const { jwt: token, user } = response.data;

    console.log('TOKEN', token);

    await AsyncStorage.multiSet([
      ['@Agromart:token', token],
      ['@Agromart:user', JSON.stringify(user)],
    ]);

    setData({ token, user });

    await registerDeviceInfo(user.id);
  },
  [],
);
```

Figura 11 – Ajuste de parâmetro e url Login - depois

```
127  const signUp = useCallback(
128-   async ({ username, password, email }: SignUpCredentials) => {
129      // TODO REMOVE THIS HARD CODED URL THIS IS JUST FOR TESTING
130      const baseUrl =
131-       (await AsyncStorage.getItem('@BaseUrlChosen')) ||
132-       'https://agromarttcc.shop/api/';
133
134-     const response = await axios.post(`${baseUrl}auth/local/register`, {
135-       username,
136-       password,
137        email,
138      });
139
140      const { jwt: token, user } = response.data;
141
142      await AsyncStorage.multiSet([
143        ['@Agromart:token', token],
144        ['@Agromart:user', JSON.stringify(user)],
145      ]);
146
147      setData({ token, user });
148
149      await registerDeviceInfo(user.id);
150    },
151    [],
152  );
```

Figura 12 – Ajuste de parâmetro e url Cadastro - antes

```
130+   async ({ nome, senha, email }: SignUpCredentials) => {
131      // TODO REMOVE THIS HARD CODED URL THIS IS JUST FOR TESTING
132      const baseUrl =
133+       (await AsyncStorage.getItem('@BaseUrlChosen'))
134
135+     const response = await axios.post(`${baseUrl}/usuarios`, {
136+       nome,
137+       senha,
138        email,
139      });
140
141      const { jwt: token, user } = response.data;
142
143      await AsyncStorage.multiSet([
144        ['@Agromart:token', token],
145        ['@Agromart:user', JSON.stringify(user)],
146      ]);
147
148      setData({ token, user });
149
150      await registerDeviceInfo(user.id);
151    },
152    [],
153  );
```

Figura 13 – Ajuste de parâmetro e url Cadastro - depois

```
20  const handleSave = async () => {
21    try {
22      const api = await initializeApi();
23-     const { data } = await api.put(`users/${user.id}`, {
24-       ...user,
25        username: name,
26        email,
27      });
28
29      await updateUser(data);
30
31      Alert.alert('Tudo certo :)', 'Dados atualizados com sucesso!');
32    } catch (err) {
33      Alert.alert('Ops :(', 'Não foi possivel atualizar seus dados');
34    }
35  };
```

Figura 14 – Ajuste de parâmetro e url Usuário - antes

```
20  const handleSave = async () => {
21    try {
22      const api = await initializeApi();
23+     var { data } = await api.put(`usuarios/${user.id}`, {
24        username: name,
25        email,
26      });
27+
28+     data = {
29+       username: data.nome,
30+       email: data.email,
31+       id: user.id
32+     }
33
34      await updateUser(data);
35
36      Alert.alert('Tudo certo :)', 'Dados atualizados com sucesso!');
37    } catch (err) {
38      Alert.alert('Ops :(', 'Não foi possivel atualizar seus dados');
39    }
40  };
```

Figura 15 – Ajuste de parâmetro e url Usuário - depois

4.4.3 Impactos da Refatoração

A refatoração trouxe benefícios significativos para o funcionamento do aplicativo móvel, incluindo:

- Redução do tempo de resposta das requisições em cenários de baixa demanda, pois os servidores não permanecem ativos o tempo todo.
- Otimização do consumo de recursos, reduzindo custos operacionais devido à natureza sob demanda da arquitetura FaaS.

- Facilidade de escalabilidade, permitindo que as funções Lambda se ajustem automaticamente conforme o volume de requisições.

4.5 Análise de Serviço "Sempre Gratuito"da AWS

Após conversas com o orientador Prof. Dr. André Luiz Lanna, foi levantada a necessidade de se ter um ambiente de deploy totalmente gratuito para o agricultor. Escolhemos a melhor combinação no mercado que é sempre gratuito, respeitando algumas limitações, que é o serviço Lambda junto ao DynamoDB.

Os limites impostos pela AWS para a Lambda e para o DynamoDB são estes (SERVICES, 2025a) (SERVICES, 2025b) :

- AWS Lambda:

- Solicitações mensais gratuitas: 1 milhão de invocações.
- Tempo de computação gratuito: 400.000 GB-segundos por mês.
  GB-segundo mede o consumo de memória ao longo do tempo. Por exemplo, uma função Lambda com 1 GB de memória que executa por 1 segundo consome 1 GB-segundo.
- Transferência de dados: 100 GiB de respostas HTTP por mês, além dos primeiros 6 MB por solicitação, que são gratuitos.

- DynamoDB:

- Armazenamento: 25 GB gratuitos.
- Capacidade provisionada: 25 unidades de capacidade de leitura (RCUs) e 25 unidades de capacidade de gravação (WCUs), suficientes para processar até 200 milhões de solicitações por mês.

  Unidade de leitura: 1 unidade permite até 2 leituras por segundo para itens de até 4 KB (fortemente consistente).
  Unidade de escrita: 1 unidade permite 1 escrita por segundo para itens de até 1 KB.

  Exemplo prático: Se sua aplicação precisa ler e escrever itens pequenos (até 1 KB), as unidades gratuitas permitem:

  Até 50 leituras/segundo (eventualmente consistentes) ou 25 leituras/segundo (fortemente consistentes) Até 25 gravações/segundo

4.5.1 Por que a AWS é melhor que as opções "ilimitadas"do Google e Oracle?

Três grandes plataformas se destacam para a implantação de projetos FaaS: AWS, Google Cloud e Oracle. Mas, por que a AWS foi a escolha?

A AWS se destaca na arquitetura FaaS e bancos NoSQL gratuitos porque oferece o único plano verdadeiramente ilimitado para pequenas e médias aplicações. Mas para entender por que o Google Cloud e o Oracle Cloud não são tão vantajosos, é fundamental analisar as limitações ocultas dessas plataformas.

- AWS X Google Cloud

- **Execuções FaaS grátis:** Nesse ponto o Goolgle oferece o dobro de requisições gratuitas que a AWS, 2 milhões.
- **Armazenamento gratuito** O Firestore disponibiliza somente 1 GB de armazenamento gratuito equanto que a AWS libera 25gb dedicados a cada aplicação.
- **Leituras e Gravações** Aqui já temos os limites rígidos inviabilizando o Google Cloud. 50.000/dia de leituras gratuitas (depois cobra 0.06 dólares por 100K leituras), 20.000/dia de escritas gratuitas (depois cobra 0.18 dólares por 100K gravações) e 10.000/dia de deleções gratuitas (depois cobra 0.02 por 100K exclusões). Google Cloud cobra por cada requisição diferente da AWS que caso você não passe das 25/50 unidades de leitura e 25 de escrita não será cobrado nunca.(GOOGLE, 2025a)(GOOGLE, 2025b)

  Resumo : Se sua aplicação faz 100.000 leituras/dia, no DynamoDB é gratuito (dentro das 25 unidades provisionadas), mas no Firestore custa 0.06/dia → 1.80/mês. Se precisar de 500.000 gravações/dia, o custo no Firestore sobe para 0.90/dia → 27/mês.

- AWS X Oracle

- **Execuções FaaS grátis:** Também oferece o dobro comparado a AWS.
- **Armazenamento gratuito:** 25gb dedicados a cada aplicação. Já a "Oracle"disponibiliza os 25 GB porém compartilhado entre todos os serviços !
- **Leituras e Gravações:** A "AWS"é Ilimitada dentro das 25/50 RCU e 25 WCU. Já a Oracle 133 milhões de RCU (depois cobra 0.30 de dólares por milhão) e 25 milhões WCU (depois cobra 2.25 por milhão) (CLOUD, 2025)

Resumo: No DynamoDB é gratuito dentro da capacidade provisionada. Na Oracle se sua aplicação fizer 150 milhões de leituras/mês custa: 17 milhões extras × 0.30/milhão = 5.10/mês. Se fizer 50 milhões de gravações/mês, no Oracle custa: 25 milhões extras × 2.25/milhão = 56.25/mês

A AWS foi escolhida pois trabalha com limites de de taxa de transmissão, e não limites estáticos. Caso seja respeitados estas taxas não será cobrado do agricultor. Nos entrega maior previsibilidade e controle de custos. Flexibilidade para crescer sem cobranças inesperadas

4.6 Estratégias para Minimização de Custos no AWS Lambda

No serviço Lambda :

- **Definir um limite de simultaneidade (tag "reservedConcurrency")**

  Bloqueia o número máximo de execuções simultâneas. Se atingir o limite, novas execuções serão enfileiradas ou rejeitadas.

Código 4.10 – reservedConcurrency em uma function

```
1
2     getLojas:
3         handler: handlers/loja.getLojas
4         reservedConcurrency: 5
5         events:
6             - http:
7                   path: lojas
8                   method: get
9                   cors: true
```

- **Configurar Timeout (tag "timeout")** Evita funções rodando por muito tempo e consumindo mais GB-segundos.

Código 4.11 – "reservedConcurrency"em uma function

```
1
2     provider:
3         timeout: 5
```

- **Reduzir Memória Alocada (tag "memorySize")** Menos memória alocada para cada function evitando o consumo de GB-segundos.

Código 4.12 – "reservedConcurrency"em uma function

```
1
```

```
    provider:
        memorySize: 128
```

No Amazon DynamoDB :

- **Configurar "Throttling"**

  Se a capacidade provisionada for atingida, o banco não cobra extra, apenas torna as requisições mais lentas ou as rejeita com erro HTTP 400 (ProvisionedThroughputExceededException).Essa condição é atingida configurando as tags "BillingMode"e "ProvisionedThroughput".

Código 4.13 – BillingMode em uma tabela

```

    LojasTable:
            Type: AWS::DynamoDB::Table
            Properties:
                TableName: LojasTable
                AttributeDefinitions:
                    - AttributeName: id
                      AttributeType: S
                KeySchema:
                    - AttributeName: id
                      KeyType: HASH
                BillingMode: PROVISIONED
                ProvisionedThroughput:
                    ReadCapacityUnits: 1
                    WriteCapacityUnits: 1
```

- **Configurar o "auto scaling"do "DynamoDB":**

  Queremos ter uma escalabilidade controlada . O Auto Scaling aumenta somente até 3 unidades na leitura, evitando gastos excessivos.

Código 4.14 – auto scaling

```

AgroMartTableScalingRead:
    Type: AWS::ApplicationAutoScaling::ScalableTarget
    Properties:
        MaxCapacity: 3
        MinCapacity: 1
        ResourceId: table/AgroMartTable
        RoleARN: arn:aws:iam::${aws:accountId}:role/
           DynamoDBAutoScalingRole
        ScalableDimension: dynamodb:table:ReadCapacityUnits
        ServiceNamespace: dynamodb
```

```
11
12 AgroMartTableScalingWrite:
13     Type: AWS::ApplicationAutoScaling::ScalableTarget
14     Properties:
15        MaxCapacity: 3
16        MinCapacity: 1
17             ResourceId: table/AgroMartTable
18             RoleARN: arn:aws:iam::${aws:accountId}:role/
                   DynamoDBAutoScalingRole
19         ScalableDimension: dynamodb:table:WriteCapacityUnits
20         ServiceNamespace: dynamodb
```

Com essas configurações visamos limitar a autonomia da AWS para com a nossa aplicação. Definimos limites para que que o backend do Agromart esteja preso e obtenhamos uma segurança e confiança maior na questão de custos.

4.7 Análise de Carga e Dimensionamento das APIs FAAS

4.7.1 Levantamento de dados sobre a api-dicionário em FAAS

Neste trecho, serão levantados dados sobre as duas APIs, analisando os impactos de cada uma sobre o *Always Free Tier* da AWS. A seguir, será detalhado em que momentos do uso do *AgroMart* são feitas requisições para a API do *Dicionário*.

As requisições para a API do Dicionário ocorrem em quatro momentos específicos:

- Ao criar uma nova *CSA*.
- Ao recuperar múltiplas *CSAs*.
- Ao recuperar uma única *CSA*.
- Ao deletar uma *CSA*.

Quando o usuário seleciona o ID de uma *CSA* no aplicativo móvel, esse ID é salvo para as próximas utilizações, ou seja, a busca pelo ID de uma *CSA* ocorre apenas uma vez por sessão de uso. ((FREITAS; CELLA, 2023))

Quais cenários de testes têm a tendência de bater 1 milhão de requisições no mês ?

- **1 cenário**: O usuário busca todas as csa's e escolhe uma para se acessar.

  Somente 1 requisição ao endpoint "GetAll"é utilizado.

- **2 cenário**:Os administradores Decidem criar mais uma csa.

  Somente 1 requisição ao endpoint "Creat"é utilizado".

- **3 cenário**:Os administradores buscam todas as csas para ver em algo em específico dentre elas.

  Somente 1 requisição ao endpoint "GetAll"é necessário, visto que esse endpoint já retorna alumas informações importantes das CSA's.

- **4 cenário**:Os administradores buscam uma em específico para análise.

  Somente 2 requisição com os endpoint's "GetAll"mais o "GetById".

- **5 cenário**:Os administradores buscam deletar uma csa.

  Somente 1 requisição ao endpoint "Delete".

Será que será extrapolado o armazenamento máximo do free tier do Dynamo?

Após a realização de diversas implantações de csa's e popularmos a api-dicionário com suas "URl's", vimos que em média o tamanho médio de um item no DynamoDB da api-dicionário se dá por volta dos **170 Bytes.**

Discussão dos resultados

Após o levantamento de dados podemos ver o pouquissímo impacto da api-dicionário nos limites estabelecidos para o "Aways Free tier"da AWS.

As funcionalidades que necessitam da api dicionária são extremamente pontuais e não ordinárias. São decisões tomadas no nível gerencial de uma csa, o que não acarreta medo a cerca do limite provisionado da AWS para requisições.

E sobre o banco de dados seriam necessárias quase que 150 milhões de csa's para se extrapolar o limite da Aws.

4.7.2 Análise de Carga e Dimensionamento de Usuários na API Principal FAAS

Esta sessão visa analisar a quantidade de requisições recebidas por uma API desenvolvida em AWS Lambda, identificando o consumo por usuário e determinando a quantidade máxima de usuários suportados dentro do limite gratuito de um milhão de requisições por mês.

4.7.2.1 Cenários de caso de uso do backend

Para entender o impacto da utilização da API e calcular a quantidade de requisições geradas por usuário, foi realizado um mapeamento detalhado de todos os casos de

uso da aplicação. Esse processo envolveu a identificação de cada funcionalidade disponível para o usuário e a análise das interações que resultam em chamadas à função AWS Lambda.

O mapeamento foi documentado por meio de diagramas de casos de uso, que estão listados logo abaixo, ilustrando as principais operações executadas pelos usuários e a frequência estimada de cada uma delas. A partir dessa análise, foi possível calcular a quantidade média de requisições geradas por um único usuário durante um período de um mês.

4.7.2.2 Casos de Uso

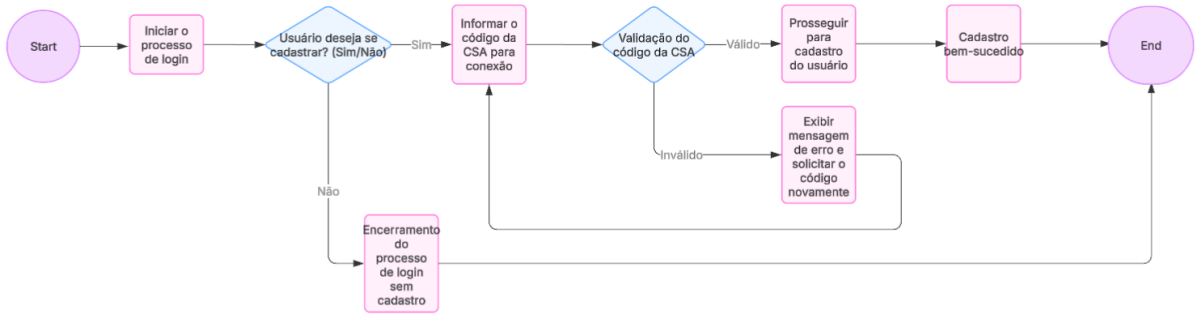


Figura 16 – Fluxo Cadastro de Usuário

- Requisição 1: Validar CSA na Api Dicionario

  GET: .../dev/idCsa

- **Requisição 2**: Criar usuário no backend

  POST: .../dev/usuarios.

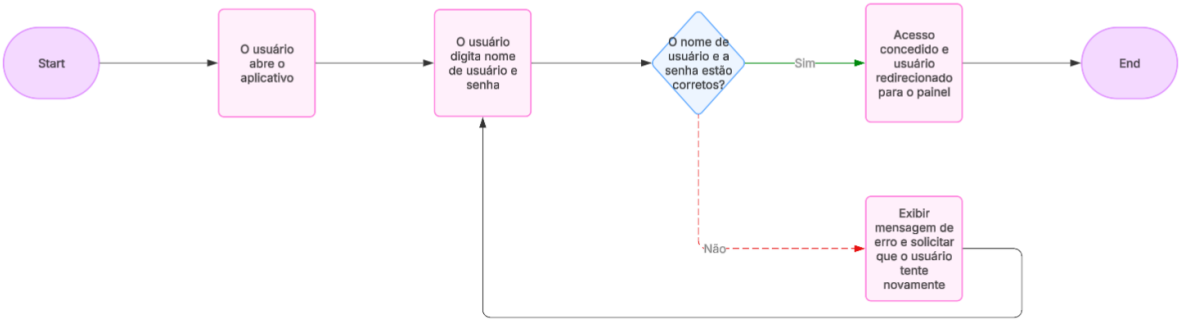


Figura 17 – Fluxo Login

- **Requisição 1**: Verificar Usuário e Senha informados

  POST: .../dev/auth

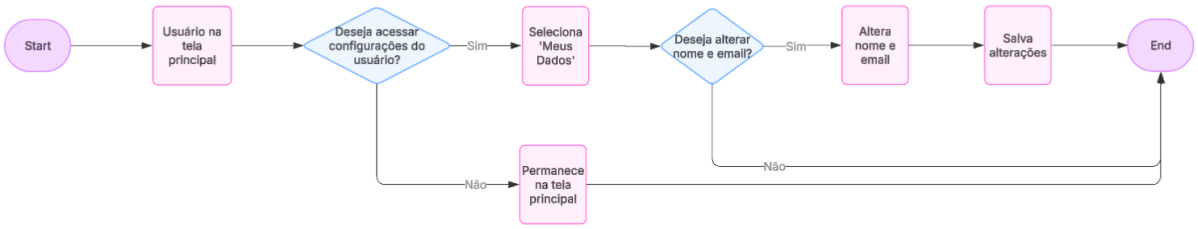


Figura 18 – Fluxo Meus Dados

- **Requisição 1**: Atualizar dados usuário

  PUT .../dev/usuarios/idUsuario

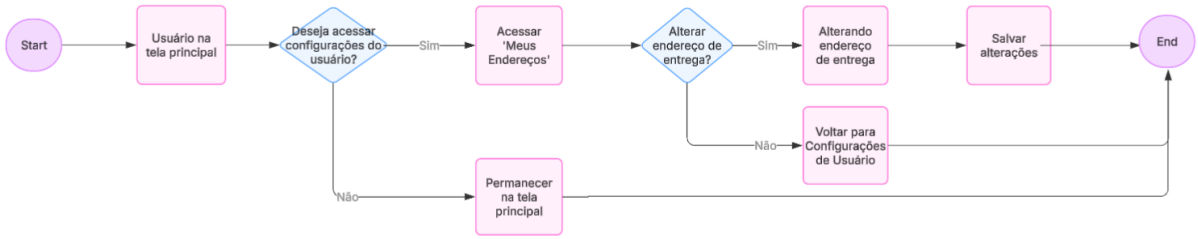


Figura 19 – Fluxo Meus Endereços

- **Requisição 1**: Criar/Atualizar endereço

  PUT .../dev/endereco

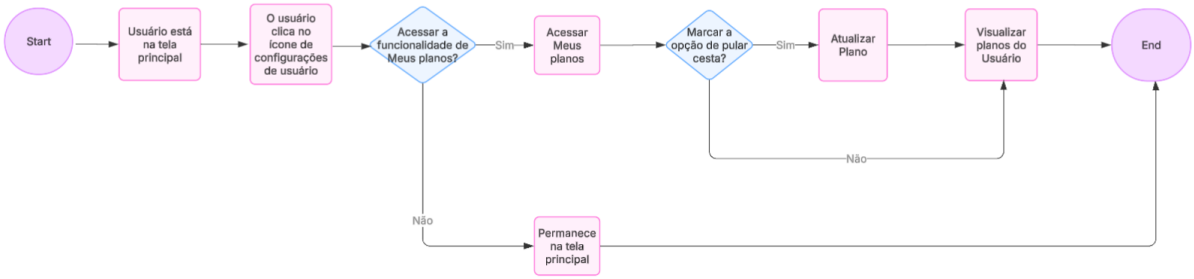


Figura 20 – Fluxo Planos

- **Requisição 1**: Procurar Planos

  GET ...dev/assinantes

- **Requisição 2**: Pular Cesta

  PUT ...dev/assinantes/70d38276-6d0e-4af2-b3d8-c91bd556e3fe

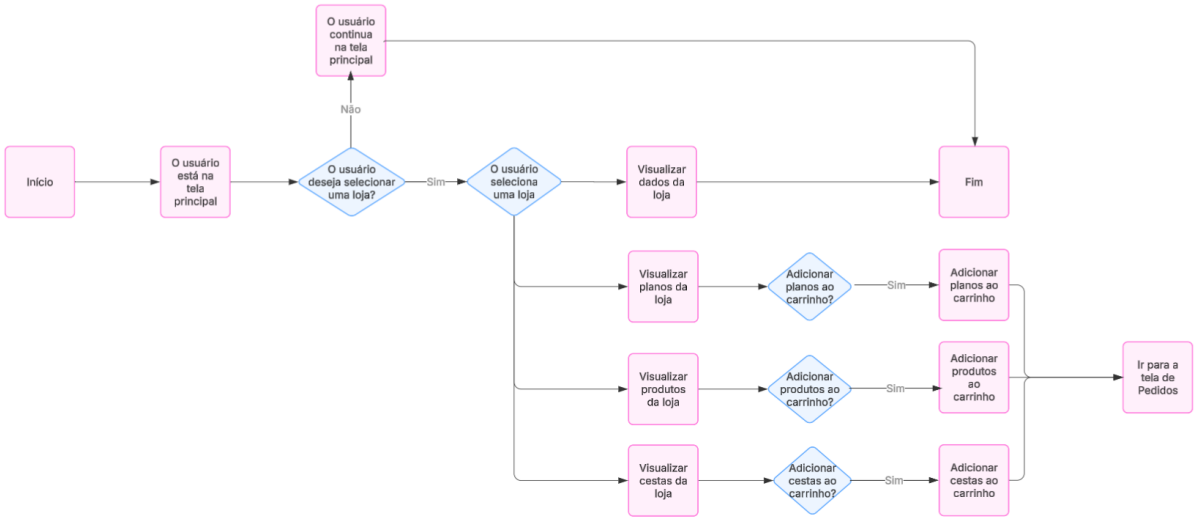


Figura 21 – Fluxo Pedidos

- **Requisição 1**: Obter lojas

  GET ...dev/lojas

- **Requisição 2**: Obter histórico de compras

  GET ...dev/extratoes?user=cfa7ee6e-7054-4ae1-891a-990ef738ab7e

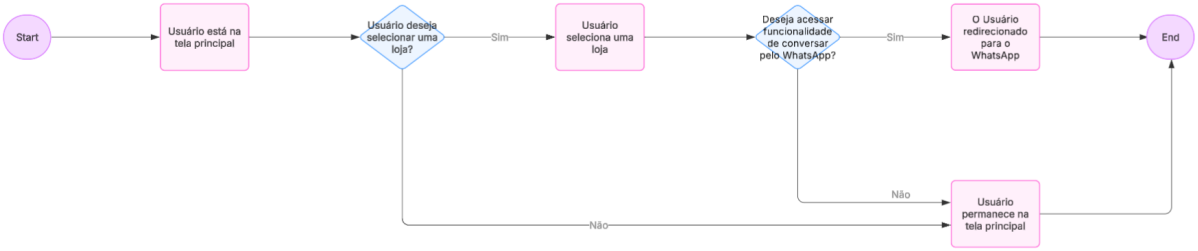


Figura 22 – Fluxo WhatsApp

- **Requisição 1**: Obter lojas

  GET ...dev/lojas

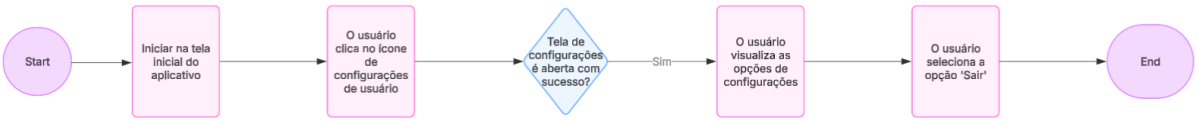


Figura 23 – Fluxo Logout

- Não possui requisições

Cada operação da aplicação, como consultas, inserções, atualizações e exclusões de dados, foi analisada para determinar sua contribuição no volume total de requisições. Com base no comportamento esperado dos usuários e nos fluxos de uso mapeados, chegou-se à estimativa de que um único usuário realiza, em média, 96 requisições mensais à API.

Com essa métrica definida, foi possível calcular a capacidade máxima de usuários suportados antes que a aplicação ultrapasse o limite gratuito da AWS Lambda. Considerando que o serviço permite até 1.000.000 de requisições gratuitas por mês, a capacidade de usuários pode ser determinada pela seguinte fórmula:

$$U_{\max} = \frac{R_{\max}}{R_u} \tag{4.1}$$

Onde:

- $U_{\max}$ representa o número máximo de usuários suportados dentro do limite gratuito;
- $R_{\max}$ é o limite máximo de requisições mensais (1.000.000);
- R_u corresponde à média de requisições por usuário (96).

Aplicando os valores:

$$U_{\max} = \frac{1.000.000}{96} \approx 10.416 \tag{4.2}$$

Ou seja, a API pode atender aproximadamente 10.416 usuários por mês em cada CSA antes de ultrapassar o limite gratuito da AWS Lambda.

4.7.2.3 Resultado

A API pode atender aproximadamente 10.416 usuários por mês em cada CSA antes de ultrapassar o limite gratuito da AWS Lambda. É um número bem distante da atual realidade do AgroMart, pois nosso levantamento revela que a maior CSA (Floresta) possui cerca de 30 usuários mensais.

Dessa forma os mantenedores das CSAs poderão trabalhar de forma tranquila sem ter que se preocupar com gastos de infraestrutura, pois as atuais condições são mais que suficientes para a aplicação funcionar e, além disso, suportam com folga um eventual pico de usuários.

Vale destacar que um monitoramento contínuo das métricas da AWS CloudWatch (disponível no site da AWS) é essencial para evitar custos inesperados e garantir a eficiência da solução. O mapeamento detalhado dos casos de uso também pode servir como base para futuras otimizações, permitindo ajustes na lógica da aplicação para minimizar o consumo desnecessário de recursos computacionais.

4.7.3 Analise de carga e dimensionamento do banco na API Principal FAAS

Esta seção tem como objetivo avaliar o consumo de armazenamento da API backend do Agromart, implementada em um ambiente FaaS, e verificar em quais circunstâncias esse consumo pode ultrapassar o limite de 25 GB estabelecido pelo Free Tier do Amazon DynamoDB.

Para isso, será determinado o tamanho médio dos registros de cada uma das principais entidades armazenadas no DynamoDB. Com base nesses valores, será realizada uma simulação considerando cinco registros por entidade, permitindo estimar o impacto no uso total de armazenamento e identificar possíveis cenários que levem à superação do limite gratuito.

4.7.3.1 Registros das entidades

Para a nossa análise iremos ter como base os seguintes registros, com seus respectivos espaço médio ocupado no Banco DynamoDB.

- **Assinante: 185 Bytes**


```
{
    "nome": "Murilo  ",
    "id": "007a0769-15a1-443f-a031-abe361cf3e57",
    "cestas_disponiveis": 5,
    "pular_cesta": false,
    "planos": [
        "99c79e10-581a-4a81-9c8d-a8d9baad2dae"
    ],
    "lojas": [
        "f3e44c5a-3b5d-4735-be96-d867340a07f8"
    ],
    "created_at": "2025-02-24T20:27:01.564Z",
    "usuario_id": "a55f63e6-142b-4d7c-917e-"
},
```


Figura 24 – Assinante

- **Cesta: 200 Bytes**

```
{
    "id": "3aaea8bd-e104-4606-acc9-ea41fe2b4c4a",
    "valor": 50,
    "quantidade": 5,
    "descricao": "Cesta básica de frutas ",
    "imagem": "base64 imagem",
    "lojas": []
}
```


Figura 25 – Cesta

- **Endereço: 280 Bytes**


```
{
    "cidade": "Endereco Murilo",
    "complemento": "Do lado da Distribuidora",
    "numero": "151",
    "bairro": "Guara",
    "id": "a068fc9c-53aa-4de7-8703-463c21247cae",
    "userId": "a55f63e6-142b-4d7c-917e-8a147c70b58b",
    "cep": "99999999",
    "rua": "14"
}
```


Figura 26 – Endereço

- **Usuário: 185 Bytes**


```
{
    "senha": "MuriloSenha",
    "created_at": "2025-02-24T20:07:57.805Z",
    "nome": "Murilo Schiler Lopes Santana",
    "id": "a55f63e6-142b-4d7c-917e-8a147c70b58b",
    "email": "muriloschiler@email.com"
}
```


Figura 27 – Usuário

- **Produtos: 180 Bytes**


```
[
    {
        "valor": 25,
        "imagem": "Base 64 imagem",
        "nome": "Farofa",
        "unidade_medida": "Gramas",
        "id": "fbca1737-91bb-46f4-84d7-4b9d00c47d0e",
        "quantidade": 10,
        "loja_id": [],
        "descricao": "Farofa"
    }
]
```


Figura 28 – Produtos

- **Planos: 250 Bytes**


```
{
    "quantidade_de_cestas": 4,
    "valor": 150,
    "imagem": "base 64 Imagem",
    "assinantes": [
        "a55f63e6-142b-4d7c-917e-8a147c70b58b"
    ],
    "nome": "Plano Trimestral",
    "id": "73c8d317-2419-4e0f-aca1-03e691f4e3c4",
    "lojas": [],
    "quantidade": 10,
    "descricao": "Descrição Plano Trimestral"
}
```


Figura 29 – Planos

- **Loja : 500 Bytes**

```
{
  "nome": "Loja Exemplo",
  "descricao": "Descrição da loja",
  "banner": "http://exemplo.com/banner.jpg",
  "tipos_de_entrega": "Entregar",
  "contato": 1234567890,
  "cnpj": 12345678000195,
  "endereco": "",
  "cestas": ["3aaea8bd-e104-4606-acc9-ea41fe2b4c4a"],
  "planos": ["99c79e10-581a-4a81-9c8d-a8d9baad2dae"],
  "assinantes": ["29725055-40c0-4686-ba91-5bf1097fa78b"],
  "produtos": ["fbca1737-91bb-46f4-84d7-4b9d00c47d0e"]
}
```


Figura 30 – Planos

4.7.3.2 Análise

Vamos começar dimensionando a quantidade total de cada umas das entidades sobre o tamanho total do DynamoDB disponível.

O armazenamento gratuito do DynamoDB é de 25 GB, equivalente a:

$$25 \times 1024^3 = 26.843.545.600 \text{ bytes} \tag{4.3}$$

Considerando as entidades armazenadas no banco, os tamanhos médios dos registros são apresentados na Tabela 1.

Entidade	**Tamanho Médio (bytes)**
Usuários	185
Cesta	200
Endereço	280
Assinantes	228
Produtos	180
Planos	250
Loja (1 cesta, 1 produto, 1 assinante, 1 plano)	500

Tabela 1 – Tamanhos médios das entidades armazenadas no DynamoDB.

A partir desses valores, podemos estimar a quantidade máxima de registros armazenáveis utilizando a equação:

$$N = \frac{26.843.545.600}{T} \tag{4.4}$$

onde N representa o número máximo de registros e T o tamanho médio de cada entidade. A Tabela 2 apresenta os valores obtidos.

Entidade	**Máximo de Registros**
Usuários	145.073.223
Cesta	134.217.728
Endereço	95.155.521
Assinantes	117.691.878
Produtos	149.130.809
Planos	107.374.182
Loja	53.687.091

Tabela 2 – Número máximo de registros armazenáveis por entidade.

- Cenário com 5 registros de cada entidade

Considerando as entidades armazenadas no banco,o tamanho total de cada tabela representando um entidade apresentado na tabela 3.

Entidade	**Tamanho Médio vezes 5 (bytes)**
Usuários	925
Cesta	1000
Endereço	1400
Assinantes	1140
Produtos	900
Planos	1250
Loja (5 cestas, 5 produtos, 5 assinantes, 5 planos)	2500
Total	9115

Tabela 3 – Tamanhos médios das entidades armazenadas no DynamoDB considerando múltiplos registros.

A partir desses valores, podemos estimar a quantidade máxima de cenários armazenáveis utilizando a equação:

$$N = \frac{26.843.545.600}{T} \tag{4.5}$$

Onde N representa o número máximo de bytes e T o tamanho total de Bytes(9115).

Ao aplicar essa equação, obtemos um total de 2.944.985 cenários idênticos. Isso significa que, considerando um cenário em que cada entidade possui cinco registros, esse é o limite estimado de armazenamento garantido para os agricultores no Agromart dentro das restrições do Free Tier do Amazon DynamoDB.

4.7.3.3 Resultados

Os resultados indicam que a api principal dificilmente irá extrapolar os limites da AWS para o "Aways free tier". Mesmo considerando a entidade "Loja", que representa um

conjunto de registros, seriam necessárias mais de 53 milhões de instâncias para atingir o limite, mostrando que a cota gratuita é suficiente e dificilmente se tornará um fator limitante.

5 Conclusão

Este trabalho investigou a viabilidade técnica e econômica da migração do backend do AgroMart de uma arquitetura baseada em Infraestrutura como Serviço (IaaS) para um modelo de computação serverless utilizando Função como Serviço (FaaS) na AWS Lambda. A proposta visou reduzir custos operacionais, otimizar o consumo de recursos computacionais e garantir maior escalabilidade ao sistema.

A análise demonstrou que a manutenção de instâncias EC2 para hospedar as APIs do AgroMart apresenta um custo significativo, com um valor estimado de R$ 171,55 mensais para um servidor t3.medium, sem considerar despesas adicionais como armazenamento, transferência de dados e licenciamento de software. Além do impacto financeiro, essa abordagem exige um gerenciamento manual da infraestrutura, o que dificulta a escalabilidade dinâmica da aplicação, podendo comprometer a experiência do usuário em períodos de alta demanda.

Em contrapartida, a adoção da arquitetura FaaS mostrou-se uma alternativa promissora. Com a execução sob demanda das funções na AWS Lambda, eliminam-se os custos associados à ociosidade dos servidores, garantindo um uso mais eficiente dos recursos computacionais. Além disso, a escalabilidade automática do modelo serverless permite que a aplicação se adapte dinamicamente ao volume de requisições, sem a necessidade de intervenções manuais na infraestrutura.

Durante o processo de migração, foi necessária a adaptação do aplicativo mobile para consumir a nova API hospedada na AWS Lambda. A refatoração envolveu mudanças nos endpoints e ajustes no formato das requisições. Essas alterações garantiram a compatibilidade do frontend com a nova estrutura backend e minimizaram impactos na experiência do usuário.

Dessa forma, os resultados obtidos indicam que a migração do AgroMart para o modelo FaaS pode reduzir significativamente os custos operacionais, ao mesmo tempo em que melhora a eficiência da alocação de recursos e a escalabilidade da aplicação. No entanto, a viabilidade total da migração depende da análise contínua dos custos à medida que a demanda pelo sistema cresce, principalmente em relação às limitações da camada gratuita da AWS e aos custos adicionais que possam surgir.

Por fim, este estudo contribui para o entendimento das vantagens e desafios da adoção da computação serverless no AgroMart, servindo como base para futuras pesquisas que busquem otimizar ainda mais a eficiência e a sustentabilidade financeira deste sistemas.

6 Desenvolvimentos Futuros

A migração do *AgroMart* para um ambiente *serverless* demonstrou ser uma alternativa viável, mas sua evolução exige a definição de novos desafios e aprimoramentos. Dentre os principais aspectos a serem explorados, destacam-se a análise de desempenho, a escalabilidade do sistema, a implementação de novas funcionalidades e o reforço das diretrizes de segurança.

Um dos desafios primordiais é a realização de um estudo detalhado sobre o desempenho da nova arquitetura. É necessário avaliar o comportamento do *AgroMart* diante de um aumento no número de usuários simultâneos, requisições e interações com o banco de dados. Além disso, torna-se fundamental estabelecer os limites operacionais do *Always Free Tier*, determinando a capacidade máxima de usuários e itens suportados dentro dessa camada gratuita. Com base nessa análise, será possível identificar até que ponto o desempenho permanece adequado e quais otimizações podem ser aplicadas sem comprometer a segurança e a estabilidade da aplicação.

Outro aspecto essencial é a definição de estratégias de escalabilidade para o *AgroMart*. Quando os limites do *Always Free Tier* forem excedidos, será necessário modificar as configurações do sistema e estabelecer um modelo de planos de uso escaláveis. Isso permitirá que os agricultores tenham diferentes possibilidades de crescimento dentro da plataforma, garantindo continuidade e previsibilidade nos custos operacionais.

Além da escalabilidade, a implementação da funcionalidade de pagamento surge como uma necessidade crítica para o *AgroMart*. Atualmente, essa funcionalidade não está disponível, e sua introdução deve ser planejada considerando não apenas a experiência do usuário, mas também o impacto que essa operação pode ter sobre os limites do plano gratuito da AWS. Dessa forma, será possível assegurar que a aplicação continue operando dentro de um modelo sustentável, sem comprometer sua viabilidade financeira.

Outro aprimoramento essencial diz respeito à autenticação e autorização dos usuários. Para garantir a conformidade com boas práticas de segurança, é necessário estabelecer diretrizes robustas que protejam tanto os agricultores quanto os consumidores. Esse refinamento contribuirá para a integridade dos dados e a confiabilidade do sistema.

Por fim, a transição para *serverless* resultou na perda do painel administrativo anteriormente fornecido pelo *Strapi*, que permitia a gestão das CSAs pelos agricultores. Assim, faz-se necessário o desenvolvimento de uma nova solução administrativa, assegurando que os produtores tenham acesso a uma interface eficiente para o gerenciamento de suas operações.

Dessa forma, os desenvolvimentos futuros do *AgroMart* deverão ser pautados na busca por uma solução mais estável, escalável e acessível, permitindo que a plataforma continue a atender às necessidades dos agricultores de maneira eficiente e sustentável.

Referências

BECK, K. *Extreme Programming Explained: Embrace Change*. [S.l.]: Addison-Wesley Professional, 2000. Citado 2 vezes nas páginas 21 e 22.

CAPPELLOZZA, O. P. S. . A. Antecedentes da adoção da computação em nuvem: Efeitos da infraestrutura, investimento e porte. 2012. Citado 2 vezes nas páginas 19 e 20.

CLOUD, O. *Preços Oracle*. 2025. Acesso em: 21 fev. 2025. Disponível em: <https://www.oracle.com/cloud/free/>. Citado na página 42.

FERREIRA IGOR FARIAS, P. L. G. C. S. Uma análise comparativa entre serviços saas (awsecs) e iaas (aws-ec2). 2023. Citado 2 vezes nas páginas 11 e 12.

FERREIRA, U. J. S. Análise de tecnologias de virtualização e hardware de baixo custo para infraestrutura de nuvem de pequeno porte. 2017. Citado na página 21.

FRANçA AUDREY TELES DOS SANTOS, I. D. C. d. J. S. M. T. W. A. G. d. A. . L. D. d. L. P. M. T. A utilizaÇÃo da computaÇÃo em nuvem como auxÍlio À escalabilidade e disponibilidade de serviÇos online. 2023. Citado na página 20.

FREITAS, A. A.-A. de; CELLA, P. V. de S. *Uma evolução do projeto Agromart: implantação individualizada e automatizada de um ambiente de CSA*. Dissertação (Trabalho de Conclusão de Curso (TCC)) — Universidade de Brasília - UnB, Faculdade UnB Gama - FGA, 2023. Citado na página 45.

GOOGLE. *Preços do Firestore*. 2025. Acesso em: 21 fev. 2025. Disponível em: <https://cloud.google.com/functions/pricing-overview?hl=pt-br>. Citado na página 42.

GOOGLE. *Preços do Firestore*. 2025. Acesso em: 21 fev. 2025. Disponível em: <https://cloud.google.com/firestore/pricing?hl=pt-br>. Citado na página 42.

NOGUEIRA, P. . T. Computação em nuvem. 2013. Citado na página 19.

PROGRAMMING, E. *Extreme Programming: A Gentle Introduction*. [S.l.], 2013. Disponível em: <http://www.extremeprogramming.org>. Acesso em: 10 set. 2023. Citado na página 21.

RIBEIRO, A. F. C.; MAGALHãES, R. L. T. d. Associação para aplicações agromart de uma csa em cloud. 2023. Citado 2 vezes nas páginas 14 e 21.

SERVICES, A. W. *AWS Lambda Pricing*. 2025. Acesso em: 21 fev. 2025. Disponível em: <https://aws.amazon.com/pt/lambda/pricing/>. Citado na página 41.

SERVICES, A. W. *AWS Lambda Pricing*. 2025. Acesso em: 21 fev. 2025. Disponível em: <https://aws.amazon.com/pt/free/faqs/>. Citado na página 41.

SILVA;CARVALHO. Análise de mecanismos de serverless computing em ambientes de nuvens computacionais. 2021. Citado 2 vezes nas páginas 11 e 14.

SOMMERVILLE, I. *Software Engineering (10th Edition)*. [S.l.]: Pearson, 2015. Citado na página 18.

SUTHERLAND, J. *Scrum: A arte de fazer o dobro do trabalho na metade do tempo*. [S.l.]: Currency, 2014. Citado na página 21.

SWANSON, E. B. *The Dimensions of Maintenance*. [S.l.: s.n.], 1976. Citado na página 18.

TELES, L. Estudo comparativo sobre métodos ágeis de desenvolvimento de software. 2017. Citado 2 vezes nas páginas 21 e 22.

WAZLAWICK, R. S. Engenharia de software - conceitos e práticas. 2013. Citado na página 18.