Relatório da A2 de Computação Escalável:

Plataforma de E-commerce

26 de Junho de 2025

Integrantes:

Daniel de Miranda Almeida

João Felipe Vilas Boas

Lívia Verly

Luís Felipe de Abreu Marciano

Paulo César Gomes Rodrigues

Visão Geral

Este trabalho propõe a transformação do pipeline de processamento de dados desenvolvido na A1 em uma arquitetura distribuída e escalável na nuvem AWS, com foco em ingestão de alto volume e processamento em tempo real. A solução implementa um sistema híbrido capaz de operar tanto em ambiente local quanto na nuvem, utilizando serviços gerenciados da Amazon Web Services (AWS). Como estudo de caso, aplicamos o pipeline a um cenário de E-commerce, processando eventos de vendas, comportamentos de usuários e dados históricos para gerar 10 métricas estratégicas. Todo o código e configurações estão disponíveis em nosso repositório GitHub.

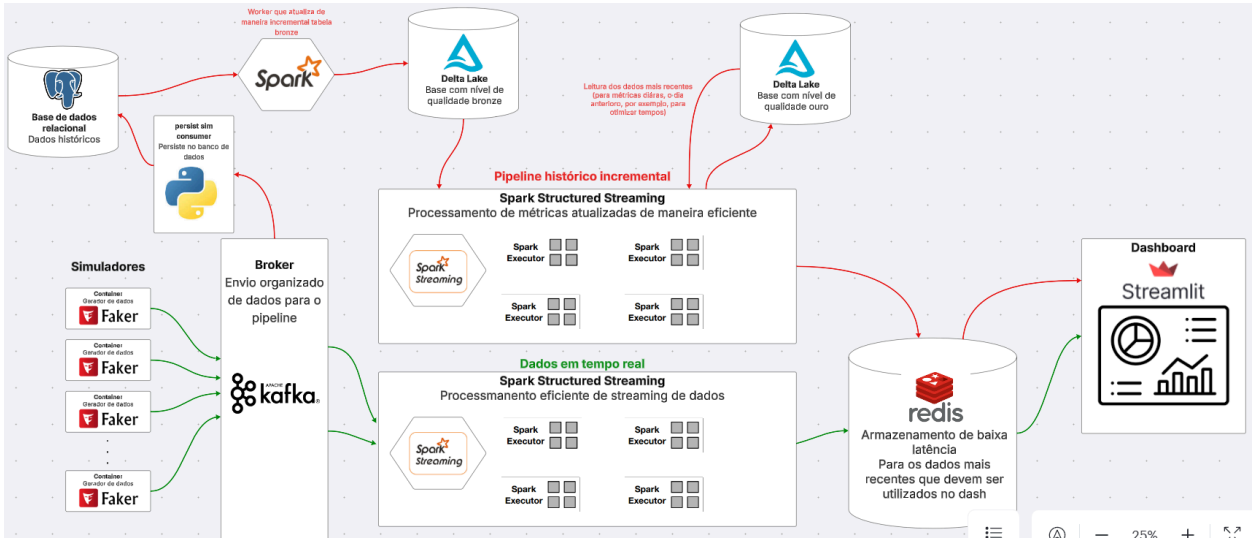
Objetivos

O principal objetivo deste trabalho é projetar e implementar um pipeline de processamento de dados escalável e distribuído, capaz de lidar com fluxos contínuos de eventos em tempo real, enquanto integra dados históricos para análises estratégicas. A proposta é desenvolver uma solução que combine eficiência no processamento paralelo com flexibilidade arquitetural, permitindo sua execução tanto em ambientes locais quanto em nuvem AWS.

Projeto Exemplo: Plataforma de E-commerce

Fizemos um pipeline de processamento de dados para um sistema de e-commerce com gestão logística. Esse cenário foi escolhido por representar desafios reais que podem envolver grande volume de informações, diversos tipos de dados e a necessidade de processamento rápido.

Modelagem



Para a simulação de dados gerados em tempo real, utiliza-se a biblioteca Python *Faker* Essa escolha se deveu à sua capacidade única de produzir, por meio de múltiplos processos locais, dados realistas com padrões variados enquanto mantém consistência nas relações entre entidades. Sua flexibilidade nos permitiu criar perfis de usuários, produtos e transações com comportamentos probabilísticos próximos da realidade, tudo isso garantindo a reprodutibilidade dos testes através de seeds fixos e respeitando requisitos de privacidade por não utilizar dados reais. A combinação do Faker com multiprocessing mostrou-se particularmente eficiente, gerando volumes massivos de dados para testar os limites do pipeline.

Uma vez gerados, os dados são enviados para o *Apache Kafka*, escolhido como o sistema de mensageria devido à sua capacidade de lidar com altos volumes de dados em tempo real enquanto garante durabilidade e processamento ordenado dos eventos. Ele atua como um *broker*, recebendo os dados dos simuladores e organizando em tópicos temáticos particionados, permitindo que múltiplos consumidores acessem os dados de forma eficiente e escalável.

O armazenamento de dados históricos é feito com o PostgreSQL, um banco de dados relacional e robusto. Dados novos são lidos periodicamente por um worker Spark dedicado e capaz de ler em paralelo com várias partições e que faz a atualização incremental da camada bronze do DeltaLake. Essa integração permite que o Spark execute transformações complexas em dados brutos antes de enviar

os dados consolidados para o Dashboard e atualizá-lo. A conexão entre essas tecnologias é gerenciada através de pools de conexão e configurações de buffer que equilibram desempenho e estabilidade.

Assim que processados, os dados são enviados diretamente para o dashboard construído com Streamlit, sem uma camada de cache. Essa decisão foi tomada por termos a natureza menos frequente das atualizações dessas métricas e pelo volume reduzido de dados processados.

Quanto aos dados em tempo real, utiliza-se do Spark Streaming (extensão do Spark voltada para fluxos contínuos de dados). Os dados provindos do Kafka são processados à medida que chegam e, então, os resultados dessas transformações são enviados para o Redis, banco de dados chave-valor em memória para armazenar os dados mais recentes . O Redis atua como um buffer de alta velocidade para o dashboard, mantendo sempre os últimos valores calculados prontos para consulta imediata, enquanto oferece recursos como expiração automática (TTL) para dados temporais e atomicidade nas operações.

Local

Bases de Dados

Para execução local, geramos uma base de dados PostgreSQL com a estrutura necessária para receber os dados históricos gerados pelos simuladores. Todos os arquivos necessários para este processo estão localizados em “src/db”, contendo scripts especializados para cada etapa do processo:

- `db_config.py`: importa configurações para acesso ao banco de dados PostgreSQL;
- `create_db.py`: cria o esquema do banco e insere nele a estrutura de tabelas;
- `carga_inicial.py`: popula o banco com dados realistas;
- `verifica_db.py`: verifica se o banco de dados está construído corretamente.

Essa estrutura não apenas suporta o desenvolvimento local, mas também serve como blueprint para a implementação em nuvem, garantindo consistência entre ambientes. Os scripts permitem execuções repetidas sem corromper dados existentes.

Processamento de dados históricos

Ao lidar com a parte que envolve dados históricos, utiliza-se uma estrutura de camadas e diversos workers, que busca otimizar a alocação de recursos para partes que necessitam mais a longo prazo, visando organizar e estruturar o processamento de dados. Ela é composta por:

- **Carga inicial:** O container *historical-load* faz a carga inicial de todos os dados atualmente presentes no Postgres para a tabela Bronze do DeltaLake
- **Carga incremental:** A partir daqui, o container *incremental-bronze-worker* assume a responsabilidade de ler periodicamente as tabelas do PostgreSQL buscando por novos dados e fazendo a carga incremental na camada Bronze do DeltaLake, de forma a otimizar o tempo de cargas novas de dados
- **Bronze:** Camada com menos qualidade dos dados históricos. É a fonte inicial dos cálculos das métricas pois oferece mais performance na manipulação de uma stream contínua de dados históricos.
- **Pipeline Incremental**: Aqui é onde a maior parte do trabalho acontece. O container *historical-streaming* é um worker rodando Spark Structured Streaming responsável por ler as novas linhas na camada bronze do DeltaLake, junto das métricas mais recentes na camada ouro e atualizá-las continuamente no Redis utilizando foreachPartition para a melhor performance, onde esses dados são lidos pelo Dashboard.
- **Ouro:** Fonte da verdade dos dados e “banco de estado”. Tem como principal função auxiliar no cálculo eficiente de métricas históricas do Spark Structured Streaming

Toda a parte histórica do pipeline está contida no diretório src/historical-pipeline. Esta pasta contém:

- `historical_load.py`: Faz a carga inicial e “organiza” a camada bronze do Deltalake;
- `incremental_bronze_worker.py`: Worker que lê periodicamente do PostgreSQL a atualiza a camada bronze do DeltaLake;
- `streaming_hist_metrics.py`: Implementa a lógica principal do pipeline histórico incremental. Lê dados novos na base bronze continuamente com Spark Structured Streaming, utilizando a base ouro para salvar as métricas mais recentes (trimestre anterior no caso de *top* produtos por trimestre, e.g.) e atualiza continuamente o Redis.

Toda a estrutura é definida com o fim de possibilitar calcular métricas de forma eficiente. As métricas escolhidas no âmbito histórico do pipeline são:

- Taxa de carrinhos abandonados: Calcula a razão dos carrinhos criados que não evoluem para uma compra efetiva. Implementada em `metric_abandoned_cart_rate.py`.
- Top 10 produtos mais vendidos por trimestre: Retorna os 10 produtos com maior número de vendas nos últimos quatro trimestres completos. Implementada em `metric_most_sold_product.py`.
- Crescimento da receita: Crescimento percentual da receita em relação ao dia anterior. Implementada em `metric_revenue_grouth.py`.

Os outros arquivos presentes na pasta são de versões anteriores de teste do pipeline e não foram apagados por falta de tempo hábil.

Decisões de projeto

As principais decisões giram em torno da escolha de um padrão em camadas, o uso do Spark Structured Streaming e a divisão do trabalho em três containers. O uso do DeltaLake com as bases a partir das quais as métricas são calculadas permite uma performance muito melhor. Além disso, como a parte mais custosa do trabalho reside no streaming histórico, dividir a carga entre três containers permite que o spark possa escalar automaticamente a etapa mais custosa conforme a necessidade, além aumentar o desacoplamento, modularização e simplicidade do código. Por fim, a escrita no Redis teve como objetivo manter um padrão único de leitura dos dados pelo Dashboard.

Potenciais problemas

Ao rodar o projeto todo com “docker compose up” pode ser que demore um longo período de tempo até que as métricas históricas comecem a ser publicadas. Isso se deve a fato de que não conseguimos aumentar a capacidade de uso de memória do Spark, o que faz com que ele precise dividir a manipulação dos dados em vários pedaços (como é possível de ver pelos logs); e pelo fato de que é feita uma carga inicial de todos os dados, que naturalmente demora um pouco mais. Em uma das nossas runs na nuvem esses problemas foram mitigados pelo escalamento automático do ERM.

Processamento de dados em tempo real

Os dados em tempo real são simulados no arquivo `producer.py`. O script começa carregando um banco de dados com informações válidas (como produtos e usuários) que servirão como base para simular eventos coerentes.

O mesmo também carrega uma função responsável por simular diferentes ações que um usuário pode realizar, como login, criação de carrinho, adição de produtos, finalização de compra, entre outros. A escolha das ações é feita de forma probabilística, conferindo realismo ao comportamento dos usuários.

Cada evento gerado (representado como um dicionário Python) é convertido para o formato JSON e enviado para tópicos específicos do Apache Kafka. Todo esse processo ocorre em um loop infinito, onde, a cada iteração, um usuário aleatório é selecionado e uma nova ação é simulada e enviada ao Kafka.

O módulo `spark_consumer.py` é responsável por consumir e processar os dados transmitidos em tempo real pelo Kafka, utilizando o Apache Spark Structured Streaming. Nele, é iniciada uma sessão Spark configurada para streaming estruturado, de forma que o sistema lê continuamente dois tópicos principais: um relacionado às transações e outro às ações dos usuários. As mensagens JSON são transformadas em DataFrames e organizadas com base em schemas previamente definidos, permitindo análises eficientes. Ao fim, várias métricas são calculadas:

- Métricas globais: receita total, número total de pedidos e ticket médio;
- Receita por categoria: total de receita gerada por categoria de produto;
- Top 5 produtos mais vendidos: produtos com maior número de vendas;
- Contagem de logins;
- Contagem de carrinhos criados;
- Contagem de carrinhos convertidos: carrinhos que tornaram-se em uma compra efetiva.

As métricas calculadas são gravadas continuamente no banco de dados Redis utilizando o método `foreacPartition()`, do spark, para aproveitarmos os nós em paralelo do spark para a escrita no redis sob chaves específicas. Em vez de criar uma nova conexão com o Redis para cada uma das linhas que chegam no stream, com esse método, o Spark cria a conexão apenas uma vez para cada partição. Isso melhora muito a escrita e permite a criação de dashboards dinâmicos e atualizados em tempo real. São utilizados checkpoints para garantir tolerância a falhas, possibilitando que o processamento continue do ponto em que parou em caso de falhas no sistema.

O arquivo `persist_sim_consumer.py` também atua como consumidor do Kafka, mas com o foco em persistir os dados gerados. Ele consome os mesmos tópicos Kafka utilizados no processamento em tempo real, de forma que as mensagens são agrupadas em lotes para inserção otimizada. Ele está configurado para, de 10 em 10 segundos, ou quando houver 100 mensagens acumuladas, enviar os dados novos para o Postgres. Por fim, utiliza a função `psycopg2.extras.execute_values` para inserir os dados no banco PostgreSQL de forma eficiente, garantindo alta performance mesmo com grandes volumes de dados.

Nuvem

A migração para a nuvem consistiu na substituição de cada componente local por serviços gerenciados e escaláveis da AWS, garantindo maior disponibilidade, segurança e facilidade de manutenção. A seguir, detalhamos os principais serviços utilizados e suas respectivas funções dentro do pipeline:

- **Virtual Private Cloud (VPC):** configuramos com sub-redes públicas e privadas, distribuídas em duas zonas de disponibilidade (AZs) para garantir alta disponibilidade e tolerância a falhas. As sub-redes públicas abrigam recursos que precisam de acesso externo, como instâncias EC2 com dashboard, enquanto as privadas hospedam componentes sensíveis, como bancos de dados e serviços internos.
- **Security Groups:** a segurança da infraestrutura foi reforçada com o uso de Security Groups, que atuam como firewalls virtuais. Cada serviço possui regras específicas de entrada e saída, limitando o tráfego apenas ao necessário. Por exemplo, o acesso do EMR ao RDS foi restrito exclusivamente à porta 5432, reforçando a proteção dos dados.
- **Relational Database Service (RDS):** Implementamos o PostgreSQL como serviço gerenciado. Foi adotado como o banco de dados relacional principal, sendo responsável pelo armazenamento estruturado de dados do e-commerce, incluindo informações transacionais e métricas históricas. Trata-se de um serviço gerenciado com suporte a alta disponibilidade, replicação e backups automáticos, reduzindo a carga operacional
- **Elastic Compute Cloud (EC2):** Hospedagem dos producers, consumidores históricos, broker kafka, persistencia dos dados dos producers e do redis. Utilizamos instâncias com niveis de processamento variados de acordo com

a função. A ideia inicial era que o broker kafka utilizasse o serviço MSK da AWS, mas não tínhamos acesso a ele. O redis também, a princípio, íamos utilizar o ElastiCache, mas não julgamos que seria necessário.

- **Simple Storage Service (S3):** foi utilizado como repositório central para scripts, artefatos de deploy, datasets históricos e arquivos de log. Armazenamos código-fonte do Spark (`spark_consumer.py`), dumps de backup do RDS e artefatos de deploy.
- **Elastic MapReduce (EMR):** nele, realizamos o processamento distribuído dos dados. Implementamos dois tipos de jobs essenciais para o pipeline: um job de streaming que consome eventos diretamente dos tópicos Kafka com o script `spark_consumer.py`, calculando e enviando métricas para o Redis em tempo real. O EMR permite escalar o nível de processamento, adicionando mais cores no cluster.

De forma geral, temos:

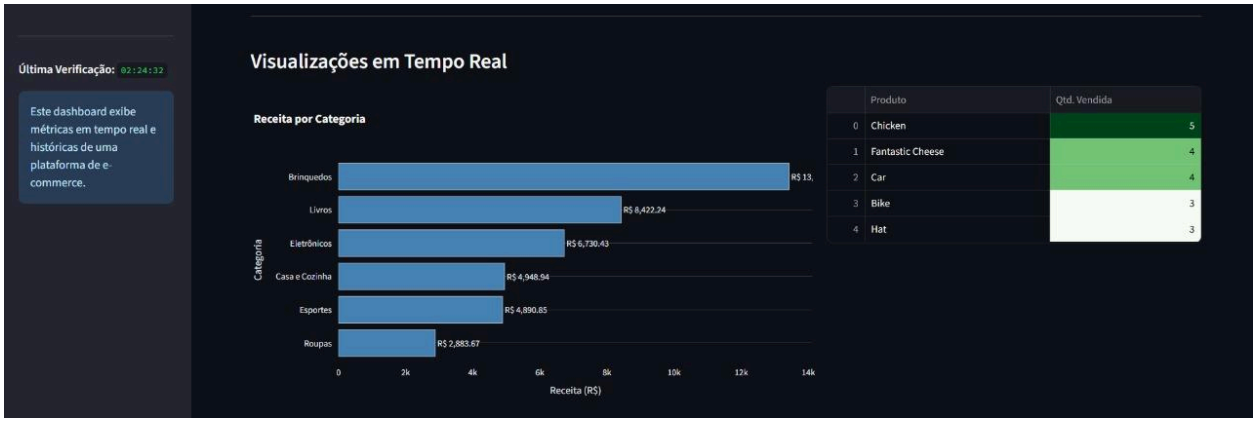
Componente local	Serviço AWS	Configuração específica
PostgreSQL (Docker)	Amazon RDS for PostgreSQL	Multi-AZ, Free Tier
Spark Consumer	EMR	1 master + 2 core nodes
producer.py	EC2	t2.micro (Free Tier)
persist_sim_consumer.py	EC2	t3.large
redis	EC2	t3.medium
Spark Histórico	EC2	t3.large
Dashboard	local	

Resultados

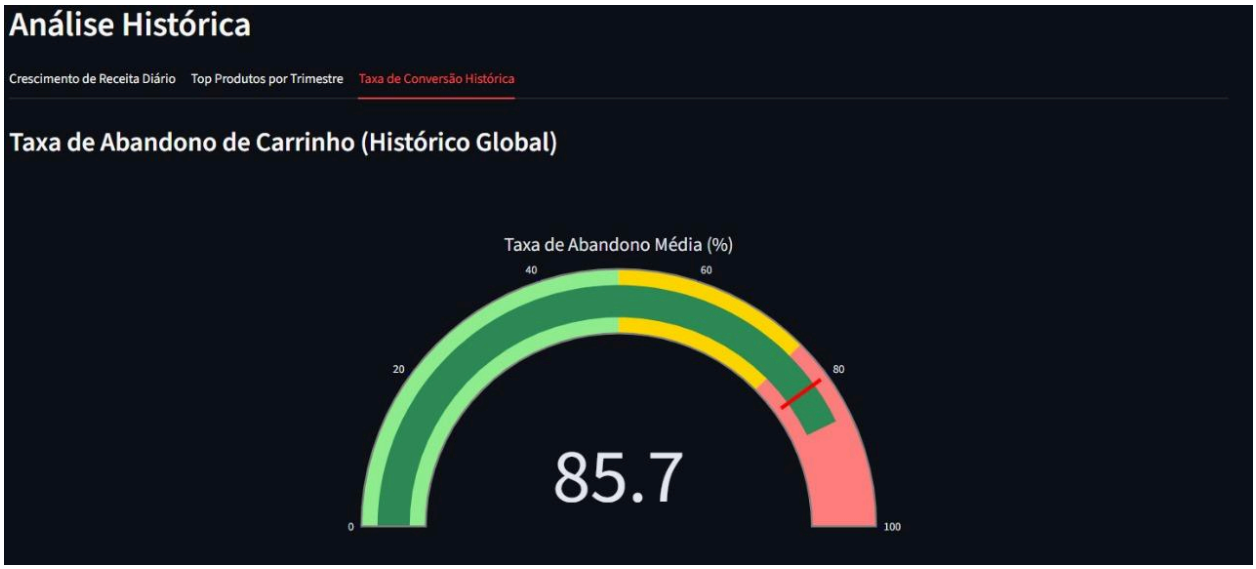
Dashboard

O dashboard de saída do nosso pipeline de exemplo ficou da seguinte forma:


Deploy
E-commerce Live View
Última Verificação: 02:24:12
Este dashboard exibe métricas em tempo real e históricas de uma plataforma de e-commerce.
Dashboard Completo de E-commerce
Métricas em Tempo Real
Receita Total
R$ 38,945.44
Pedidos Totais
81
Ticket Médio
R$ 480.81
Análise do Funil de Conversão
Etapas do Funil
135 Carrinhos Criados
32 Compras Concluídas
Taxa de Conversão
23.70%
Taxa de Abandono
76.30%


Última Verificação: 02:24:32
Este dashboard exibe métricas em tempo real e históricas de uma plataforma de e-commerce.
Visualizações em Tempo Real
Receita por Categoria
Brinquedos
Livros
Eletrônicos
Casa e Cozinha
Esportes
Roupas
R$ 8,422.24
R$ 6,730.43
R$ 4,948.94
R$ 4,890.85
R$ 2,883.67
Categoria
Receita (R$)
Produto
Qtd. Vendida
0 Chicken 5
1 Fantastic Cheese 4
2 Car 4
3 Bike 3
4 Hat 3


Análise Histórica
Crescimento de Receita Diário
Top Produtos por Trimestre
Taxa de Conversão Histórica
Evolução da Receita Diária por Segmento de Cliente
Receita Diária por Segmento de Cliente
Receita Total (R$)
Data
Segmento
novo
VIP
fiel


Análise Histórica
Crescimento de Receita Diário
Top Produtos por Trimestre
Taxa de Conversão Histórica
Taxa de Abandono de Carrinho (Histórico Global)
Taxa de Abandono Média (%)
0
20
40
60
80
100
85.7