Computação Escalável

Análise de Sistema de Reservas de Viagens

Gustavo Tironi, Kauan Mariani Ferreira, Matheus Fillype Ferreira de Carvalho,
Pedro Henrique Coterli, Sillas Rocha da Costa

25 de junho de 2025

Conteúdo

1 Introdução

Neste trabalho, propomos e implementamos um sistema de pipeline escalável para o processamento de dados de um sistema de reservas de viagens. Nosso objetivo principal é criar uma arquitetura que não apenas gerencie e transforme grandes volumes de dados de forma eficiente, mas que também seja capaz de escalar sua capacidade de processamento em resposta a diferentes cargas de trabalho, aplicando os mecanismos e padrões arquiteturais estudados na disciplina de Computação Escalável.

Atuamos como uma API de análise de dados, com o foco na criação de um fluxo contínuo que recebe dados de reservas, os processa através de uma pipeline e, a partir disso, gera análises e estatísticas relevantes para o negócio, que são exibidas em um dashboard proprietário.

Ao longo deste documento, detalhamos a modelagem da arquitetura e do banco de dados, as principais decisões de projeto, e a forma como o sistema foi projetado para permitir a avaliação de seu desempenho sob diferentes condições de carga, finalizando com a apresentação e discussão dos resultados obtidos.

2 Modelagem Geral

A arquitetura do sistema foi projetada para ser modular e distribuída, pensada para processar dados de reservas de voos e hotéis em um pipeline de dados em "quase tempo real". Uma característica relevante do nosso projeto é que a modelagem foi concebida para funcionar tanto localmente, com os serviços orquestrados via Docker Compose, quanto em ambiente de nuvem, por meio da Amazon Web Services (AWS). Isso é crucial, uma vez que a execução local é mais adequada para desenvolvimento e testes, enquanto a solução em nuvem permite a escalabilidade e um volume maior de recursos para processamento, fundamental para uma aplicação em larga escala.

Toda a solução foi construída em cima do nosso contexto escolhido, que simula um serviço de análise de dados para um sistema de reserva de viagens. Portanto, o primeiro passo foi definir as "funcionalidades"do nosso serviço de API, ou seja, quais estatísticas retornaríamos para o usuário e como faríamos isso por meio do nosso pipeline, organizando e agrupando o máximo possível para garantir eficiência. O pipeline desenhado é representado na Figura 1.

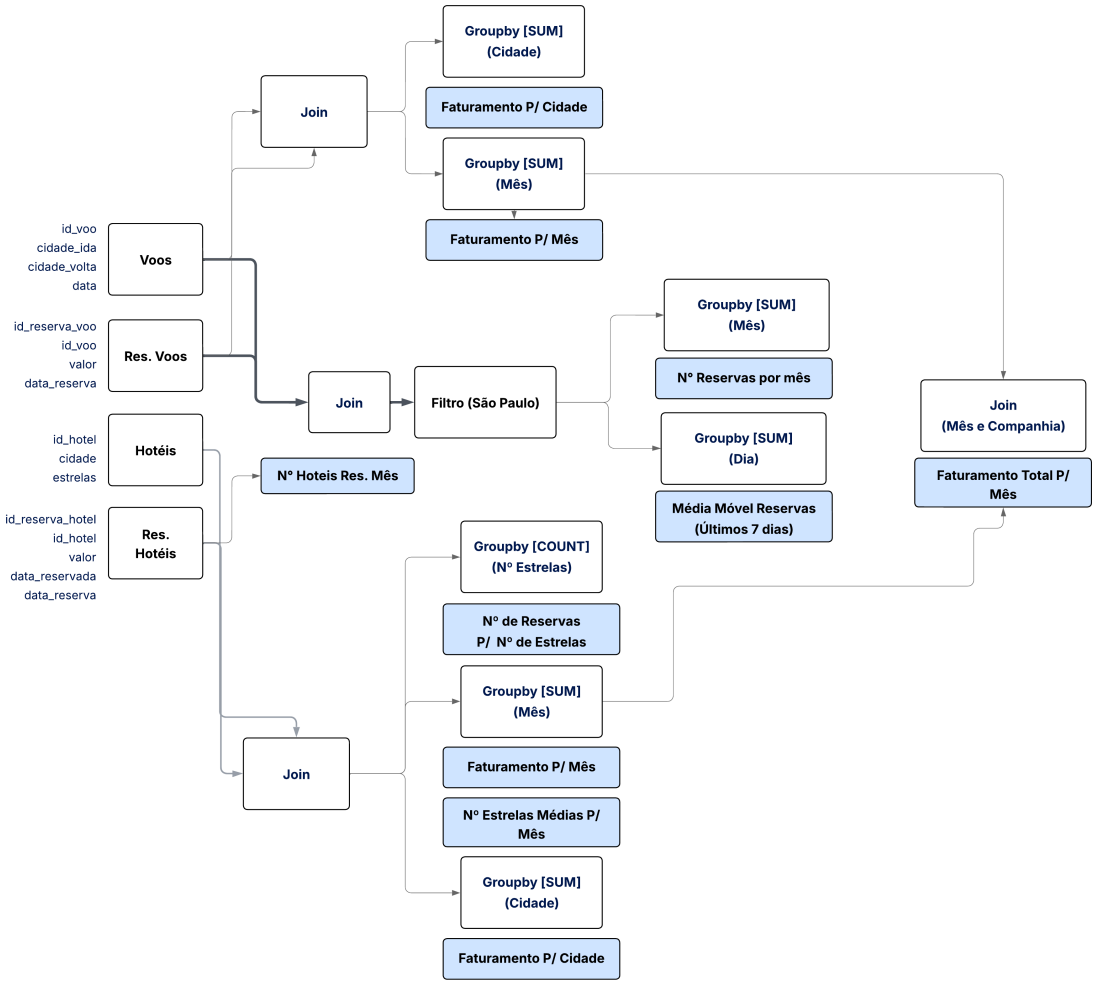

Figura 1: Pipeline Geral do Sistema

Podemos ver que todo o pipeline depende de informações de quatro tipos de dados: voos, hotéis, reservas de voos e reservas de hotéis. Eles foram modelados para funcionar de formas distintas.

As bases de "hoteis"e "voos"são consideradas bases fixas. A ideia é que elas contenham as informações de todos os hotéis e voos disponíveis, sendo comuns entre diferentes clientes e, portanto, não faz sentido cada cliente enviar esses dados repetidamente. Por conta disso, resolvemos manter essas bases como estruturas fixas e estáticas, que serão usadas de forma otimizada conforme a necessidade. Criamos um mock responsável por criar e popular essas bases de dados fixas. Este serviço executa uma única vez, garantindo que os dados sejam persistidos em volumes Docker ou diretamente em um banco de dados em nuvem, evitando a regeneração a cada execução.

Por outro lado, cada cliente pode vender o mesmo voo ou hotel a preços diferentes, e pode haver outras informações específicas de cada empresa. Por isso, na hora de modelar as bases de dados que serão enviadas pelos clientes (dados de reservas), adicionamos algumas informações extras. No geral, a ideia é que os clientes enviem apenas as reservas que

foram feitas pela sua respectiva empresa, contendo algumas informações que permitem a identificação do hotel e voo, além dos valores da transação e do identificador do cliente. Essas bases são específicas de cada cliente e são enviadas para nossa solução em batches, linha a linha ou com a base toda, funcionando de todas as formas, independentemente do formato de envio. Criamos mocks que funcionam como instâncias de um sistema de reserva que geram eventos de vendas de voos e hotéis em paralelo e os publicam imediatamente.

2.1 Modelagem da Arquitetura

A arquitetura do pipeline de processamento de dados é desenhada para alta escalabilidade e eficiência, conforme ilustrado na Figura 2.

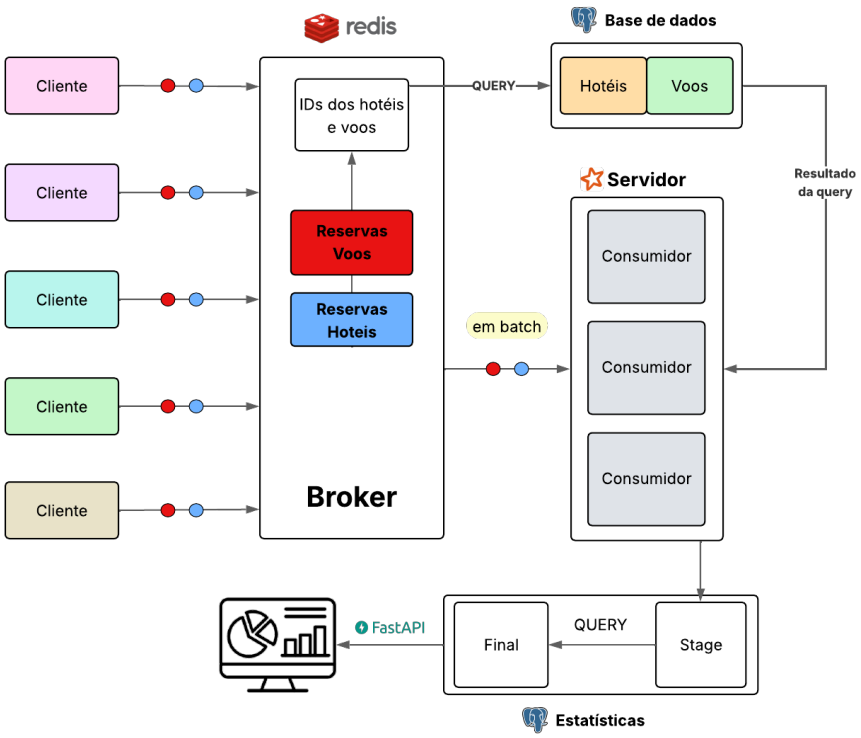

Figura 2: Visão Geral da Arquitetura

Clientes e geração de reservas: Os clientes, representados pelos mocks de geração das reservas, simulam sistemas que continuamente geram eventos de vendas de voos e hotéis. Esses eventos são publicados imediatamente e se comunicam com nossa solução por meio de um broker, utilizando pub-sub.

Pub-Sub: A comunicação inicial dos clientes com o pipeline se dá por meio de um broker Redis. O uso de um broker é altamente benéfico neste contexto, pois desacopla

o produtor do consumidor, permitindo que os geradores de dados continuem publicando eventos mesmo que o processador esteja momentaneamente indisponível ou lento. Isso garante a confiabilidade e a disponibilidade da ingestão. Além disso, o Redis, sendo um armazenamento de dados em memória, oferece alta taxa de transferência para a ingestão de eventos, característica essencial para um sistema escalável e em tempo real.

Processamento em lotes: Dentro do Redis, os dados são agrupados em lotes (batches) implicitamente nas listas. Assim que o volume de mensagens atinge um limiar, eles são enviados para processamento. Essa abordagem em batch é eficiente, pois reduz a sobrecarga de processamento individual de cada mensagem e otimiza o uso de recursos computacionais. Para otimizar ainda mais a comunicação e o processamento, o sistema, ao receber os dados de reservas, já guarda os IDs de hotéis e voos pertinentes ao batch em sets no Redis. Dessa forma, conseguimos mandar apenas as informações necessárias das bases fixas, não precisando gastar processamento e memória para dados não relevantes ao batch atual.

Processamento com spark: Uma vez que os dados de reserva, juntamente com os dados relevantes da base fixa, são enviados para o servidor de processamento, eles são tratados internamente usando Apache Spark. O Spark é a escolha ideal para esta etapa por sua capacidade de lidar com grandes volumes de dados de forma otimizada e escalável. Ele permite que as transformações e agregações sejam executadas em paralelo por múltiplos workers, garantindo que a pipeline consiga acompanhar o volume de dados gerado. Isso é essencial para as 10+ métricas e estatísticas que o pipeline deve produzir, algumas das quais utilizam dados de múltiplas fontes e dados históricos.

Pós-processamento: Ao finalizar o processamento Spark, os dados agregados (ainda em batch) são diretamente salvos em um banco de dados temporário. Como os dados em batch podem contêm informações parciais, é necessário um processo de consolidação para garantir a corretude das estatísticas históricas e agrupadas. Para isso, criamos uma função SQL que é executada periodicamente - ou quanto o usuário acessa o dashboard - e move os dados da tabela de staging para as tabelas finais de estatísticas. Este tratamento é extremamente rápido e não custoso, pois é feito diretamente no banco de dados e opera sobre dados já processados, garantindo que os dados já consolidados não sejam reprocessados, sendo excluídos da base de staging após a consolidação.

Acesso aos Dados (FastAPI e Dashboard): A comunicação entre o banco de dados final de estatísticas acontece por meio de uma API construída com FastAPI. Isso permite que o nosso dashboard (ou qualquer outra aplicação cliente) consulte as estatísticas de forma padronizada e rápida, sem acesso direto ao banco de dados, protegendo a camada de persistência e fornecendo um ponto de acesso robusto aos resultados.

Toda a arquitetura pode ser balanceada para processar mais dados de forma dinâmica,

subindo mais máquinas tanto no broker (Redis), para aumentar a capacidade de ingestão, quanto nos consumidores (servidores com Spark), para acelerar o processamento paralelo. Essa flexibilidade é a chave para a capacidade de escalabilidade do sistema.

2.2 Modelagem das bases de dados

O projeto utiliza duas instâncias de PostgreSQL distintas, otimizadas para suas respectivas funções.

a) Banco de Dados de Dados Fixa

Este banco de dados armazena as informações fixas e de referência, que não mudam com frequência. Sua finalidade é fornecer dados contextuais para o processamento das reservas.

- **Tabela** `hoteis`: Contém informações sobre os hotéis disponíveis.
  - `id_hotel`: `INTEGER` (PRIMARY KEY)
  - `cidade`: `VARCHAR(255)`
  - `estrelas`: (0-5)
- **Tabela** `voos`: Contém informações sobre os voos disponíveis.
  - `id_voo`: `INTEGER` (PRIMARY KEY)
  - `cidade_origem`: `VARCHAR(255)`
  - `cidade_destino`: `VARCHAR(255)`
  - `dia`: `INTEGER`
  - `mes`: `INTEGER`
  - `ano`: `INTEGER`

b) Banco de Dados de Estatísticas

Este banco de dados é o repositório final para as estatísticas agregadas e é o principal ponto de consulta para a API e o Dashboard.

- **Tabelas de Staging**: O Spark escreve os resultados de seus processamentos nesta camada temporária. Existe uma tabela de staging para cada uma das estatísticas. As colunas dependem da estatística.

  - Colunas comuns: `id`, `processed`.
  - Colunas específicas dependem da métrica (ex: `mes`, `ano`, `company_id`, `cidade`, `faturamento_total`, `contagem_reservas`, `estrelas`, etc.).

c) Modelagem dos Dados de Usuário (Reservas)

Apesar de não serem armazenados em um banco de dados, a modelagem dos dados de reserva enviados pelos clientes é fundamental para o entendimento do sistema.

- **Dados de Reserva de Hotéis**:
  - `id_hotel`: `INTEGER` (PRIMARY KEY)
  - `company_id`: `INTEGER`
  - `valor_pago_hotel`: `DECIMAL`
  - `data_viagem`: `DATE` (Data para a qual o hotel está reservado)
  - `data_compra`: `DATE` (Data em que a reserva foi efetuada)
- **Dados de Reserva de Voos**:
  - `id_voo`: `INTEGER` (PRIMARY KEY)
  - `company_id`: `INTEGER`
  - `valor_pago_voo`: `DECIMAL`
  - `data_compra`: `DATE` (Data em que a reserva foi efetuada)

3 Decisões de Projeto

A arquitetura do nosso sistema foi guiada por decisões que visam otimizar a escalabilidade e a velocidade no processamento de grandes volumes de informações de reservas de viagens, buscando um processamento o mais próximo possível do tempo real, considerando o contexto de negócio.

A comunicação inicial dos dados de reserva com o pipeline foi implementada utilizando um padrão de **publicação-assinatura (pub-sub)**, com o **Redis** atuando como *broker*. Essa decisão foi motivada principalmente pela necessidade de alcançar alta velocidade de ingestão e por desacoplar o produtor do consumidor. O Redis, com seu armazenamento em memória, oferece uma taxa de transferência excepcional, necessária para absorver picos de carga e organizar os dados em lotes (batches), garantindo que os geradores de

dados possam continuar publicando mesmo que o processador esteja momentaneamente sobrecarregado. Embora o Redis apresente uma menor garantia de persistência de dados em cenários de falha, para nossa aplicação, a velocidade e a eficiência na ingestão superam a necessidade de não perda de dados, uma vez que oferecemos "Processing as a Service" (PaaS) e a perda de dados, caso ocorra, apenas significaria um novo envio.

Uma vez que as mensagens são recebidas, o **Apache Spark** foi a escolha natural para o processamento de dados devido à sua robustez, capacidade de lidar com grandes volumes de dados e sua forte integração com o pub-sub. A arquitetura distribuída do Spark permite que as transformações e agregações sejam executadas em paralelo por múltiplos *workers*, o que é fundamental para acompanhar o volume de dados gerado e para produzir as diversas métricas e estatísticas requeridas pelo negócio.

Para otimizar o uso dos recursos computacionais do Spark e permitir o processamento incremental de dados muito grandes (como 1TB), optamos por um **modelo de processamento em lotes (Batch-Oriented)**. Em vez de processar cada evento individualmente ou esperar que todos os dados de reserva cheguem, o sistema aguarda que um número mínimo de mensagens se acumule nas listas do Redis antes de iniciar um ciclo de processamento. Essa abordagem em lote maximiza a eficiência para volumes maiores de dados, tanto por diluir os custos de inicialização e gerenciamento de tarefas do Spark (para mensagens pequenas), quanto por permitir o processamento gradual de grandes arquivos, tornando-o mais econômico e eficaz em um cenário de alta vazão de dados, como é o nosso contexto.

Outra decisão estratégica foi a forma como os dados fixos (hotéis e voos) são utilizados. Pensando em eficiência, os **IDs de hotéis e voos** das reservas são temporariamente armazenados em Redis Sets. Essa técnica permite que, ao iniciar um novo lote de processamento, o sistema use apenas os dados relevantes das bases fixas, evitando o carregamento desnecessário de todo o conjunto de dados. Essa abordagem otimiza o uso da memória e do processamento do Spark, garantindo que os recursos sejam dedicados apenas às informações essenciais para o *batch* atual. Além disso, construímos o *set* de IDs de forma dinâmica ao receber as mensagens dos clientes, de modo a não aumentar o tempo de processamento uma vez que o *batch* está pronto.

Para **separar as camadas de consumo**, criamos uma **API de acesso com FastAPI**, que serve como interface para acesso às estatísticas finais, enquanto o **Dashboard** (desenvolvido em Streamlit) atua como a camada de visualização. A API fornece um ponto de acesso padronizado e rápido às estatísticas processadas, protegendo a camada de persistência (PostgreSQL) e garantindo uma experiência fluida para o usuário final. Isso é relevante também para a integração com sistemas de clientes, um aspecto valioso para o negócio.

No que diz respeito à **implementação na AWS**:

- Os **geradores de dados (mocks)** foram implantados no **Amazon ECS (Elastic Container Service)**.
- O **Redis** foi provisionado como um cluster **Amazon ElastiCache for Redis**.
- As **bases de dados fixas** foram inicialmente colocadas em um **Bucket S3** para simplificar, contudo, foram migradas para um **Amazon RDS (Relational Database Service)** para maior robustez.
- O **banco de dados de estatísticas finais**, que armazena os resultados agregados do Spark, foi implementado utilizando **Amazon RDS** também.

4 Problemas e Soluções

Ao longo do desenvolvimento, enfrentamos alguns desafios que abordamos com soluções estratégicas:

Concorrência na Atualização dos Dados Finais

Um problema significativo era a concorrência de múltiplos consumidores tentando alterar os dados finais simultaneamente, o que levaria a condições de concorrência e a atrasos devido a *locks* de banco de dados. **Solução:** Para contornar isso, criamos **tabelas de *staging*** no banco de dados. O Spark escreve os resultados processados nessas tabelas temporárias. Posteriormente, uma função PL/pgSQL é executada periodicamente para consolidar os dados das tabelas de *staging* para as tabelas finais de estatísticas, somando novos valores aos existentes. Este processo desacopla a escrita massiva do Spark da leitura da API/Dashboard, minimizando o impacto no desempenho e garantindo a consistência e a atualização quase em tempo real dos dados para os usuários finais.

Processamento de Grandes Volumes de Dados de Entrada (ex, 1TB)

A necessidade de lidar com a ingestão de um volume massivo de dados sem ter que esperar o arquivo completo chegar, representou um desafio. **Solução:** Projetamos uma arquitetura que permite o **processamento incremental de dados em lotes**. Em vez de aguardar o recebimento de todo o arquivo de 1TB, o sistema processa os dados

pouco a pouco à medida que são recebidos. Os resultados parciais são agregados no final, garantindo que a ingestão e o processamento sejam contínuos e eficientes, mesmo com grandes volumes de entrada.

Otimização do Uso de Dados Fixos (Hotéis e Voos)

Evitar o carregamento desnecessário de todas as bases de dados fixas (hotéis e voos) para cada lote de processamento era crucial para otimizar o desempenho. **Solução:** Implementamos o armazenamento temporário das **IDs de hotéis e voos** reservas em Redis Sets. Ao receber as mensagens dos clientes, o Redis já guarda as informações de IDs relevantes para o lote. Dessa forma, o Spark usa apenas os dados pertinentes das bases fixas ao iniciar o processamento, eliminando a necessidade de carregar *datasets* inteiros e evitando atrasos no pipeline (por guardar assim que recebemos os dados), mantendo o processo otimizado.

Gerenciamento de Custos e Rede na AWS

Enfrentamos desafios com as redes de segurança da AWS, que eram complexas de configurar, e o rápido consumo do crédito inicial ($50) da conta. A solução para isso foi muito tempo gasto e a troca de conta constante enquanto aprendíamos a mexer. =)

5 Resultados Experimentais

Nesta seção, apresentamos os resultados do pipeline para o cliente e os resultados obtidos a partir da simulação do sistema com diferentes cargas de requisições ao servidor.

5.1 Dashboard

Começaremos como o resultado do pipeline. Para comunicar de forma clara e interativa os resultados do nosso processamento de dados ao cliente, desenvolvemos um dashboard utilizando a biblioteca Streamlit. Como mencionado anteriormente, a aplicação foi implantada em um serviço ECS com acesso externo habilitado, o que permite que os clientes consultem os resultados diretamente pela FastAPI.

A seguir, apresentamos algumas amostras das visualizações geradas pela nossa pipeline de dados, que incluem métricas sobre hotéis, voos e faturamento total:

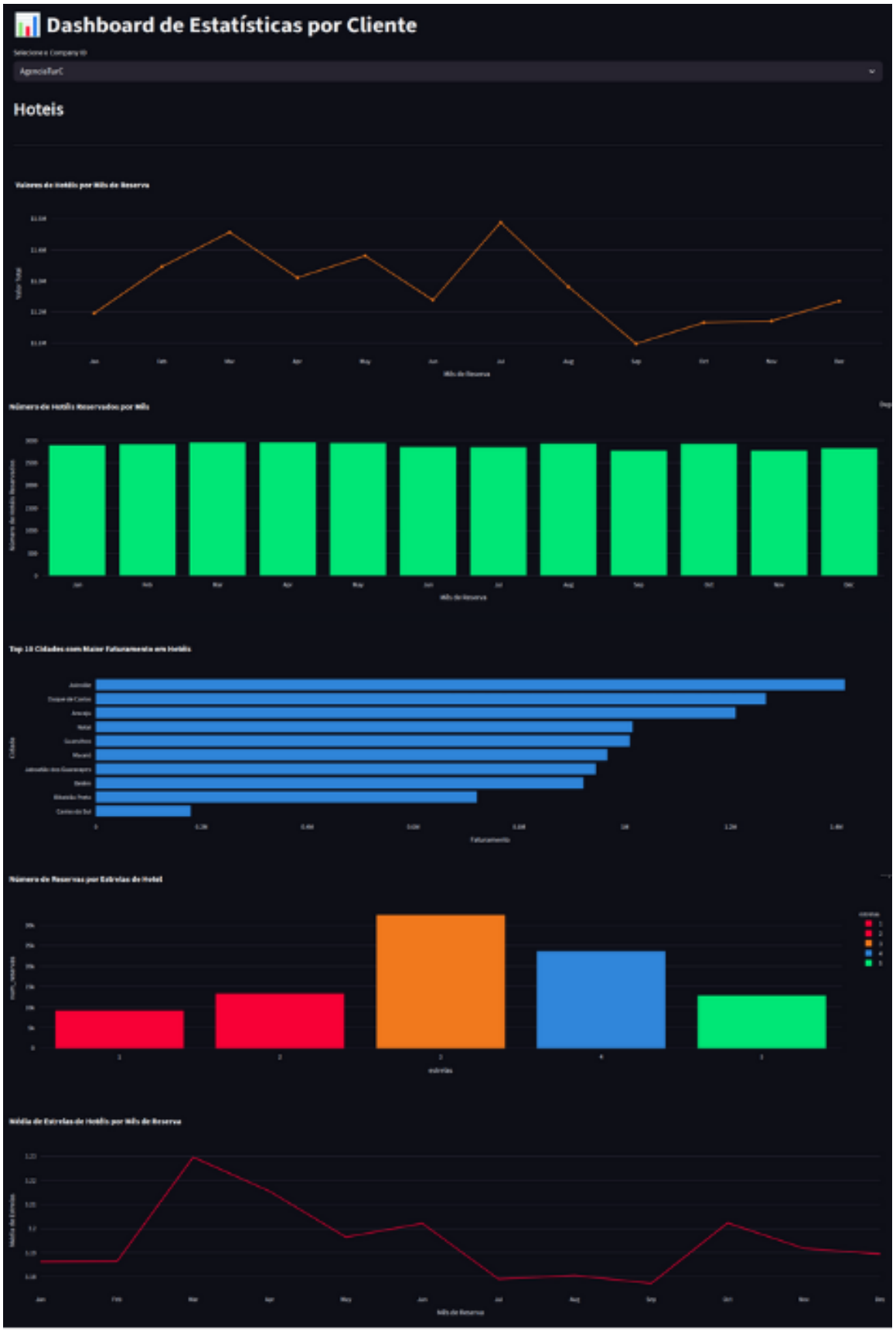

Figura 3: Estatísticas derivadas de hotéis.

A Figura 3 ilustra visualizações focadas em estatísticas de hotéis, como o faturamento total e a distribuição de reservas por número de estrelas dos hotéis.

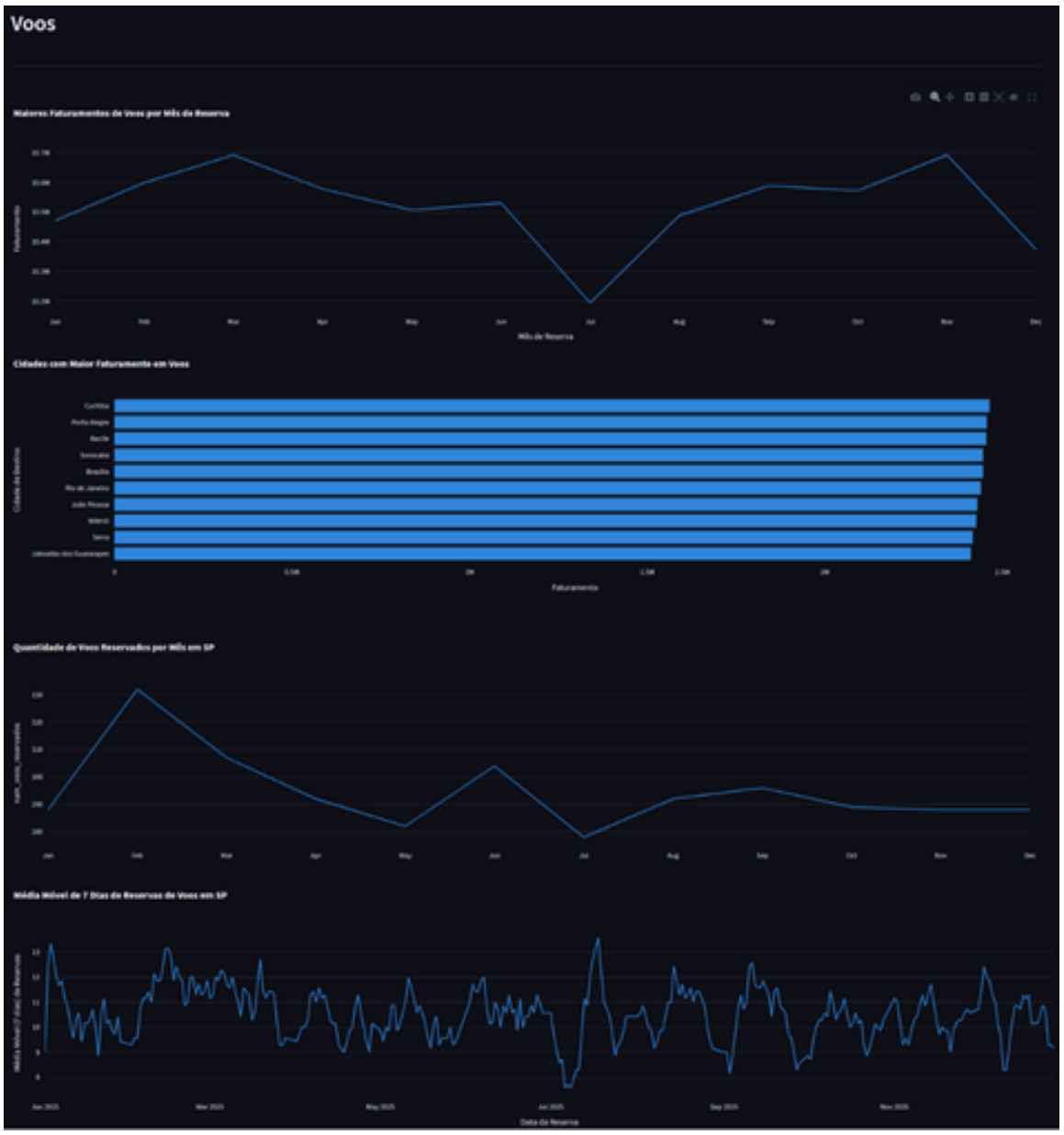

Figura 4: Estatísticas derivadas de voos.

A Figura 4 mostra estatísticas derivadas de voos, como o faturamento total por mês e um resumo da média móvel e volume de compras focado na cidade de São Paulo. Essas métricas são cruciais para entender as tendências de viagens e o desempenho das vendas.

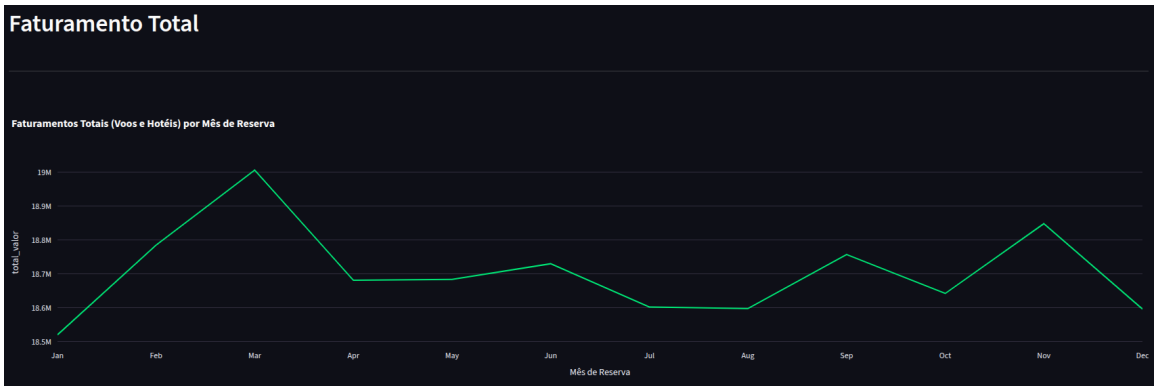

Figura 5: Estatística do faturamento total por mês.

A Figura 5 apresenta uma estatística consolidada do faturamento total por mês, combinando dados de hotéis e voos. Esta visão geral permite identificar tendências sazonais e o desempenho financeiro geral do negócio ao longo do tempo.

5.2 Teste de Carga

Como um dos objetivos desse relatório era a comparação com um trabalho anterior utilizando gRPC, é importante ressaltar que essa comparação não será 100% realista. Houveram várias mudanças significativas no nosso pipeline atual, incluindo uma modelagem de dados muito mais abrangente e a introdução de uma base de dados fixa com mais de 1 milhão de registros, o que não era uma realidade no contexto do gRPC. No trabalho anterior com gRPC, utilizávamos como métrica o número de clientes enviando dados ao mesmo tempo. Cada cliente enviava um batch de dados com um intervalo médio de 6,5 segundos, resultando em 10 envios por minuto. Cada batch continha cerca de 10 mil dados, significando que cada cliente gerava 100 mil linhas por minuto. A Figura 6 mostra um histograma dos tempos de resposta obtidos com o gRPC.

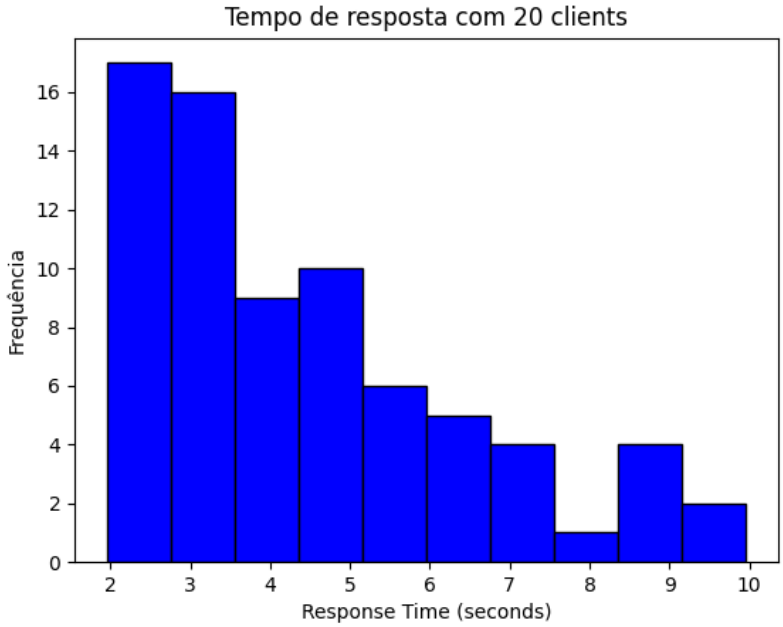

Figura 6: Histograma dos tempos de resposta com GRPC

Tendo dado as ressalvas na comparação, ao rodar o novo pipeline em um teste de carga em nuvem, os resultados obtidos estão mostrados na Figura 7.

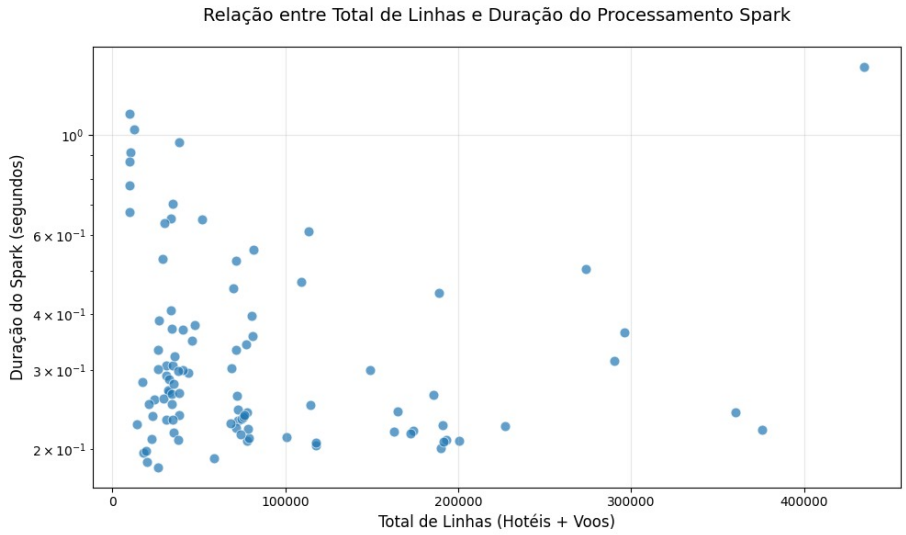

Figura 7: Resultado do teste de carga em nuvem

A comparação direta entre os dois sistemas é, de fato, complexa devido às diferenças arquiteturais e de volume de dados. No entanto, a análise da Figura 7 no contexto da nova arquitetura permite observar que conseguimos uma escalabilidade excepcional. O aumento das linhas enviadas para processamento pelos clientes não resultou em um aumento significativo no tempo de processamento. Aqui, é crucial mencionar que várias

máquinas estavam sendo usadas para processamento, o que confere uma vantagem à nova arquitetura ao distribuir tanto a geração de carga quanto o processamento. No cenário anterior, clientes e processamento compartilhavam os mesmos recursos computacionais.

Vale ressaltar também que um aumento ainda maior da carga, ou testes de longa duração, com certeza poderiam mudar esse resultado e revelar novos gargalos. Sendo necessários testes mais robustos e com mais tempo de execução para chegar a uma conclusão efetiva sobre os limites de escalabilidade. De qualquer forma, os resultados apresentados demonstram que efetivamente conseguimos chegar a uma solução robusta e escalável, capaz de lidar com volumes de dados significativamente maiores e manter a performance.

6 Conclusão

Este trabalho demonstrou a construção e implementação de um pipeline de dados escalável e robusto para análise de reservas de viagens, utilizando princípios e tecnologias de computação distribuída. Conseguimos criar um sistema que aborda desafios como a ingestão de grandes volumes de dados em "quase tempo real", o processamento paralelo e a consolidação eficiente de estatísticas históricas.

Além disso, este trabalho foi essencial para consolidar nosso entendimento sobre a importância do desacoplamento de serviços utilizando filas de mensagens (Redis), o poder do Apache Spark para processamento massivo e distribuído, as melhores práticas de modelagem de bancos de dados para cenários de alta carga (com tabelas de staging), e conteinerização com Docker para testes locais. A experiência em projetar um sistema que pode operar tanto localmente quanto em ambiente de nuvem (AWS) reforçou a compreensão das capacidades e desafios ineratórios a cada ambiente. Dando uma enfaze na AWS, notamos o quão dificl é rodar os processos em mutiplas maquians e orquestrat todas as permisões e configuraçõesp araqu

Por fim, para o futuro, vemos diversas possibilidades de evolução e aprimoramento para esta solução. Poderíamos explorar a incorporação de modelos de Machine Learning para previsão de demanda de voos e hotéis ou para a personalização de ofertas. A transição de um processamento em *batch* (mesmo que "quase em tempo real") para um verdadeiro processamento de *streams* com tecnologias como Apache Kafka Streams ou Apache Flink poderia reduzir ainda mais a latência e permitir análises em tempo real estrito. Além disso, a implementação de mecanismos de alta disponibilidade e tolerância a falhas mais avançados para todos os componentes, como replicação de banco de dados e clusters Redis Sentinel, e a exploração de arquiteturas serverless em nuvem para otimização de custos e gerenciamento, seriam passos lógicos. Finalmente, a integração com mais fontes de dados externas (como dados meteorológicos, eventos, ou indicadores econômicos) poderia

enriquecer as análises e fornecer insights ainda mais valiosos para o negócio de viagens.