

Breno Marques Azevedo
Bruno Pereira Fornaro
Luis Fernando Laguardia
Vanessa Berwanger Wille
Vinicius Hedler

Pipeline de Processamento de Dados - Simulação de Rodovias

Brasil
Junho de 2023

Breno Marques Azevedo
Bruno Pereira Fornaro
Luis Fernando Laguardia
Vanessa Berwanger Wille
Vinicius Hedler

Pipeline de Processamento de Dados - Simulação de Rodovias

Trabalho elaborado para a disciplina de Computação Escalável com o objetivo de criar um pipeline escalável que simule um monitoramento de rodovias, com geração, extração, transformação e exibição de dados.

Fundação Getulio Vargas - RJ
Escola de Matemática Aplicada

Professor: Thiago Pinheiro de Araújo

Brasil
Junho de 2023

Sumário

Sumário	i	
1	INTRODUÇÃO	1
2	MODELAGEM DO PROJETO	2
2.1	Modelagem arquitetural	2
2.2	Modelagem da base de dados	4
3	PRINCIPAIS DECISÕES DE PROJETO	5
3.1	Simulador	5
3.2	Banco de dados	5
3.3	Publisher/subscriber	5
3.4	Análises - Spark	6
3.4.1	Número de carros na simulação	6
3.4.2	Número de rodovias na simulação	6
3.4.3	Lista de carros com risco de colisão	6
3.4.4	Lista de carros acima da velocidade	6
3.4.5	Número de carros com risco de colisão	6
3.4.6	Número de carros acima da velocidade	6
3.4.7	Ranking dos top 100 veículos	7
3.4.8	Estatísticas das rodovias	7
3.5	Carros proibidos de circular	7
3.5.1	Análise alternativa	7
4	EXECUÇÃO LOCAL E EM NUVEM	8
4.1	Execução local	8
4.2	Execução em nuvem	8
5	CONCLUSÕES FINAIS	10

1 Introdução

Dando continuidade ao projeto iniciado na Avaliação 1, este trabalho consiste na implementação de um pipeline de processamento de dados provenientes de um sistema de monitoramento de veículos. Para lidar com volumes massivos de dados gerados por um número crescente de instâncias de monitoramento, passamos a utilizar serviços de nuvem. Nesse sentido, realizamos adaptações no *Simulador*, introduzimos técnicas de comunicação entre máquinas, o que não era feito anteriormente, e empregamos computação em nuvem para aumentar o número de instâncias simuladas. Como resultado, as análises derivadas desse processamento serão exibidas em um *Dashboard*, assim como nas etapas anteriores.

Dentro desse contexto, a fim de processar essa grande quantidade de dados e gerar as análises de forma eficiente, é crucial considerar o uso de técnicas de paralelismo e concorrência. Assim, iremos aplicar os cuidados e as práticas adquiridos ao longo do curso, adaptando-os para o ambiente de computação em nuvem. Nosso objetivo será identificar e resolver problemas de concorrência e paralelismo, fazendo uso dos mecanismos apresentados em sala de aula, de maneira a otimizar essas situações de forma eficiente e eficaz.

2 Modelagem do Projeto

2.1 Modelagem arquitetural

Com base nas especificações do projeto, foi dado um foco inicial ao estudo dos requisitos, técnicas e ferramentas disponíveis. A partir desse estudo, foi elaborado um planejamento para a implementação do projeto. É importante ressaltar que toda essa modelagem foi realizada antes de colocar as ideias em prática, portanto, estava sujeita a possíveis mudanças ao longo do desenvolvimento.

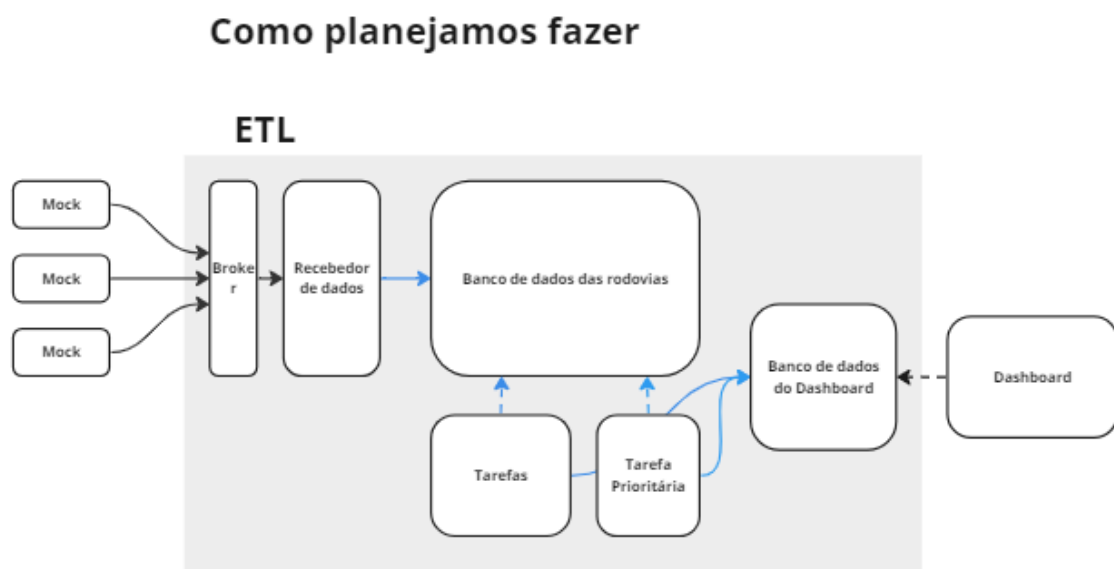


Figura 1 – Modelagem inicial

Nesse contexto, a figura acima mostra a nossa modelagem inicial para o desenvolvimento do projeto. Ela foi apresentada para o professor em sala de aula.

Resumidamente, pensamos em definir um *Broker*, usando o *Redis* e suas ferramentas, como intermediador na comunicação entre as múltiplas instâncias de simuladores e o nosso receptor de dados. Com isso, a intenção era injetar os dados em dois bancos distintos, um que armazenaria dados pontuais e outro dados históricos. Em seguida, esses dados seriam processados de forma simultânea no processamento e salvos em um banco de dados dedicado ao *Dashboard*.

Várias dessas características pareceram necessárias dadas a natureza do trabalho e das análises que teríamos que enviar. Como por exemplo, a abordagem híbrida nos processos de ETL e o uso de dois bancos de dados para separar as transformações. Contudo, ocorreram várias mudanças a medida que o projeto começou a tomar suas formas iniciais e nós fomos ficando mais familiares com as ferramentas que deviam ser utilizadas.

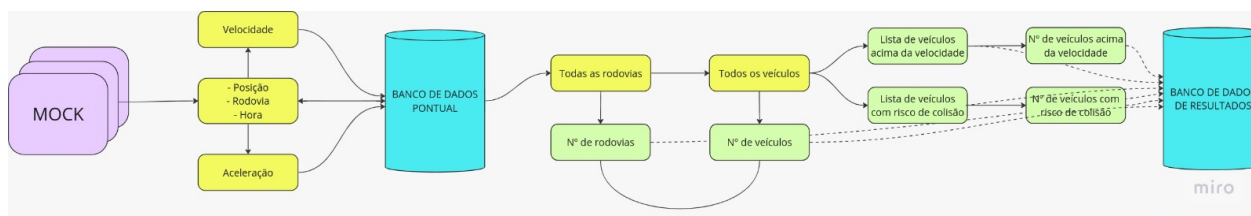


Figura 2 – Modelagem das Análises Pontuais

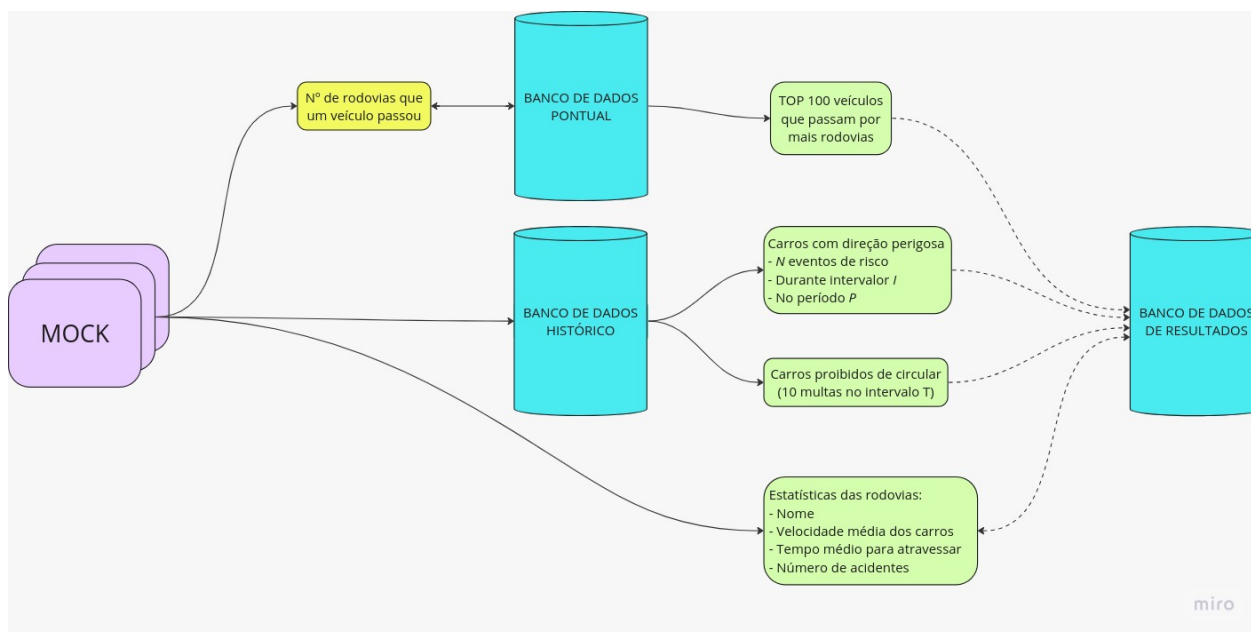


Figura 3 – Modelagem das Análises Históricas

Entretanto, com o decorrer do projeto, percebemos que essa abordagem não somente não nos traria as vantagens desejadas, como também ocasionaria em algumas desvantagens. Uma delas, talvez a principal, era que o pré processamento dos dados poderia criar um problema na fila de mensagens: caso a fila do *broker* se enchesse, poderíamos vir a perder mensagens (ou seja, perder dados). Isso não era viável, até porque um dos requisitos do projeto era de fazer análises históricas que usariam todos os dados (não podendo haver perdas).

Tendo isso em vista, resolvemos fazer uma sequência de tarefas mais simples; criamos o dados no *mock*, ele passa pelo pub/sub com o *broker*, sendo publicado, depois ele é recebido por um *subscriber* que por sua vez também é responsável por colocar os dados no banco (que é um banco de dados histórico, com todos os dados, sem pré processamento). Após isso, os dados são recuperados pelo Spark, que irá ser responsável pelo ETL e posteriormente esses dados são enviados novamente para o banco de dados em Redis, mas dessa vez um banco separado para os dados do *dashboard*, já transformados. Esses dados, por fim, são recuperados pela implementação do dashboard (separada, feita com a biblioteca Dash, no Python) e são exibidos.

Como fizemos na prática

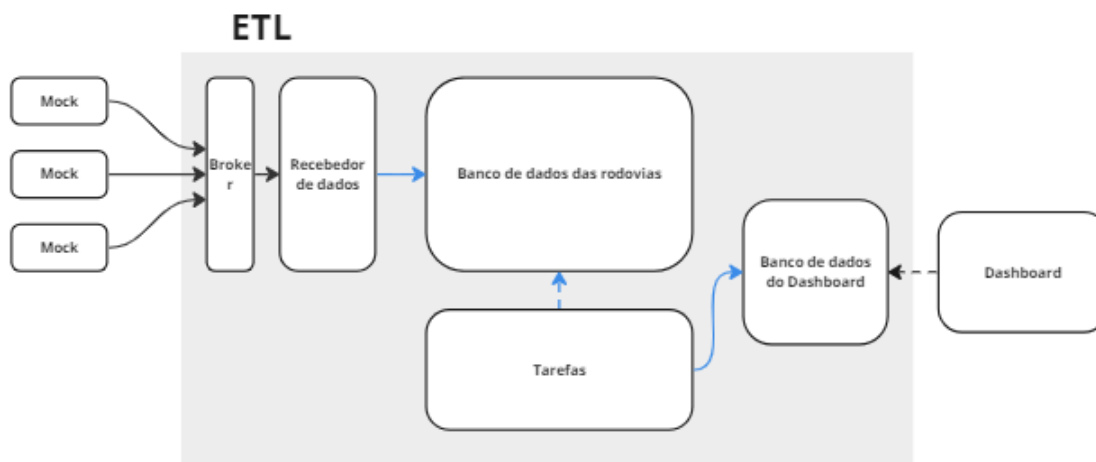


Figura 4 – Modelagem implementada

2.2 Modelagem da base de dados

Tendo em vista que o banco de dados principal, com os dados históricos de todas as "leitura" dos carros enviadas, o banco de dados consiste basicamente em uma abstração de um banco relacional, onde temos uma chave que é formada pelo horário (em *Unix epoch*, em milionésimo de segundo) e a placa do carro (estamos considerando que não está sendo enviado mais de um dado para o mesmo carro no mesmo milionésimo de segundo e, caso seja, é sobrescrito, pois essa é a menor variação de tempo que conseguimos identificar, logo não faz sentido notar nenhuma variação no carro se o tempo não mudar. Com essa chave, salvamos os valores, que são a posição do carro (qual pista ele está e quão distante da rodovia ele está), a rodovia atual e momento que o dado foi criado.

Além disso, para os dados do *dashboard*, estamos utilizando outro banco de dados Redis (que é criado ao mesmo tempo no servidor Redis, não sendo um problema de complexidade ter mais de um banco, pois é apenas outra abstração e separação do banco de dados NoSQL). Neste banco, salvamos uma chave para cada análise de requisito e como valor salvamos o resultado da *query Spark* realizada de forma que: quando o resultado é um número, ele é salvo dessa forma mesmo, mas quando é uma tabela é salvo no valor em formato de um CSV.

Ambos os banco foram projetados para serem simples de serem implementados e também interpretados posteriormente, sendo fácil de dar manutenção e também razoável de fazer com as ferramentas utilizadas nas conexões, sem muito detrimento à velocidade de processamento.

3 Principais decisões de projeto

Ao longo do projeto algumas decisões foram tomadas e, dentre essas, algumas delas precisaram ser alteradas conforme percebemos a viabilidade e inviabilidade dos planejamentos.

3.1 Simulador

No simulador, era necessário permitir que vários carros se movessem de maneira independente. Além disso, decidimos modificar a forma como as rodovias eram executadas. Anteriormente, usávamos vários comandos para executar mais de uma rodovia, mas agora optamos por executar várias rodovias em um único script Python usando a biblioteca multiprocessing. No entanto, para lidar com os carros, que apenas precisavam de um certo indeterminismo na posição e não necessariamente de verdadeira execução paralela, utilizamos threads.

No entanto, surgiram problemas de comunicação com o banco Redis devido ao uso de multiprocessamento e multithreading. Precisávamos enviar a posição de cada carro imediatamente após sua mudança, e para isso estávamos criando uma conexão com o banco Redis para cada carro. No entanto, isso rapidamente excedia o limite de conexões permitidas pelo banco. Para resolver esse problema, tentamos criar a conexão na rodovia, mas isso apresentou um novo obstáculo.

No Windows, o spawn é usado para criar novos processos em vez do fork, o que significa que era necessário fazer pickle nos objetos a serem compartilhados entre processos, mas não era possível fazer pickle nas classes do Redis. A solução encontrada foi criar a conexão dentro da função "cycle", que é a função target do multiprocessing. Dessa forma, no momento da criação dos novos processos, a conexão Redis ainda não existe e, portanto, não precisa ser picklada. Essa conexão criada dentro de cada rodovia é acessada pelos carros quando eles enviam suas mensagens, logo após se movimentarem.

3.2 Banco de dados

Como já foi dito anteriormente, começamos com um planejamento diferente dos bancos de dados, onde havia a intenção de serem criados três bancos distintos, mas ao longo do projeto notou-se que usar apenas dois seria suficiente e proporcionaria mais simplicidade no processo do ETL (considerando o restante da modelagem das *queries* envolvida).

3.3 *Publisher/subscriber*

Desde o começo, a ideia foi de utilizar o Redis como *broker* para o pub/sub, tendo em vista que já utilizamos ele como banco de dados e a ferramenta de pub/sub dele é boa e fácil de ser utilizada.

Entretanto, no começo, não estávamos entendendo muito bem como as funções do Redis funcionavam e estávamos "pulando" a etapa do pub/sub, fazendo *upload* direto dos dados no banco (de uma máquina para outra, numa mesma rede). Isso aconteceu pois o Redis tem funções que fazem essa conexão direta sem precisar estar na mesma máquina (não sabíamos que essa ferramenta era oferecida) e também é possível "escutar" esse envio de dados, como uma função de *subscriber*.

Embora dessa forma fosse mais simples de implementar, nós mudamos a implementação assim que percebemos que isso estava acontecendo. Não somente pelos requisitos do projeto, mas também tendo em mente uma "situação

real" na qual queremos simular (dentro do possível) é mais razoável que utilizemos o pub/sub de fato, pela maior flexibilidade que ele nos oferece para a transferência de dados descentralizada, pelo *broker* que irá armazenar as mensagens ainda não recebidas de forma que não haja perda de dados neste processo de comunicação e também para que não sejam utilizadas ferramentas de tão alto nível que comecem a se tornar "caixas pretas" (partes que não conhecemos o funcionamento por trás) no projeto.

3.4 Análises - Spark

Para realizar as análises de requisito do projeto, utilizamos o Spark, recebendo dados do banco de dados Redis e enviando para outro banco de dados Redis ao final das análises.

No geral, buscamos utilizar as funções oferecidas pelo Spark, principalmente a função de `lag()`, que nos permite comparar os dados de uma linha com os dados da linha abaixo. Dito isso, vamos explicar brevemente como nossas análises estão funcionando.

3.4.1 Número de carros na simulação

Nós pegamos a contagem de valores distintos de placa de carro (excluindo nulos).

3.4.2 Número de rodovias na simulação

Nós pegamos a contagem de valores distintos de nome de rodovia (excluindo nulos).

3.4.3 Lista de carros com risco de colisão

Ordenamos os carros por, rodovia, placa do carro e distância que o carro está na rodovia.

Vemos a velocidade e posição do carro anterior.

Calculamos se o carro está em risco de colisão vendo se ocorrerá uma colisão caso um carro permaneça no lugar e o outro siga em sua direção na velocidade atual por 2 segundos. Vamos deixar essa constante como parâmetro para ser definido no projeto, pois como estamos modelando com milionésimos de segundo pode não fazer sentido menor o risco de colisão em segundos como indica a Polícia Rodoviária Federal.

3.4.4 Lista de carros acima da velocidade

Faz *inner join* dos dados dos carros e das rodovias (que são fixos, criados pelo nosso world creator). Não conseguimos evitar esse *join*, por conta da natureza da consulta.

Com isso, verificamos se a velocidade atual do carro é maior do que a velocidade da rodovia, filtrando apenas os que estiverem acima da velocidade.

Por fim, reunimos duplicatas, retornando apenas um cado para cada par de placa de carro e rodovia.

3.4.5 Número de carros com risco de colisão

Contamos o número de linhas da lista de carros com risco de colisão.

3.4.6 Número de carros acima da velocidade

Contamos o número de linhas da lista de carros acima da velocidade.

3.4.7 Ranking dos top 100 veículos

Para esta análise bastou selecionar os pares distintos de placa de carro e nome de rua.

3.4.8 Estatísticas das rodovias

Para realizar a análise estatística das rodovias, coletamos informações como o nome da rodovia, velocidade média dos carros, número de acidentes e tempo médio de travessia. Combinamos os dados dos carros com os dados das rodovias em uma única base de dados. Em seguida, realizamos um agrupamento por nome da rua e instante de tempo, e calculamos algumas métricas agregadas.

As métricas calculadas foram: a média da velocidade dos carros presentes, o número de carros com velocidade zero (indicando o número de acidentes) e o tempo médio de travessia da rodovia, calculado como o comprimento da rua dividido pela média da velocidade dos carros naquela rua.

3.5 Carros proibidos de circular

Primeiramente, filtramos os dados dos carros para pegar apenas os dados no período de interesse e apenas as colunas de interesse.

Fazemos *join* dessa tabela com a tabela de dados das rodovias (world creator).

Calculamos os carros acima da velocidade (verificando a velocidade do carro e o limite da pista). Atribuímos, em uma nova coluna, 1 se o carro estiver acima do limite de velocidade e 0 caso não esteja.

Reordenamos os dados e utilizamos `lag()` para ver se o carro está sequencialmente acima da velocidade, subtraindo da verificação atual de se está acima da velocidade com a anterior. Dessa forma, queremos filtrar apenas os dados que resultem 1.

Com isso, podemos agrupar por carro ("car_plate") e contar, somando, para cada carro, quantas vezes ele excedeu a velocidade.

Por fim, pegamos apenas os carros que tiverem a contagem maior ou igual a 10.

3.5.1 Análise alternativa

Para atender ao requisito alternativo, consideramos várias opções e optamos por realizar a análise histórica (lista de carros com direção perigosa). Para isso, desenvolvemos uma lógica baseada em três condições: velocidade do carro acima do limite seguro máximo, aceleração do carro acima do limite seguro máximo e o carro mudando de faixa em dois instantes consecutivos.

Para cada carro e instante de tempo, calculamos uma coluna de contador de riscos, que representa o número de eventos de risco que o carro teve no último intervalo de tempo "i" (um parâmetro definido). Em seguida, utilizamos essa coluna para criar uma coluna de risco no intervalo, que indica o total de eventos de risco ocorridos para o carro dentro desse intervalo.

Por fim, verificamos se, no período de tempo "t" anterior ao período atual, o carro teve algum total de riscos que tenha ultrapassado o limite tolerável.

4 Execução local e em nuvem

4.1 Execução local

Para fazer todos os testes iniciais e primeira etapas do projeto, procuramos fazê-lo ser executado localmente.

Neste ponto, já com o pub/sub do Redis, obtivemos comunicação entre mais de um computador. Quando conseguimos fazer a comunicação entre todos os processos que poderiam ser separados, começamos a fazer testes mais robustos e, nessa etapa, ficou computacionalmente muito custoso fazer localmente nas máquinas dos integrantes do grupo.

Assim, para que fosse possível concluir essa etapa de execução local, foi necessário incluir o uso de docker, para usarmos contêiner e acessar o Redis, sem necessitar, portanto, de uma máquina com Linux.

Com esse cenário, obtivemos um sistema onde podem ser executadas instâncias do simulador (`simulator.py`) que atuam como publisher, enviando os dados do mock para o broker, onde, no caso, usamos o Redis. Já o script `"python_to_db.py"` trabalha como subscriber, ouvindo as mensagens e adicionando as informações em um banco Redis.

Esses dados passam então para etapa de transformação, executada pelo script `"db_to_spark.py"`, onde são realizadas todas as análises requeridas com o uso de Pyspark (comentadas a seguir).

Essas análises são novamente armazenadas no banco Redis e, por fim, usadas para o nosso Dashboard. Os dados são recuperados por meio do script `"dashboard_data_collector.py"` e, então, exibidos por meio do script `"dashboard.py"`, onde usamos a biblioteca Plotly Dash para montar a apresentação das análises. Essa biblioteca que permite que os dados sejam constantemente atualizados por meio de funções callbacks.

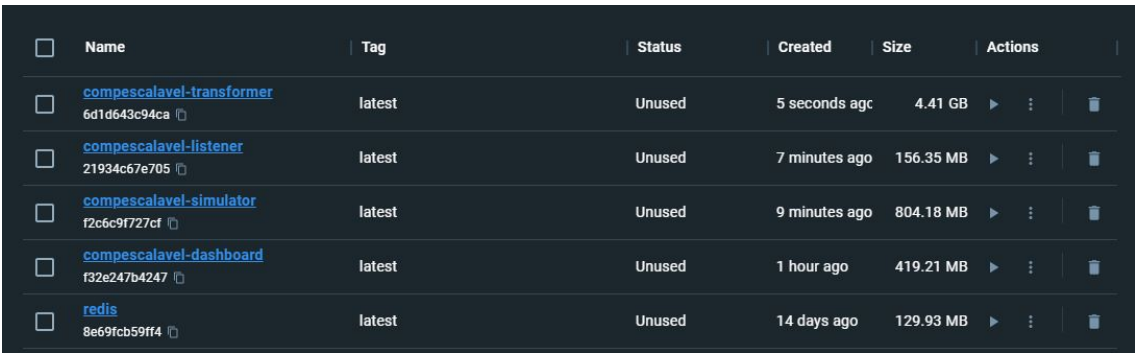
4.2 Execução em nuvem

Para realizar a execução em nuvem, idealizou-se o uso do Amazon EC2 (Elastic Compute Cloud). Essa escolha baseou-se no fato de podermos colocar nossas aplicações em contêineres Docker e então rodá-los em instâncias do EC2.

Assim, construímos 4 imagens de contêineres a partir dos nossos scripts:

- 1 para o dashboard;
- 1 para o simulador;
- 2 para o ETL.

Com isso, ficamos com as seguintes imagens necessárias para rodar o programa em nuvem ao final:



<input type="checkbox"/>	Name	Tag	Status	Created	Size	Actions
<input type="checkbox"/>	compescalavel-transformer 6d1d643c94ca	latest	Unused	5 seconds ago	4.41 GB	▶ ⋮ 🗑
<input type="checkbox"/>	compescalavel-listener 21934c67e705	latest	Unused	7 minutes ago	156.35 MB	▶ ⋮ 🗑
<input type="checkbox"/>	compescalavel-simulator f2c6c9f727cf	latest	Unused	9 minutes ago	804.18 MB	▶ ⋮ 🗑
<input type="checkbox"/>	compescalavel-dashboard f32e247b4247	latest	Unused	1 hour ago	419.21 MB	▶ ⋮ 🗑
<input type="checkbox"/>	redis 8e69fcb59ff4	latest	Unused	14 days ago	129.93 MB	▶ ⋮ 🗑

Figura 5 – As 5 imagens que geram os contêineres

Assim, conseguimos encapsular as nossas aplicações em contêineres e implantá-las em um ambiente de computação em nuvem.

É válido observar que nos deparamos com desafios relacionados ao gerenciamento de credenciais e permissões em outros serviços da AWS, como o ECS. Em particular, enfrentamos restrições que nos impediram de criar um Cluster e agrupar tarefas e configurações comuns nesse serviço específico. Inicialmente, tínhamos planejado utilizar o serviço em questão para configurar as diferentes instâncias de simuladores, permitindo ajustar os parâmetros dos cenários na simulação e controlar a quantidade de instâncias envolvidas no pipeline. No entanto, as limitações de permissões nos impuseram a impossibilidade de prosseguir com o pipeline nesse sentido.

5 Conclusões Finais

Com este projeto tivemos a oportunidade de desenvolver diversas competências no que diz respeito a computação escalável. Desde a parte de geração de dados, onde utilizamos *multithreading* e *multiprocessing* para gerar os dados a partir de várias instâncias de carros, até o pub/sub e principalmente o Spark, com o processamento paralelo de grandes volumes de dados. Embora não tenhamos conseguido cumprir com todas as etapas deste trabalho, por algumas sequência de adversidades, ainda assim conseguimos desenvolver mais nossa prática e conhecimento com o que diz respeito ao paralelismo, a Spark, a *docker*, a todas as ferramentas e serviços utilizados e até mesmo com os serviços de nuvem da AWS.

Mesmo não estando tudo completamente funcional, fizemos testes intensivos utilizando o sistema da AWS para processar e testar as *queries* desenvolvidas no Spark e também chegamos a portar o trabalho para *dockers*. O que é um indicador de que havia uma grande proximidade em deixar o sistema totalmente funcional em nuvem.

Em suma, acreditamos que conseguimos desenvolver as competências esperadas para esse projeto, conseguindo obter alguns bons resultados onde pudemos avançar mais.