**Documentação Projeto Sauter University 2025**

**Resumo Geral**

O projeto Sauter University 2025 tem como objetivo criar uma API que carrega os dados de reservatório da ONS para um Bucket do GCP, transformar os dados e deixar prontos para análise e consumo de um chatbot feito via ADK, que consegue responder sobre os dados e sobre questionamentos sobre a Sauter. E a criação de um dashboard que exiba insights sobre os dados de reservatório, tudo isso com uma infraestrutura completa levantada via Terraform. Ele integra dados da ONS, realiza pré-processamento,e transformação, e deixa os dados prontos para consumo tanto do chatbot quanto do dashboard.

**Casos de Uso**

| **Caso de uso** | **Descrição** | **Prioridade** |
| --- | --- | --- |
| **CU01** | **Conversar com chatbot para informações dos dados de reservatório.** | **Alta** |
| **CU02** | **Conversar com chatbot para informações sobre a Sauter.** | **Alta** |
| **CU03** | **Ser capaz de visualizar os dados transformados vindos do BigQuery.** | **Alta** |
| **CU04** | **Ser capaz de exportar os dados da ONS para um bucket no GCP** | **Alta** |
| **CU05** | **Visualizar dashboard com insights valiosos sobre os dados de reservatório** | **Alta** |

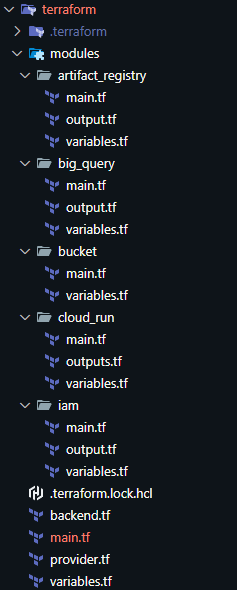
**Responsabilidades**

| **David Wesley Silva de Lucena** | **Criação de toda a infraestrutura do projeto no GCP utilizando terraform, criação da pipeline CI/CD do terraform e conexão do Workload Identity Provider com Github Actions.** |
| --- | --- |
| **Leonardo Victor Bezerra de Oliveira** | **Criação dos chatbots no ADK, incluindo um chatbot que consome os dados da ONS, um chatbot que consome os dados da Sauter, e um chatbot orquestrador que recebe inputs e faz a orquestração dos outros agentes.** |
| **Maria Eduarda Grigório de Melo** | **Transformação dos dados vindos da API, deixando-os pronto para análise e consumo pelo agente. E automação do processo de transformação dos dados.** |
| **Pedro Lucas Lima Duarte** | **Criação da API que exporta os dados da ONS para o bucket no Cloud Storage, e pega os dados tratados para exibição. Deploy da API no Cloud Run no GCP e workflow da API no Github Actions.** |

**Processo de desenvolvimento**

* **Infraestrutura via Terraform**

A infraestrutura é definida e gerenciada usando Terraform, seguindo uma abordagem modular.



O uso de módulos permite agrupar recursos relacionados, tornando o código de infraestrutura mais limpo e fácil de gerenciar. Na implementação, existem módulos distintos para cada área chave da arquitetura, sendo eles **Module "iam"**, **Module "artifact\_registry"**, **Module "bucket",**  **Module "big\_query" e Module "cloudrun".**

1. **Module "iam"**: Dedicado à criação de Service Accounts (SAs) e à atribuição de permissões, incluindo a implementação do Workload Identity Federation. Este módulo cria a Service Account **cloudrun-sa**, utilizada pelo **Module "cloudrun"** para gerar sua estrutura e dependências, e cria a Service Accounts **github-actions-sa**, que é utilizada pelo pipeline de CI/CD para automação de deploy e gerenciamento de infraestrutura. Suas saídas (Outputs) exportam os e-mails das SAs criadas, que são consumidos pelo módulo cloudrun e pelo pipeline CI/CD. Para utilização é necessário ativar previamente a **Identity and Access Management (IAM) API** e **Cloud Resource Manager API**, no GCP.
2. **Module "artifact\_registry"**: Dedicado à criação do repositório no **Artifact Registry**, responsável por armazenar as imagens Docker da API. Saídas (Outputs) repository\_id: exporta o identificador do repositório criado, permitindo que outros módulos (como o cloudrun) façam referência direta ao repositório de imagens para deploys. Requer que a **Artifact Registry API** esteja ativada previamente no projeto GCP.
3. **Module "bucket"**: Focado na criação do Google Cloud Storage (GCS). Provisiona um bucket no Google Cloud Storage (GCS) que funciona como camada de armazenamento primário para os dados. Requer que a **Cloud Storage API** esteja ativada previamente no projeto GCP.
4. **Module "big\_query"**: Gerencia o dataset e a tabela externa. Estabelece o Data Warehouse e a conexão com dados externos armazenados no GCS. Ele cria o recurso google\_bigquery\_dataset com o dataset\_id padrão "reservatorios\_externos\_dataset", que recebe os dados diretos da pasta raw na bucket. Requer que a **BigQuery API** esteja ativada previamente no projeto GCP.
5. **Module "cloudrun"**: Encapsula a implantação do serviço sem servidor da API. Responsável pelo deploy da API de dados como um serviço gerenciado e sem servidor no Google Cloud Run. responsável pelo deploy da API de dados como um serviço gerenciado e sem servidor no Google Cloud Run. O serviço é integrado à Service Service Account **cloudrun-sa**, fornecida pelo módulo IAM. Requer que a **Cloud Run API** esteja ativada previamente no projeto GCP.

**Backend**

A configuração do Terraform utiliza um backend GCS para armazenar o estado, usando o bucket terraform-state-bucket-sauter-university-2025 com um prefixo terraform/state. Garantindo persistência, versionamento e colaboração segura do estado entre múltiplos usuários.

**Integração CI/CD com GitHub Actions**

O Terraform está totalmente integrado a um pipeline de CI/CD usando GitHub Actions. Este workflow é chamado de 'Terraform CI/CD' e foi adicionado como um workflow específico. Ele é acionado por pushes ou pull requests nos branches develop ou main, ou por mudanças dentro do diretório terraform/. Todas as variáveis sensíveis, como **workload\_identity\_provider: ${{ secrets.WORKLOAD\_IDENTITY\_PROVIDER }}** e **service\_account: ${{ secrets.SERVICE\_ACCOUNT }}**, são armazenadas de forma segura na seção de Secrets do GitHub. Isso significa que os valores reais não ficam expostos no código-fonte ou nos arquivos de configuração, sendo injetados apenas durante a execução dos workflows, aumentando a segurança e evitando vazamentos de informações sensíveis.

As etapas do pipeline Terraform incluem:

1. **Autenticação**: O passo Authenticate to Google Cloud utiliza o Workload Identity Provider e Service Account (SA) secrets para autenticação. O uso de Workload Identity Federation é um requisito explícito do projeto.
2. **Setup**: Configuração do Terraform usando hashicorp/setup-terraform@v3.
3. **terraform Init**: Este comando descarrega os providers e inicializa o backend remoto para guardar o estado da infraestrutura
4. **terraform Format**: É executado com a flag -check para validar a formatação do código.
5. **terraform Plan**: Executado com variáveis **(-var="project\_id=${{ vars.GCP\_PROJECT\_ID }}")** para gerar o plano de execução, sem continuar em caso de erro. Este passo é nomeado plancx.
6. **terraform Apply**: A execução do apply é estritamente condicional: só ocorre em eventos de push nos branches develop ou main e somente se o passo de plan for bem-sucedido. O apply é realizado com a **flag -auto-approve**.

**Levantamento da infraestrutura**

Pré-requisitos Essenciais:

1. Terraform Instalado:

Verifique a instalação com o comando: **terraform --version**

1. Google Cloud SDK (gcloud) Instalado:

Verifique a instalação com o comando: **gcloud --version**

1. Acesso Autenticado ao GCP:

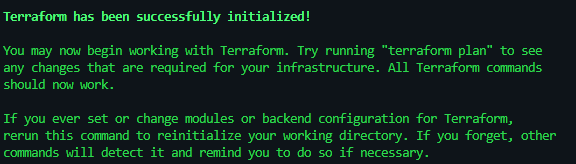
É preciso ter permissão para executar comandos no projeto. Autentique a sua máquina executando o seguinte comando. Ele abrirá um navegador para que você faça login com a sua conta Google: **gcloud auth application-default login**

Comandos de Execução do Terraform:

Abra o seu terminal e navegue para a pasta terraform/ do projeto. Execute os seguintes comandos pela ordem.

Inicializar o Terraform (**init**): Este comando descarrega os providers e inicializa o backend remoto para guardar o estado da infraestrutura.

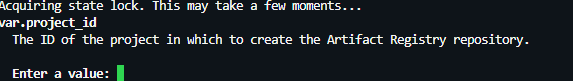
**terraform init**

****

Planear as Alterações (**plan**)

Este comando faz uma "simulação". Ele compara o seu código com a infraestrutura que já existe no GCP e mostra exatamente o que será criado, alterado ou destruído. Nenhuma alteração é feita nesta fase. Será necessário informar o ID do projeto.

**terraform plan**



Aplicar as Alterações (**apply**): Este é o comando que efetivamente cria a infraestrutura na nuvem. Ele irá mostrar o plano novamente e pedir uma confirmação final. Digite **yes** e pressione Enter para continuar.

**terraform apply**

Após a conclusão, todos os recursos (Bucket, BigQuery, IAM, etc.) estarão criados no seu projeto GCP.

Destruir a Infraestrutura (Opcional): Se precisar apagar todos os recursos que criou com o Terraform para evitar custos, use o comando abaixo. Ele também pedirá uma confirmação.

**terraform destroy**

* **API ONS**

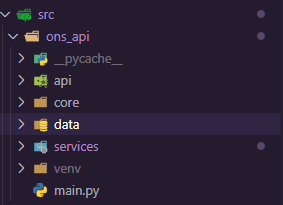
Abordando desenvolvimento da API da ONS cujo escopo era:

* API que tivesse pelo menos 2 endpoints (um de GET e um de POST)
* POST seria capaz de pegar os dados de reservatório da ONS e enviar para o CloudStorage partcionando.
* GET seria capaz de retornar os dados paginados já tratados vindos do BigQuery.

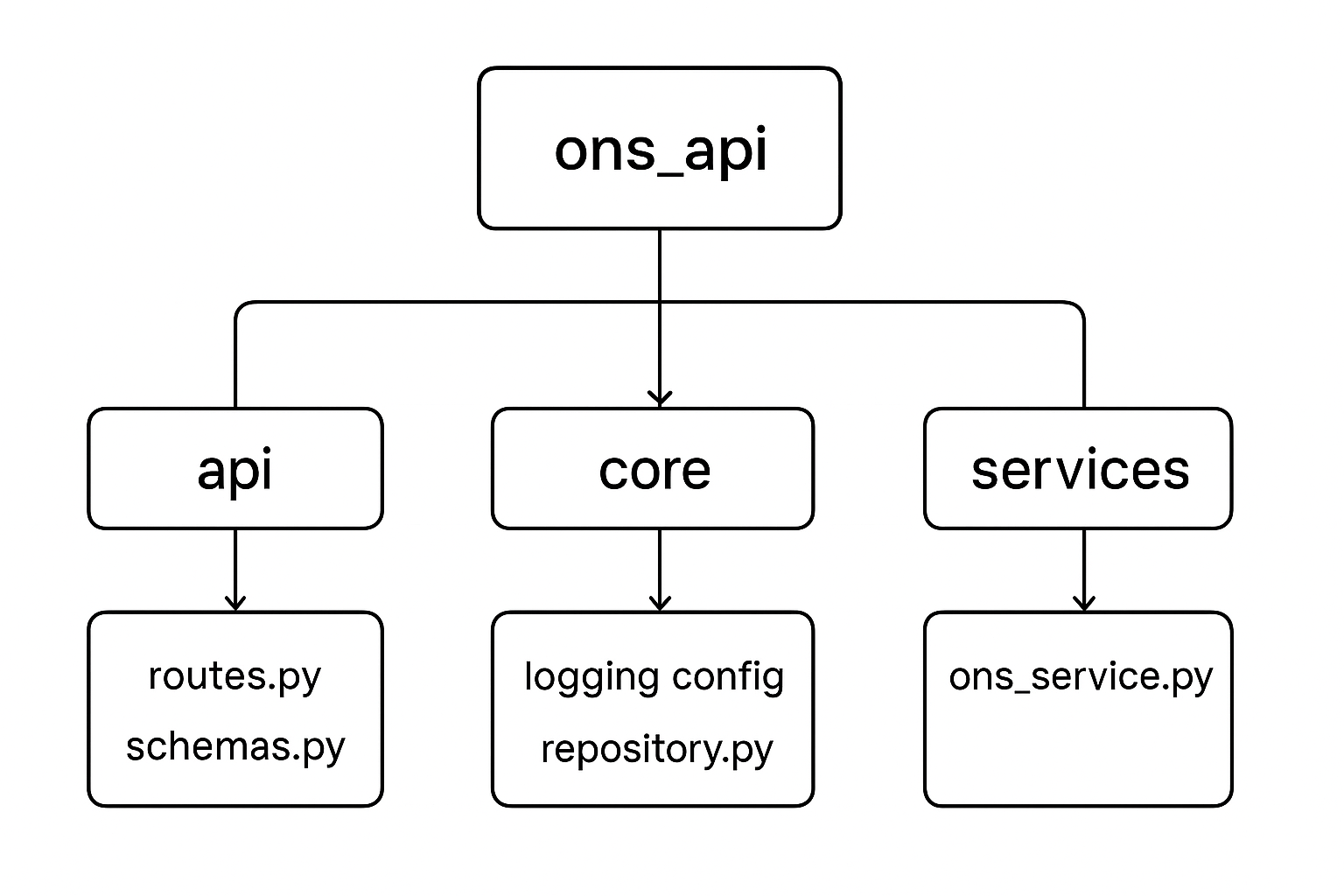
O desenvolvimento da API iniciou de forma tranquila, com a escolha da biblioteca **FastAPI** motivada pela familiaridade com a ferramenta. No início do projeto, houve um mal-entendido em relação às funcionalidades da API: acreditava-se que a rota **GET** deveria retornar os dados diretamente do site da ONS, enquanto a rota **POST** seria responsável apenas por exportar os dados para o bucket. Dessa forma, a “primeira versão” da API funcionava nesse modelo.

Com o entendimento mais claro do escopo, a rota **POST** passou a ser desenvolvida adequadamente, apresentando funcionamento correto, mas ainda realizando a requisição de arquivos diretamente pela URL base do site — algo não recomendado. Esse ponto precisou ser ajustado ao longo do processo. Apesar de não terem sido registradas imagens dessa fase inicial, a seguir é apresentada a estrutura final da API, com trechos de código explicados e as principais dificuldades enfrentadas durante o desenvolvimento.

**Estrutura da api:**



Aqui a estrutura de pastas da API, ela foi organizada de forma modular, visando manutenção facilitada, escalabilidade e clareza no fluxo de dados. A separação por responsabilidades permite que cada componente tenha um papel bem definido, evitando acoplamento excessivo e tornando o código mais legível.



***routes.py:*** define os endpoints da API, ou seja, as rotas que o usuário poderá acessar.

***schemas.py:*** contém os modelos de dados utilizados para validação e documentação das requisições e respostas.

***logging\_config.py:*** centraliza o logging.

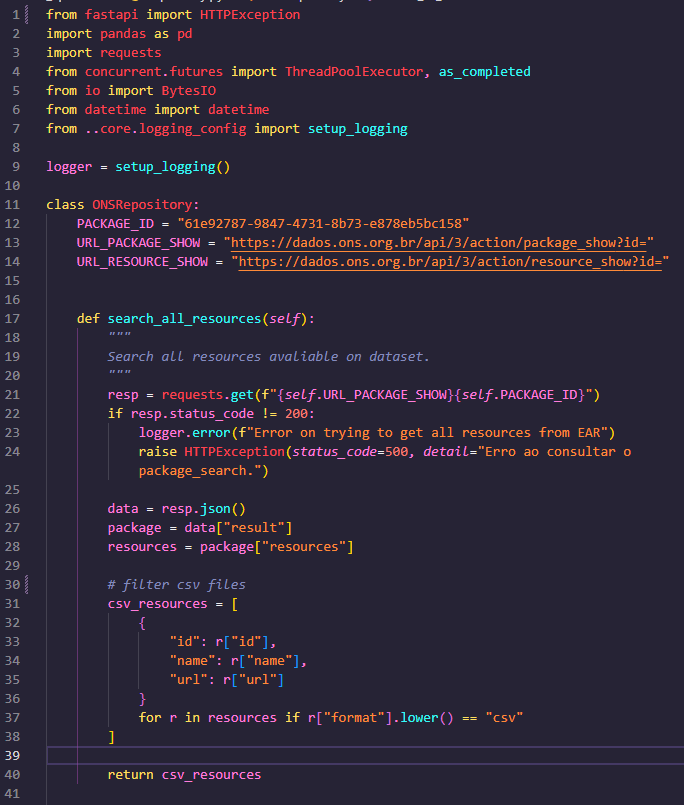
***GCPHandler.py:***  lida diretamente com serviços do Google Cloud, como BigQuery e Storage.  
***repository.py:*** atua como camada de acesso a dados, abstraindo a fonte de dados (arquivos, banco, etc.) da lógica de negócio. Essa separação permite que alterações na forma de armazenar ou acessar dados não impactem a camada de serviços.

***ons\_service.py:*** aqui reside a lógica de negócio da aplicação. É responsável por orquestrar chamadas ao repositório de dados (repository) e tratar os dados antes de enviá-los para a camada de API.

***main.py:*** localizado na raiz da API, é o ponto de entrada da aplicação, inicializando o FastAPI e incluindo os routes definidos na pasta api.

Agora passando um pouco sobre cada arquivo, explicando os pontos cruciais do código em si e a responsabilidade das funções:

* **repository.py**

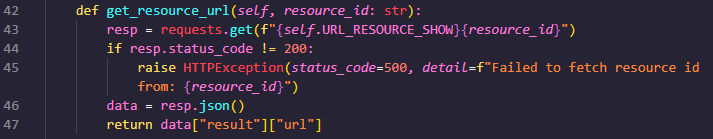


*Definindo o setup de logger que está escrito no* ***logging\_config.***

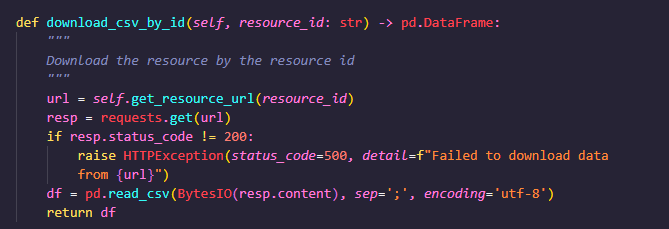
*Classe* ***ONSRepository*** *que será responsável por procurar todos os recursos disponíveis no package id (basicamente é a pasta onde estão todos os arquivos de dados de reservatórios EAR), pegar a url do recurso, realizar o download dos recursos que estão no formato csv, e extrair os arquivos necessários para um certo intervalo de datas.*

***Função search\_all\_resources(self):*** *realiza um request(pedido) para obter os recursos de um determinado* ***PACKAGE\_ID*** *que nesse caso está definido como uma constante no código, utiliza a url que mostra todos os recursos do package. Foi uma dificuldade muito grande encontrar essa url que mostrasse todos os recursos de um determinado package, mas com esses recursos encontrados, é capaz de filtrar somente os que são no formato csv através do campo “format” desses recursos, retornando assim, seus respectivos id, name e url.*

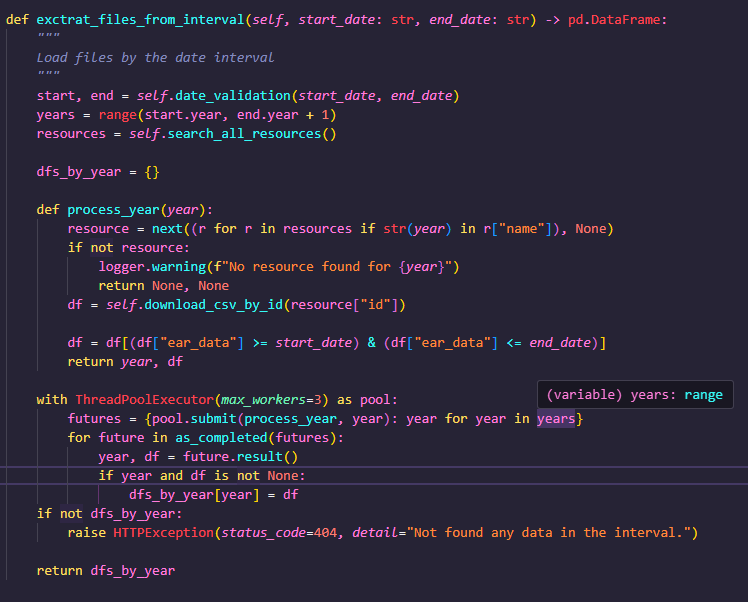
*Em questão de escalonamento, foi uma alternativa encontrada visto que sempre terá todos os recursos do package\_id, então caso seja adicionado um recurso novo, ele retornará, ou caso eu queira extrair dados de algum outro package\_id, é possível somente alterando o* ***PACKAGE\_ID.***

****

***Função get\_resource\_url(self, resource\_id: str):*** *faz uma requisição baseada na url mostra detalhes do recurso e retorna os campos “result” e “url” do recurso.*



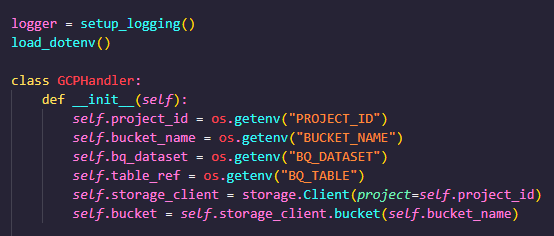
***Função download\_csv\_by\_id(self, resource\_id: str):*** *faz o download do recurso de acordo com o id. Obtém a url chamando a função* ***get\_resource\_url,*** *fazendo um request para obter essa url, e utilizando o pandas, criando um DataFrame através da função read\_csv do pandas, definindo o conteúdo do csv eu vem em Bytes, definindo o separador do csv e o modo de encoding.*

**

***Função exctract\_files\_from\_interval(self, start\_date: str, end\_date: str):*** *após buscar todos os recursos do package, a função define um dicionário dfs\_by\_year para armazenar os DataFrames correspondentes a cada ano.*

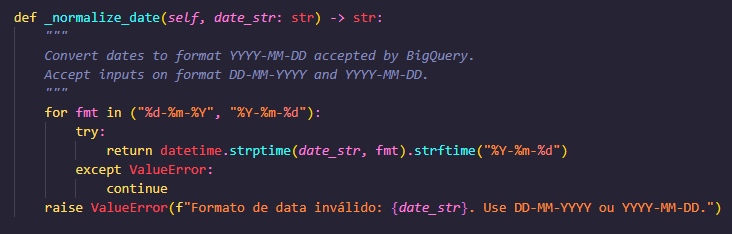
*Em seguida, é definida a função interna process\_year(year), que tem a responsabilidade de processar os dados de um ano específico, ela procura no conjunto de recursos aquele recurso o qual contém em seu nome o recurso atual do loop. Caso o recurso exista, ele faz o download do csv e em seguida filtra as linhas do DataFrame para incluir apenas aquelas cujo valor da coluna ear\_data esteja entre o intervalo de datas. Com tudo isso sendo feito para conseguir obter todos os recursos de um package id, pegar a url de cada um desses recursos, fazer o download do csv, depois filtrar para somente os do intervalo de datas, a api estava ficando extremamente lenta ao fazer o POST dos dados. Então foi necessário utilizar o paralelismo para o processamento de cada ano do intervalo com 3 threads simultâneas, a medida que as threads terminam, o resultado é adicionado ao dicionário dfs\_by\_year, associando cada ano ao seu dataFrame filtrado.*

* **GCPHandler.py**

****

***Class GCPHandler:*** *Essa classe é responsável por interagir com Google Cloud Storage (GCS) e BigQuery (BQ), abstraindo operações comuns como upload de arquivos, verificação de existência e consulta de dados.*

***\_\_init\_\_;***  *Inicializa a classe carregando variáveis de ambiente (projeto, bucket, dataset e tabela BQ), cria um cliente de Storage (storage.Client) e obtém referência ao bucket, todas as operações do GCS utilizarão esse client e esse bucket.*

**

***\_\_nomarlize\_date(self, date\_str: str):*** *função interna que converte datas para o formato YYYY-MM-DD, aceitando entrada nos formatos DD-MM-YYYY ou YYYY-MM-DD, garante que consultas no BigQuery tenham datas válidas, evitando erros de SQL, pois uma grande dificuldade aconteceu ao tentar interagir com o BQ, interagir com a coluna de datas retornava um erro do BQ.*

**

***blob\_exists(self, blob\_name: str):*** *verifica se um determinado arquivo (blob) já existe no bucket do GCS, evita uploads duplicados de arquivos que já existem, então caso já tenha feito upload do arquivo, ele retorna false, desse modo garantindo que só tenha arquivos únicos no bucket.*

**

***check\_today\_ingestion(self, source\_name=”ons):*** *responsável por verificar se já houve ingestão no dia atual, listando blobs(caminhos) com prefixo* ***raw/{source\_name}/dt=DD-MM-YYYY/,*** *retornando a lista de blobs existentes ou lista vazia. É uma boa prática evitar ingestão necessária, então se os dados da ONS em teoria só são atualizados diariamente, se já houve uma ingestão no bucket hoje, não há necessidade de ser fazer outra, visto que os dados não atualizaram.*

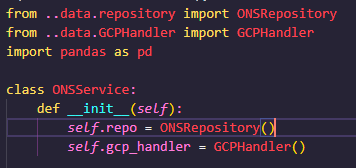


***upload\_parquet\_to\_bucket(self, dfs\_by\_year, source\_name):*** *faz upload de DataFrames para o GCS em formato Parquet, organizando por ano e dia (****raw/{source\_name}/dt=DD-MM-YYYY****). Chama a função* ***check\_today\_ingestion*** *para evitar uploads duplicados no mesmo dia e* ***blob\_exists*** *para evitar sobrescrever arquivos antigos. Como boa prática da engenharia de dados, converte o DataFrame para string para que seja enviado tudo como string e evitar erros, além disso faz o upload em memória com* ***BytesIO.*** *Retorna a lista de arquivos enviados e um booleano indicando se houve upload, mantendo organização temporal no bucket e evitando duplicidade.*

**

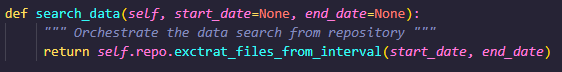
***query\_bigquery(self, start\_date, end\_date, page, page\_size):*** *faz query paginada no BigQuery, permitindo filtrar dados entre* ***start\_date*** *e* ***end\_date. I****nterage com* ***\_normalize\_date*** *para garantir datas corretas. Calcula o offset para paginação baseado em page e page\_size e executa duas queries, a primeira retorna subset dos registros, e a segunda retorna o total de linhas disponíveis no intervalo (sem paginação).Essa função permite consumir grandes volumes de dados de forma controlada e eficiente, mas não é uma função bem escrita, não está seguindo o princípio de responsabilidade única, fazendo várias coisas. O ideal seria jogar as várias funcionalidades dessa função em outras para dividir responsabilidades, como essa função é a utilizada no GET, foi a última a ser desenvolvida pois precisava que os dados estivessem tratados. Vários problemas muito complexos surgiram com essa função e o service que a consome, por algo relacionado a datas, houve uma dificuldade de depurar visto que havia uma dificuldade de entender se o problema era na função ou se era problema no tratamento de dados, mas de modo geral, só conseguia retornar dados até 2023, aparentemente os dados de 2024 e 2025 tem algum outro formato de data e por isso que fpo feita a normalização tanto aqui quanto uma conversão também no service para resolver todos os problemas de datas.*

* **ons\_service.py**

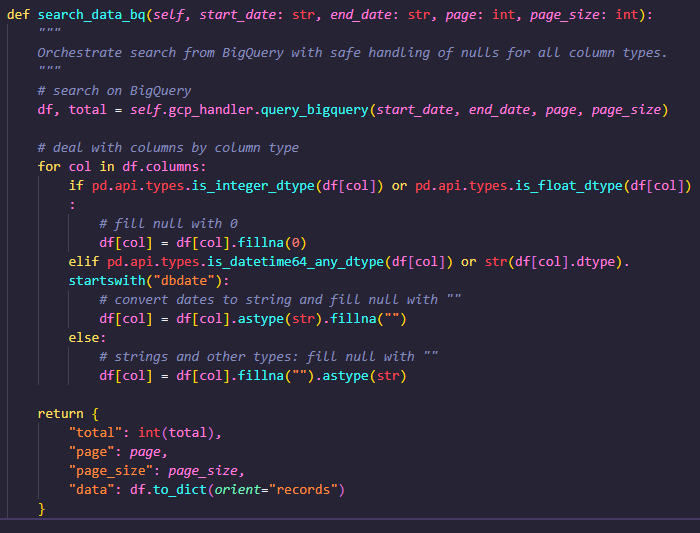
****

***Class ONSService:*** *A classe é responsável por orquestrar a busca e tratamento de dados na sua API. Ela atua como camada de serviço, intermediando a comunicação entre a camada de dados (ONSRepository e GCPHandler) e a camada de apresentação (API/rotas).*

***\_\_init\_\_(self): ONSRepository*** *abstrai o acesso a arquivos CSV ou outros dados brutos, e o GCPHandler: abstrai operações com GCS e BigQuery.*

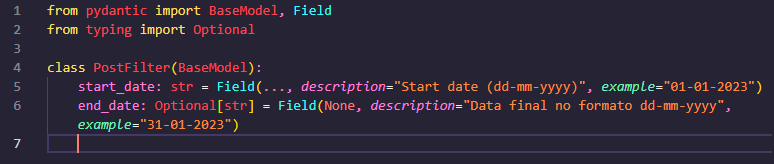
******

***search\_data(self, start\_date, end\_date):*** *orquestra a busca de arquivos brutos pelo intervalo de datas, chamando* ***extract\_files\_from\_interval*** *do* ***ONSRepository*** *que valida o intervalo de datas, busca arquivos correspondentes aos anos dentro do intervalo, baixa e filtra os arquivos CSV em DataFrames. Esta função é usada quando a API precisa de dados locais brutos para processamento ou upload.*

**

***search\_data\_bq(self, start\_date, end\_date, page, page\_size):*** *orquestra a busca de dados diretamente do BigQuery com paginação. Chama query\_bigquery do GCPHandler para normalizar datas, consultar BigQuery e retornar DataFrame e o total de registros. Faz o tratamento de colunas, pois uma dificuldade enfrentada foi a parte de tratamento de colunas, especialmente datas estava dando muitos problemas. Basicamente o GET funcionava normalmente para dados de 2000 até 2023. Porém GET em dados de 2024 para frente era retornado* ***TypeError: Invalid value '' for dtype 'Int64',*** *então foi necessário tratar todas as colunas de acordo com o tipo dela, números substitui null por 0, datas converte para string e substitui null por “”, strings e outros tipos substitui null por “”. Retorna um dicionário estruturado com o total, o número da página, o número de dados por página e os dados.*

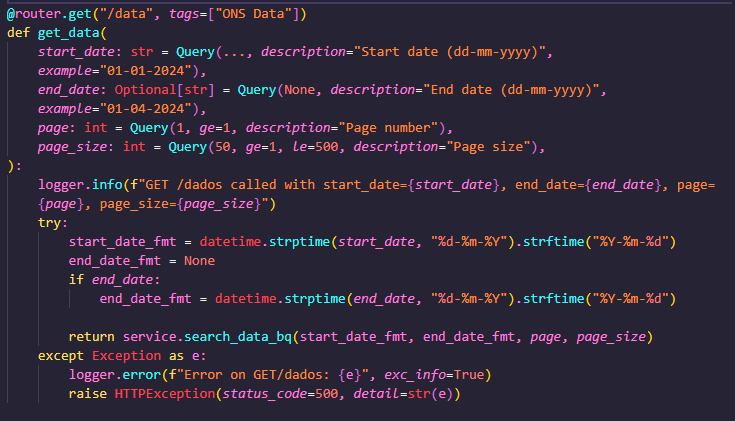
* **schemas.py**

****

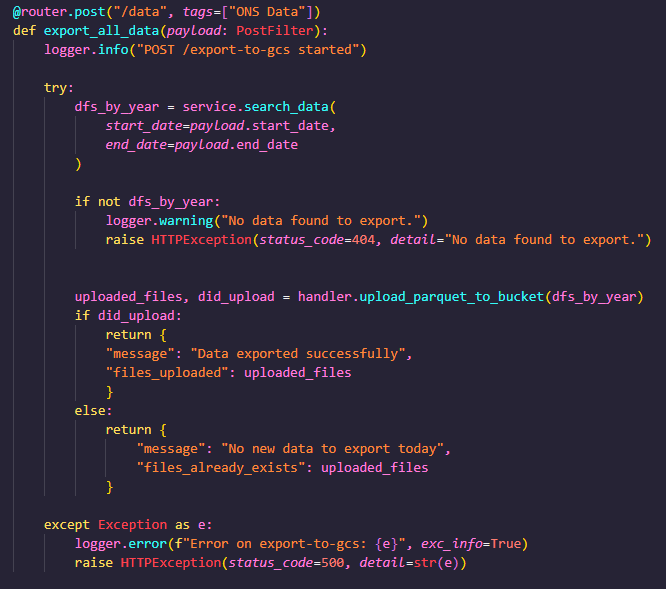
*Esse arquivo define modelos de dados utilizando o Pydantic.*

*Ele serve para validar e documentar os dados que chegam via requisição (request body, query params, etc). Esse PostFilter vai aparecer em endpoints como parâmetro de entrada.*

* **routes.py**

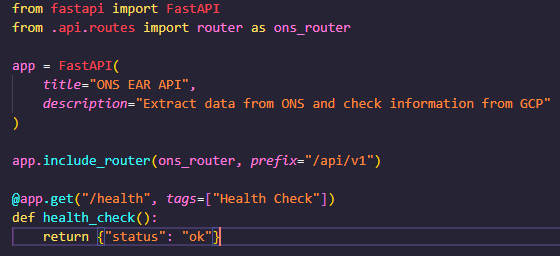
****

***Endpint GET /data:*** *converte as datas de dd-mm-yyyy → yyyy-mm-dd (padrão aceito pelo BigQuery), Chama* ***service.search\_data\_bq*** *para consultar no BigQuery, e o ONSService retorna dados paginados e já tratados. E ualquer falha gera log e retorna HTTP 500.*

**

***Endpoint POST /data:*** *recebe um payload validado pelo PostFilter (com start\_date obrigatório e end\_date opcional). Chama* ***service.search\_data*** *para buscar dados brutos (não do BigQuery, mas do repositório/arquivos). Caso encontre, usa* ***handler.upload\_parquet\_to\_bucket***  *para:*

* *Converter DataFrames em parquet.*
* *Subir no GCS.*
* *Evitar sobrescrever arquivos já existentes.*
* **main.py**

****

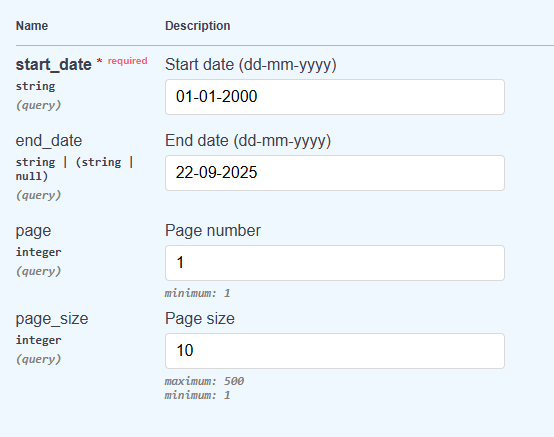
*Cria a instância principal app, que será usada pelo servidor, e anexa todas as rotas definidas em* ***routes.py.***

***Endpoint Health Check:*** *Um endpoint simples e rápido para verificar se a API está rodando.*

* **Resultados da API**

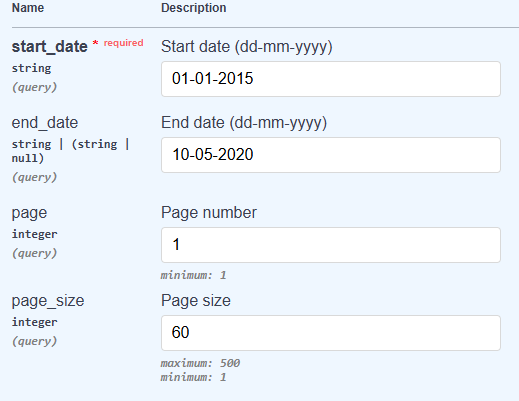
**POST com dados já enviados ao bucket:**

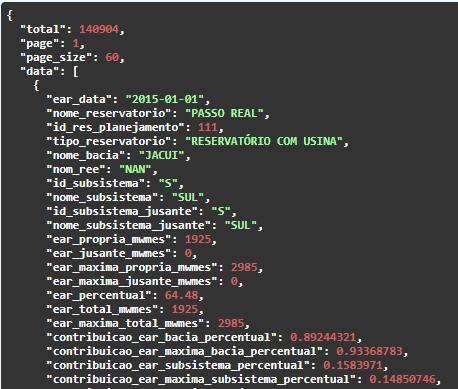
****

**RQUISIÇÃO GET: (get dados de 2000 até o último dado registrado)  
**

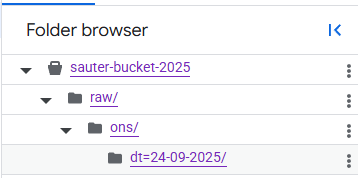
****

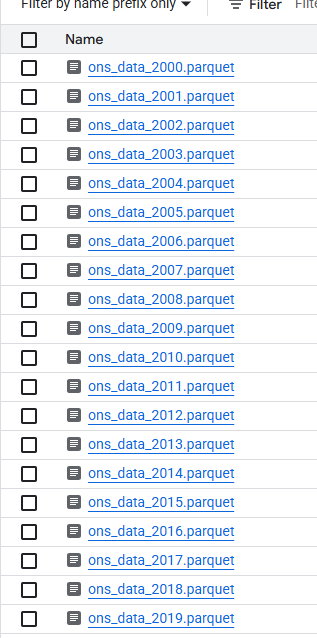
**REQUISIÇÃO GET (get dados 2015 a 2020):**

****

****

**RESULTADO NO BUCKET**

****

****

**Requisitos**

**Python 3.11+**

**Poetry ou pip para gerenciar dependências**

**Google Cloud SDK (gcloud) configurado e autenticado**

**Conta de serviço do GCP com permissões para:**

* **Cloud Storage (GCS) – leitura e escrita em buckets**
* **BigQuery – execução de queries**
* **Artifact Registry – caso seja feito deploy com container**

**Variáveis de ambiente configuradas (pode-se usar um arquivo .env):**

**PROJECT\_ID=<seu-id-do-projeto-gcp>**

**BUCKET\_NAME=<nome-do-bucket-gcs>**

**BQ\_DATASET=<nome-do-dataset-no-bigquery>**

**BQ\_TABLE=<nome-da-tabela-no-bigquery>**

**Como Rodar Localmente**

1. **Criar ambiente virtual e instalar dependências**

**python -m venv venv**

**source venv/bin/activate # Linux/Mac**

**venv\Scripts\activate # Windows**

**pip install -r requirements.txt**

1. **Rodar a API**

**uvicorn src.main:app --reload --host 0.0.0.0 --port 8000**

1. **A API estará disponível em:**

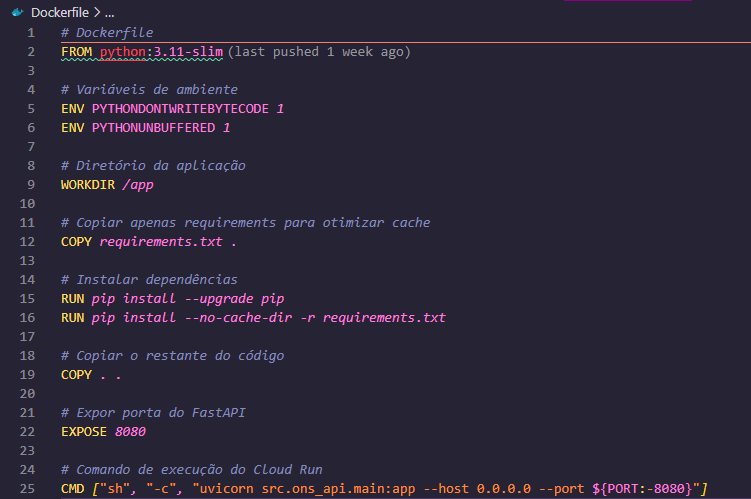
**http://localhost:8000**

* **Deploy API no CloudRun**

O objetivo consistia em construir a imagem Docker da API e realizar o deploy no serviço Cloud Run do GCP. Essa etapa do projeto apresentou diversos desafios relacionados ao Docker local, porém todas as dificuldades foram superadas e o processo foi concluído com sucesso.

1. **Construção do Dockerfile**

A primeira parte foi construir o dockerfile com as especificidades da api:



1. **Configurando docker para autenticar-se com serviços de container no GCP**

****

1. **Buildando a imagem remotamente nos servidores do Google Cloud**

**gcloud builds submit --tag us-east1-docker.pkg.dev/PROJECT\_ID/REPO\_PROJECT/ons-api**

1. **Deploy da imagem no CloudRun**

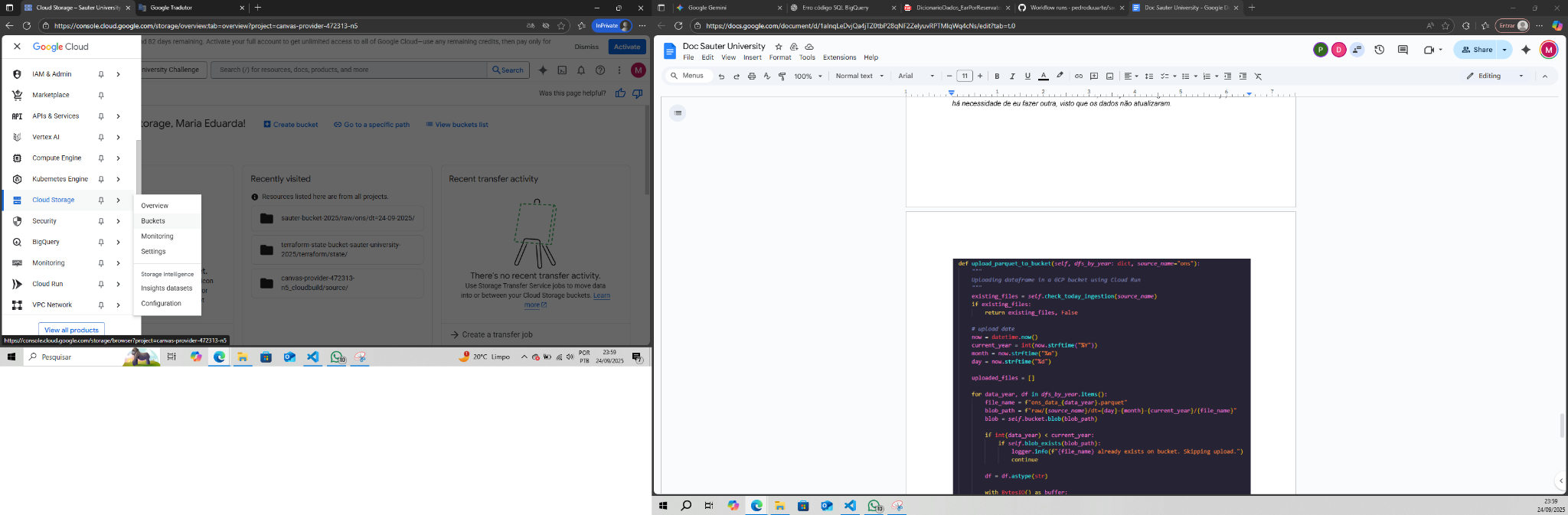
**gcloud run deploy sauter-cloud-run-service --image REGION-docker.pkg.dev/PROJECT\_ID/REPO\_PROJECT/ons-api –region REGION –platform managed –allow-unauthenticated –project PROJECT\_ID**

* **Transformação e Automação de Dados**

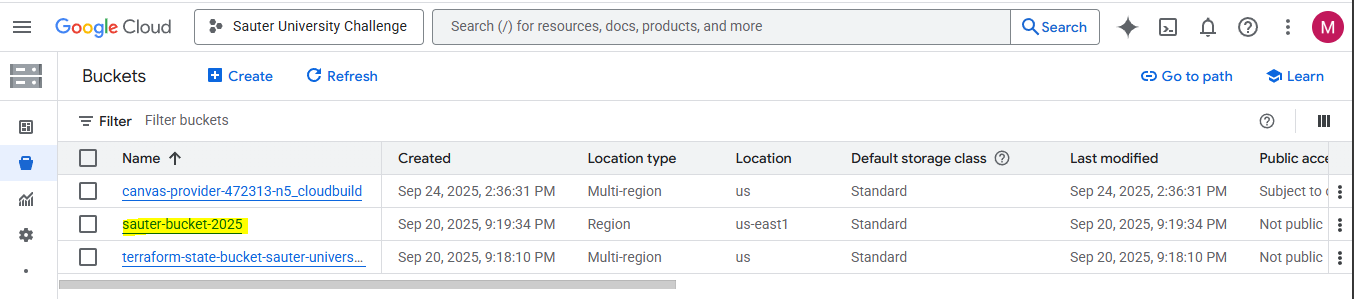
O objetivo deste projeto foi criar um pipeline robusto para a transformação dos dados de reservatórios vindos de uma API, garantindo que estejam prontos para análise e consumo. O processo foi automatizado para rodar em intervalos programados, mantendo os dados sempre atualizados.

1. ***Conexão com a Origem de Dados:***

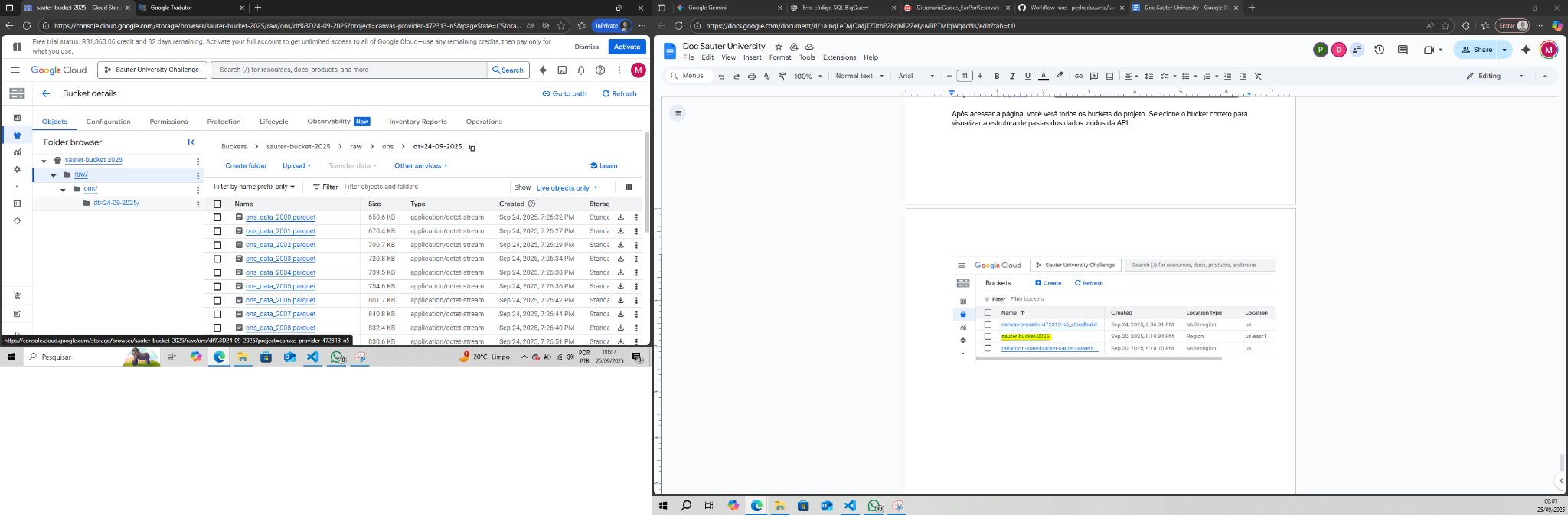
A primeira etapa foi estabelecer uma conexão com os arquivos em formato Parquet, que estão armazenados no bucket do Google Cloud Storage. Para acessar é só apertar no menu lateral esquerdo ( ≡ ) e procurar **Cloud Storage** -> **Buckets.**



Após acessar a página, você verá todos os buckets do projeto. Selecione o bucket correto para visualizar a estrutura de pastas dos dados vindos da API.



Ao acessar o bucket, você verá a estrutura de pastas dos dados vindos da API. Navegue até a pasta mais recente **dt=24-09-2024** para visualizar todos os arquivos disponíveis.



Em vez de carregar os dados fisicamente no BigQuery, foi criada uma Tabela Externa. A grande vantagem de uma tabela externa é que ela age como um "link" para a origem dos dados. Isso evita a duplicação de informações e garante que as análises sejam feitas sempre na versão mais recente dos arquivos, sem a necessidade de recarregar os dados.

## **Acesso ao BigQuery**

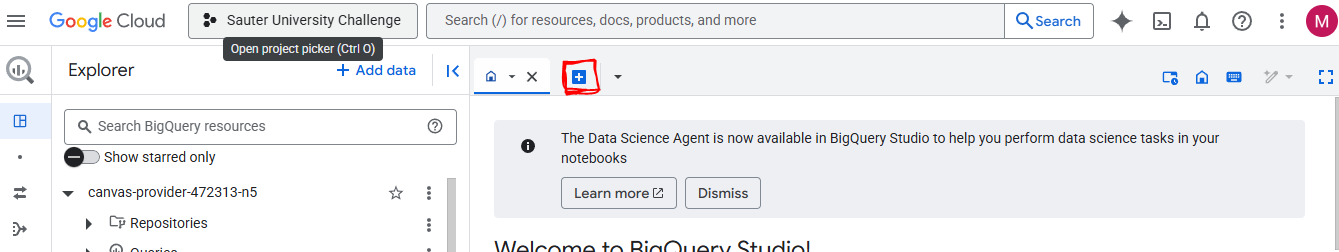
O ambiente do banco de dados foi provisionado via **Terraform**, uma ferramenta de infraestrutura como código. Para acessá-lo:

1. No console do Google Cloud, clique no menu de navegação lateral ( **≡** ).
2. Busque e selecione o serviço **BigQuery**.
3. O BigQuery irá exibir a tela do projeto canvas-provider-472313-n5, onde toda a estrutura do nosso banco de dados está localizada.



**Tabela Externa (Código):**

Para abrir o editor de consultas e inserir os comandos SQL, clique no botão **+ SQL query**

****

**Comando SQL:**

Este trecho de código SQL cria uma Tabela Externa no BigQuery. O nome da tabela é **novos\_dados\_ons** e ela está localizada no seu projeto (**canvas-provider-472313-n5**) e no seu conjunto de dados (**reservatorios\_externos\_dataset**).

O objetivo principal deste código é conectar o BigQuery diretamente aos seus arquivos em formato Parquet que estão no Cloud Storage, sem a necessidade de carregar os dados fisicamente.

***CREATE OR REPLACE EXTERNAL TABLE:*** Esta instrução cria a tabela, ou a substitui se ela já existir. Isso é útil para quando você precisa atualizar a definição da tabela, por exemplo, se a localização dos arquivos mudar.

***WITH PARTITION COLUMNS:*** Essa opção é utilizada para otimizar o desempenho. Ela indica que a tabela usará as colunas de partição (dt=24-09-2024, nesse caso) para filtrar os dados de forma mais eficiente durante as consultas.

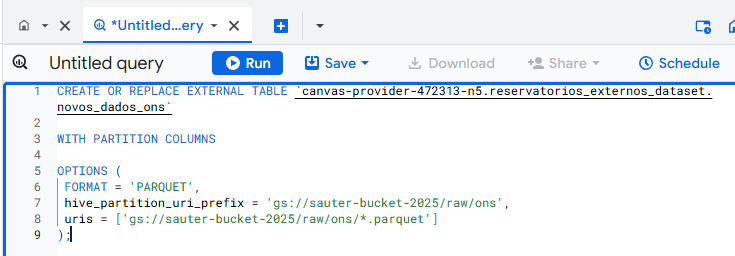
***OPTIONS:*** Dentro desta seção, você especifica os detalhes de como o BigQuery deve ler os arquivos no Cloud Storage:

***FORMAT = 'PARQUET':*** Informa ao BigQuery que o formato dos arquivos é Parquet.

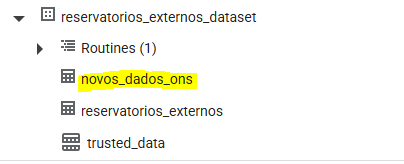
***hive\_partition\_uri\_prefix:*** Este é o caminho base onde o BigQuery vai procurar os seus arquivos particionados. Ele aponta para a pasta onde os dados brutos da API estão sendo salvos.

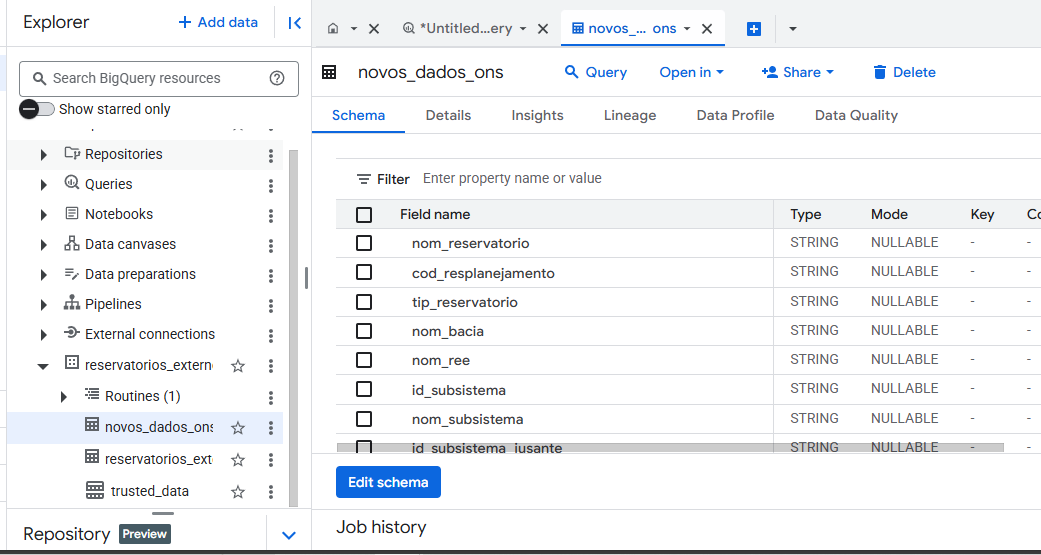
***uris:*** Esta linha define a localização exata dos seus arquivos. O \*.parquet indica que o BigQuery deve ler todos os arquivos com a extensão .parquet que estão na pasta.

Clique em **Run** para executar o comando.

****

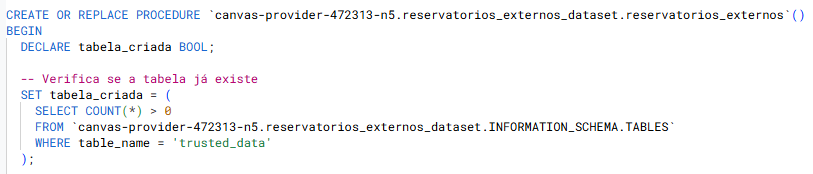
Após esse comando, será criada a tabela **novos\_dados\_ons** e ao clicar em cima dessa tabela, ela vai abrir na lateral os dados que contém dentro dela.





### **Procedimento Armazenado (Código Principal)**

O código a seguir cria um **procedimento armazenado**, que é um conjunto de comandos SQL salvos no BigQuery. Ele automatiza a lógica de transformação dos dados para garantir que a tabela final esteja sempre limpa e atualizada.



***CREATE OR REPLACE PROCEDURE ... BEGIN****:* Esta linha inicia a criação do seu procedimento armazenado. Ele recebe um nome e um local, e tudo o que estiver entre BEGIN e END será executado.

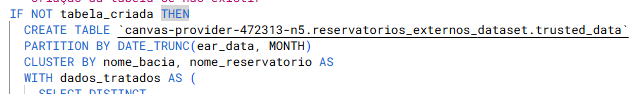
***DECLARE tabela\_criada BOOL;****:* Uma variável chamada tabela\_criada é declarada. O tipo BOOL (booleano) significa que ela só pode armazenar dois valores: TRUE (verdadeiro) ou FALSE (falso).

***SET tabela\_criada = (...)****:* A variável tabela\_criada recebe o resultado de uma consulta.

***FROM ...INFORMATION\_SCHEMA.TABLES****:* Esta é uma tabela especial do BigQuery que contém metadados sobre todas as tabelas e views em um conjunto de dados. Ao consultá-la, você pode obter informações sobre suas tabelas.

***WHERE table\_name = 'trusted\_data'****:* A consulta é filtrada para buscar apenas a tabela com o nome trusted\_data.

***SELECT COUNT(\*) > 0****:* A consulta conta quantas tabelas com o nome trusted\_data existem. Se o resultado for 1 (ou mais), significa que a tabela existe, e o resultado TRUE é atribuído à variável tabela\_criada. Se o resultado for 0, a variável recebe FALSE.



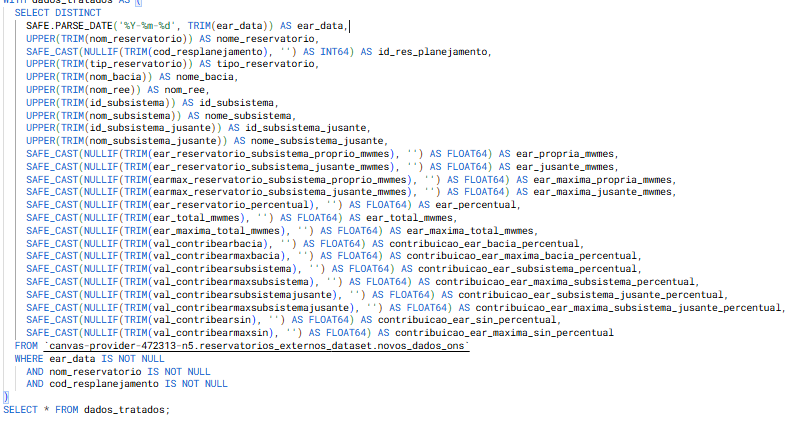
***IF NOT tabela\_criada THEN****:* Esta linha é o início da condição. Ela verifica se a tabela trusted\_data não existe no seu conjunto de dados. Se a condição for verdadeira, todo o código que vem a seguir, até o ELSE, será executado.

***CREATE TABLE ...****:* Este comando cria a tabela de destino, que é onde os dados limpos serão armazenados. A tabela recebe o nome trusted\_data e é salva no local padrão do seu projeto e dataset.

***PARTITION BY DATE\_TRUNC(ear\_data, MONTH)****:* Esta é uma etapa de otimização de performance. Ela instrui o BigQuery a dividir a tabela em partições (pequenos blocos) com base no mês do campo ear\_data. Em uma consulta, o BigQuery só precisará ler a partição necessária, o que reduz o custo e acelera a execução.

***CLUSTER BY nome\_bacia, nome\_reservatorio****:* Essa é outra técnica de otimização. Ela organiza os dados dentro de cada partição com base nas colunas nome\_bacia e nome\_reservatorio. Isso é ideal para consultas que filtram por esses campos, pois o BigQuery não precisará escanear a tabela inteira, o que melhora significativamente a velocidade da consulta.

***AS WITH dados\_tratados AS (...)****:* Esta parte do código usa uma CTE (Common Table Expression) para definir a fonte de dados que será inserida na nova tabela. A CTE é onde toda a lógica de limpeza e transformação dos dados brutos é aplicada, como vimos na explicação anterior.



Este trecho de código define uma **CTE (Common Table Expression)**, uma consulta temporária que realiza a limpeza e padronização dos dados brutos antes que eles sejam inseridos na tabela final.

***SELECT DISTINCT****:* A cláusula DISTINCT remove linhas duplicadas, assegurando que cada registro seja único.

**Funções de Limpeza e Conversão**: Cada campo passa por uma série de funções que limpam e padronizam os dados.

* ***TRIM()****:* Remove espaços em branco do início e do fim das strings.
* ***UPPER()****:* Converte todo o texto para letras maiúsculas para garantir a consistência dos dados de texto.
* ***SAFE.PARSE\_DATE()****:* Converte strings para um formato de data, e o SAFE garante que, se a conversão falhar, ele retorna NULL em vez de gerar um erro.
* ***SAFE\_CAST()****:* Converte strings para outros tipos de dados, como INT64 (número inteiro) ou FLOAT64 (número com casas decimais), também com a segurança do SAFE para evitar falhas.
* ***NULLIF()****:* Garante que campos vazios ou com espaços em branco sejam convertidos para NULL, facilitando a análise.

***FROM****:* O código obtém os dados da sua **tabela externa** novos\_dados\_ons, que é a origem dos dados brutos.

***WHERE****:* Esta cláusula de filtro descarta registros que não têm valores importantes (IS NOT NULL), como data, nome do reservatório e código de planejamento, garantindo que apenas dados de alta qualidade sejam processados.

***SELECT \* FROM dados\_tratados****:* Esta linha seleciona todas as colunas (\*) do resultado da sua consulta temporária (dados\_tratados).

**Ação:** O comando envia esses dados limpos para a tabela de destino.

* **Na criação da tabela (CREATE TABLE... AS SELECT...)**, essa consulta popula a nova tabela com os dados.
* **Na inserção incremental (INSERT INTO... AS SELECT...)**, ela envia apenas os novos registros para a tabela já existente.

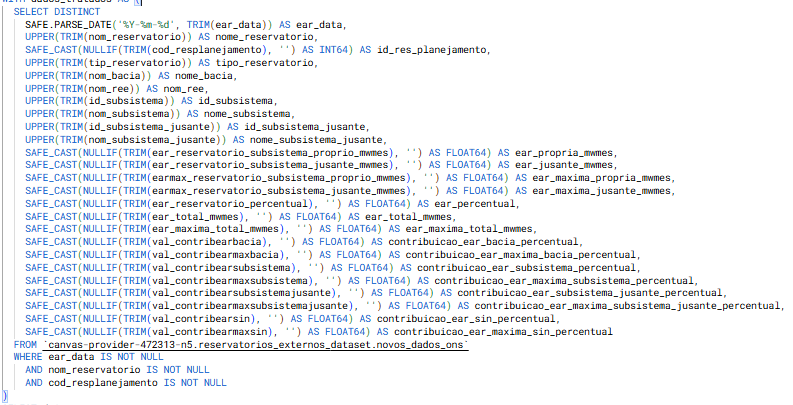


Este bloco de código é a lógica de inserção incremental do seu procedimento. Ele é executado apenas quando a tabela **trusted\_data** já foi criada.

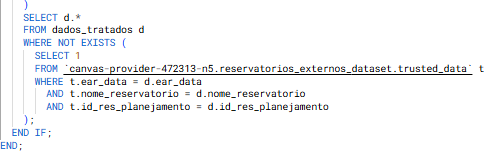
***ELSE****:* Esta cláusula indica que o código a seguir será executado se a condição inicial (IF NOT tabela\_criada) for FALSE, ou seja, se a tabela trusted\_data já existir.

***INSERT INTO ...****:* Este comando é usado para adicionar novos registros a uma tabela já existente. Ele direciona os dados para a tabela trusted\_data.

***WITH dados\_tratados AS (...)****:* A mesma CTE de limpeza e transformação é usada aqui. Ela processa os novos arquivos na tabela externa (novos\_dados\_ons) da mesma forma que na criação inicial.



***WHERE NOT EXISTS (...)****:* Esta é a parte mais importante da inserção incremental. O NOT EXISTS atua como um filtro, comparando os dados transformados (dados\_tratados) com os dados que já estão na tabela de destino (trusted\_data). A consulta só insere as linhas dos dados\_tratados que **não** possuem correspondência com os registros já existentes na tabela, usando a combinação de ear\_data, nome\_reservatorio e id\_res\_planejamento para identificar registros únicos.



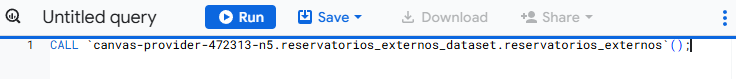
**SELECT d.\* FROM dados\_tratados d**: Esta parte seleciona todos os dados limpos e transformados que estão prontos para serem inseridos (d se refere a esses novos dados).

**WHERE NOT EXISTS (...)**: Esta é a cláusula principal. Ela faz uma checagem, linha por linha, nos seus novos dados. Para cada nova linha, ela pergunta: "Já existe uma linha na tabela de destino com a mesma combinação de data, nome do reservatório e ID de planejamento?"

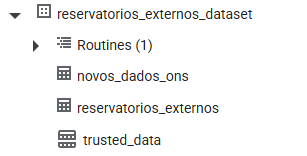
**A Subconsulta (SELECT 1 FROM ... t)**: Esta consulta interna é o que faz a verificação. Ela busca uma correspondência na sua tabela de destino (t). A combinação de campos na cláusula WHERE (***ear\_data, nome\_reservatorio, id\_res\_planejamento***) é o que define um registro como único.

Após inserir todo o comando aperto em **Run** para rodar o comando.

O próximo comando é o ponto de partida do seu agendamento no BigQuery. Ele é responsável por iniciar a execução do procedimento que faz toda a transformação dos dados e aperte **Run** novamente

****

Depois de rodar o CALL vai ser criada a tabela trusted\_data contendo os dados transformados.

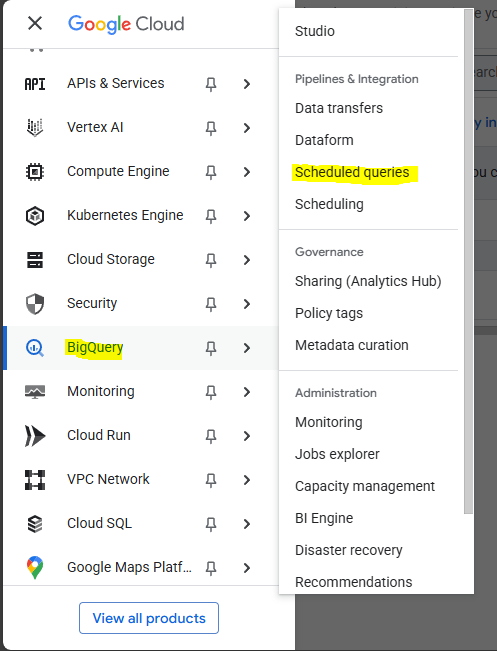


**3. Automação do Processo (Scheduler)**

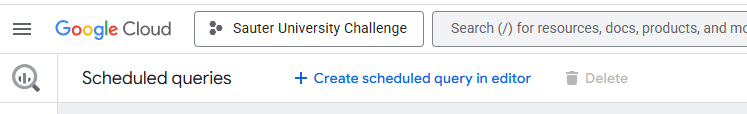
A etapa final é a automação de todo o processo de transformação de dados. Em vez de rodar o código manualmente, foi criado um agendamento (**scheduler**) que executa o procedimento armazenado em uma base diária.

O agendamento garante que a tabela de dados seja atualizada automaticamente, sem intervenção manual, a cada nova entrada de dados no bucket, tornando o pipeline de dados totalmente automatizado.

No menu lateral esquerdo procure por **BigQuery** -> **Scheduled queries**

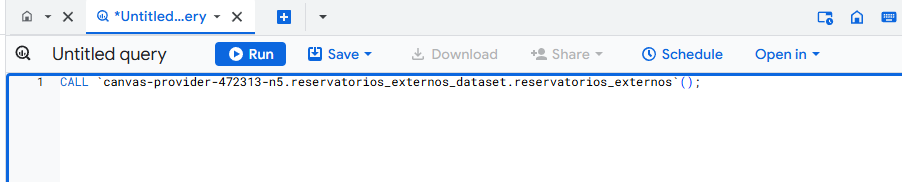


Clique em **+ Create scheduled query in editor** para abrir uma nova consulta em branco.

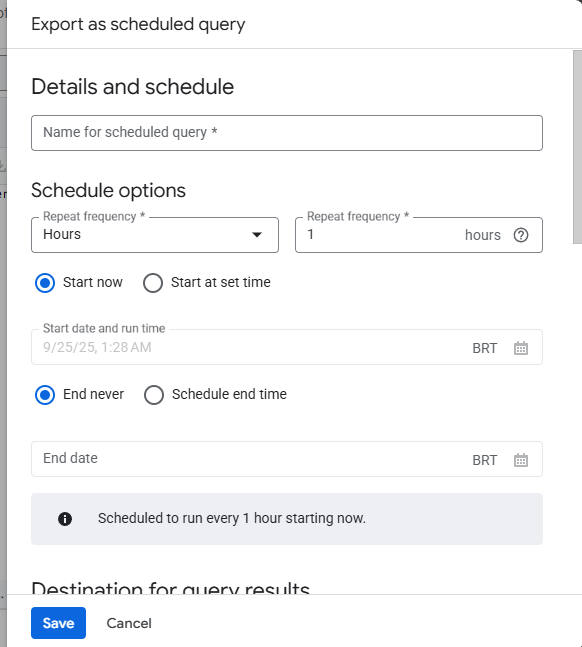
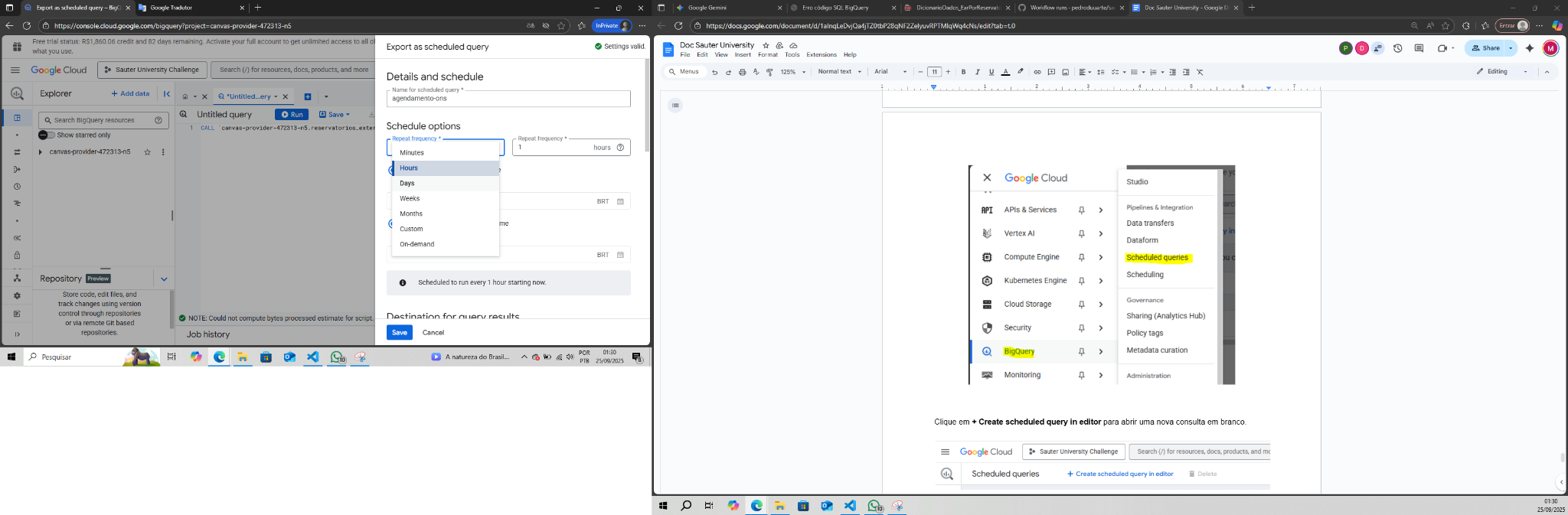


Após abrir a nova consulta em branco, insira o comando CALL no editor. Este comando é crucial, pois ele serve como a instrução do BigQuery para chamar e executar o procedimento armazenado que já existe no seu banco de dados.

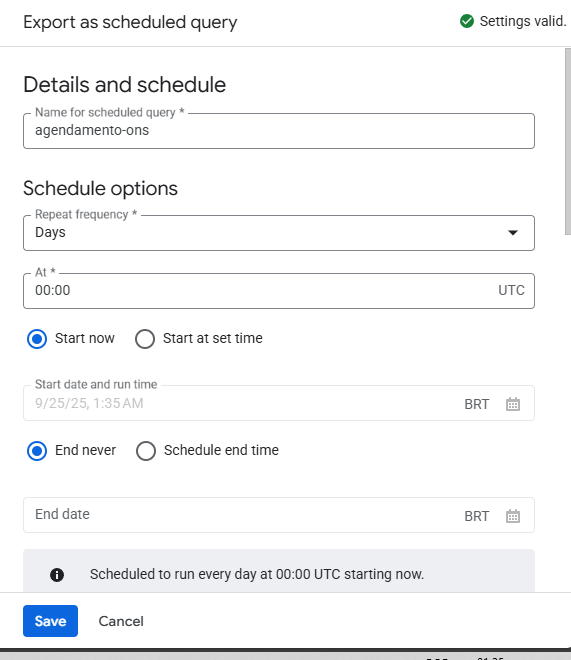
Depois de inserir o comando, clique em **Schedule** para continuar o processo de agendamento.



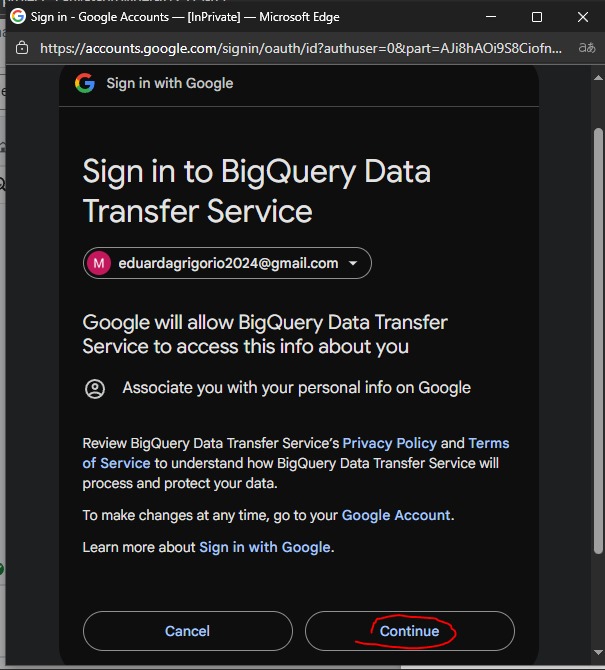
Agora, você verá a tela de configuração do agendamento. Nela, você precisa preencher os campos com os detalhes de nome e frequência de execução.



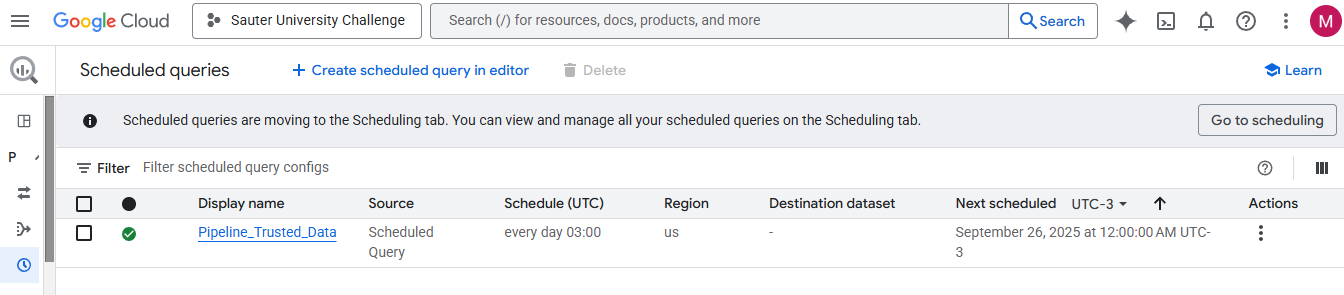
Primeiro, defina um nome para o agendamento. Na seção de frequência, é possível escolher o intervalo da atualização (horas, dias ou meses) e a quantidade de repetições.



Após preencher os campos, você será solicitado a definir o horário de execução, dependendo da frequência escolhida. Para finalizar, clique em Save.



Será solicitada uma permissão. Clique em **Continue** para autorizar e criar o agendamento.



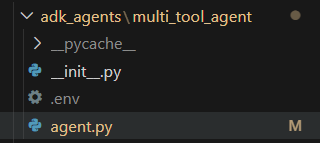
Para verificar se o agendamento foi criado, navegue até o menu lateral esquerdo e selecione **BigQuery > Scheduled queries**. Você verá seu agendamento listado na tela com todos os detalhes que foram inseridos.

**Chatbox com ADK**

O objetivo deste projeto foi criar **agentes multifuncionais**, cada um responsável por executar suas respectivas tarefas, e um **agente orquestrador**, responsável por direcionar as solicitações recebidas para o agente mais adequado.  
 Com isso, o sistema é capaz de responder perguntas sobre o site da empresa **Sauter Digital** e também realizar consultas em dados armazenados no **Google BigQuery**.

* **Arquitetura do Projeto**

O projeto deve ter a seguinte divisão de pastas:



Na pasta **agent.py**, está toda a implementação dos agentes que serão usados, sendo eles:

**Agente: sauter\_info\_agent**Responsável por buscar informações relacionadas ao site da empresa Sauter Digital, utilizando a ferramenta google\_search com filtro de domínio.

**Agente: bigquery\_agent** Especialista em análise de dados. Ele formula e executa queries SQL no **BigQuery**, consultando o dataset definido (reservatorios\_externos\_dataset & tabela trusted\_data).

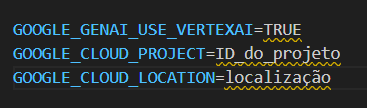
**Agente Orquestrador: HelpDeskCoordinator** Atua como o roteador principal. Recebe as solicitações dos usuários, interpreta o contexto e escolhe qual agente deve responder.

Será falado mais sobre cada um deles posteriormente.

A pasta **\_\_init\_\_.py** deve conter apenas o import



É a pasta **.env**, as credenciais



**OBS:** Jamais suba suas credenciais no github ou em qualquer local de gestão de versionamento ou repositório, coloque os arquivos responsáveis pelo armazenamento de credenciais em um **.gitignore** ou armazene apenas em um ambiente virtual, se preferir use outras ferramentas como o service account do gcp.

Imagem do **.gitignore**

****

* **Iniciando o desenvolvimento**

Este passo a passo vai guiar você desde a configuração do ambiente até a execução do agente orquestrador.

No terminal, execute:

**git clone <URL\_DO\_REPOSITORIO>**

**cd <NOME\_DO\_REPOSITORIO>**

**git checkout develop/adk-setup**

Substitua **<URL\_DO\_REPOSITORIO>** pela **URL** do seu repositório e **<NOME\_DO\_REPOSITORIO>** pelo nome da pasta criada.

É recomendado criar um **ambiente virtual** para isolar as dependências do projeto:

**python -m venv venv**

**Ative o ambiente:**

* **Windows (PowerShell):**

**.\venv\Scripts\Activate**

* **Linux / MacOS:**

**source venv/bin/activate**

Com o ambiente virtual ativo, instale todas as dependências:

**pip install google-adk**

**pip install google-adk**

**Credenciais de ambiente**

O projeto utiliza variáveis de ambiente para definir **project ID do GCP** e **dataset do BigQuery**:

1. Crie um arquivo **.env** na raiz do projeto:

**GOOGLE\_GENAI\_USE\_VERTEXAI=TRUE**

**GOOGLE\_CLOUD\_PROJECT=ID\_DO\_PROJETO**

**GOOGLE\_CLOUD\_LOCATION=LOCALIZACAO**

1. **Importante:** Não suba este arquivo no GitHub. Adicione .env no .gitignore.
2. Se você estiver usando **Service Account**, exporte a chave JSON:

**export GOOGLE\_APPLICATION\_CREDENTIALS="caminho/para/sua-chave.json"**

No Windows (PowerShell):

**$env:GOOGLE\_APPLICATION\_CREDENTIALS="caminho\para\sua-chave.json"**

Após essas etapas, vamos criar o nosso primeiro agente:

* **Criando o agente de buscas do site da Sauter**

Primeiramente, faremos os imports na pasta **agent.py**.

from google.adk.agents import Agent

from google.adk.tools import google\_search

Em seguida, construiremos o agente da seguinte forma

**# Agente para responder perguntas sobre o site Sauter Digital**

**sauter\_agent = Agent(**

**name="sauter\_info\_agent",**

**model="gemini-2.5-flash",**

**description=(**

**"Agente para responder perguntas sobre o site Sauter Digital."**

**),**

**instruction=(**

**"""**

**Você é um agente prestativo que responde perguntas sobre a empresa Sauter Digital.**

**Use a ferramenta google\_search para encontrar informações no site da empresa, garantindo que toda pesquisa inclua o filtro 'site:https://sauter.digital/' para que os resultados sejam apenas desse domínio."**

**"""**

**),**

**tools=[google\_search],**

**)**

O Agent é um **sauter\_agent** que tem o nome: **sauter\_info\_agent**

**A seu modelo é o gemini 2.5 flash**

Ele recebe uma pequena descrição e um prompt (instructions)

Esse prompt é o que define o comportamento e a capacidade de raciocinio do agente, nesse caso passamos o prompt:  
  
**Você é um agente prestativo que responde perguntas sobre a empresa Sauter Digital.  
Use a ferramenta google\_search para encontrar informações no site da empresa, garantindo que toda pesquisa inclua o filtro 'site:https://sauter.digital/' para que os resultados sejam apenas desse domínio."**

E o **tools=[...]** ele recebe as ferramentas que irá utilizar, nesse caso ele está usando o google search, que fas pesquisas na web.

Com isso o nosso agente estará pronto, em breve mostrarei o fluxo e os agentes em funcionamento.

* **Desenvolvendo o agente que lerá os dados no BQ(BigQuery)**

Antes de tudo, precisamos importar mais algumas tools que serão utilizadas pelo agente de busca do BQ

import os

from google.adk.agents import Agent

from google.adk.tools import google\_search

from google.adk.tools.bigquery import BigQueryToolset, BigQueryCredentialsConfig

from google.adk.tools.bigquery.config import BigQueryToolConfig, WriteMode

import google.auth

Em seguida, configuramos o acesso ao BigQuery:

application\_default\_credentials, \_ = google.auth.default()

credentials\_config = BigQueryCredentialsConfig(

credentials=application\_default\_credentials

)

PROJECT = os.getenv("GOOGLE\_CLOUD\_PROJECT")

DATASET = os.getenv("BIGQUERY\_DATASET")

# configurção do BigQueryToolset

tool\_config = BigQueryToolConfig(write\_mode=WriteMode.BLOCKED)

bigquery\_toolset = BigQueryToolset(

credentials\_config=credentials\_config, bigquery\_tool\_config=tool\_config

)

Com isso as nossas credenciais estarão configuradas

Agora criaremos o agente  
  
# Agente para consultar dados no BigQuery

bigquery\_agent = Agent(

name="bigquery\_agent",

model="gemini-2.5-flash",

description=(

"Agente para buscar e responder perguntas usando dados do Google BigQuery."

),

instruction=(

f"""

Você é um analista de dados especializado em consultar dados do Google BigQuery.

Ao receber uma solicitação, identifique a necessidade do usuário, formule e execute queries SQL no BigQuery usando a ferramenta disponível.

Utilizeo projeto onde o ID é '{PROJECT}' o dataset '{DATASET}' para todas as consultas.

A tabela que será utilizada é a 'trusted\_data'.

Retorne os resultados de forma clara e objetiva, explicando os dados quando necessário.

Se a consulta for complexa, explique os passos e forneça insights relevantes com base nos dados retornados.

Caso não tenha informações suficientes para responder, peça detalhes adicionais ao usuário.

"""

),

tools=[bigquery\_toolset],

)

Explicando:

* **name="bigquery\_agent"** → Nome do agente.
* **model="gemini-2.5-flash" →** Modelo LLM utilizado.
* **description** → Resume a função: consultas ao BigQuery.
* **instruction** → Prompt que define o raciocínio e comportamento.
* **tools=[bigquery\_toolset]** → O agente pode executar queries SQL diretamente no BigQuery.

Esse agente é ideal para cenários de análise de dados, relatórios e perguntas que envolvem banco de dados.

Com isso o nosso agente de buscas no banco de dados está implementado

* **Criando o agente orquestrador**

Esse será responsável por direcionar a requisição do usuário ao agente mais adequado.

Imports necessários:

from google.adk.agents import LlmAgent

from google.adk.tools import agent\_tool

Criação do agente:

# Agente orquestrador

coordinator = LlmAgent(

name="HelpDeskCoordinator",

model="gemini-2.5-flash",

instruction=

"""

Você é o agente orquestrador principal do Help Desk.

Analise cuidadosamente cada solicitação do usuário e encaminhe para o agente mais adequado:

- Use o agente 'sauter\_info\_agent' para perguntas sobre a empresa Sauter Digital ou informações do site sauter.digital.

- Use o agente 'bigquery\_agent' para solicitações que envolvam dados, relatórios ou consultas ao BigQuery.

Se não tiver certeza de qual agente usar, peça esclarecimentos ao usuário antes de encaminhar.

Sempre busque oferecer a melhor experiência, direcionando a solicitação ao agente mais especializado.

""",

description="Main help desk router.",

tools=[agent\_tool.AgentTool(agent=sauter\_agent),

agent\_tool.AgentTool(agent=bigquery\_agent)],

)

root\_agent = coordinator

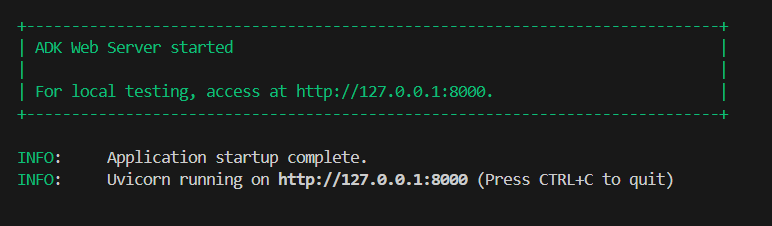
Explicando:

* **name="HelpDeskCoordinator"** → Nome do agente orquestrador.
* **model="gemini-2.5-flash"** → Modelo LLM usado para interpretar as requisições.
* **instruction** → Prompt que ensina como o orquestrador deve rotear as solicitações.
* **tools=[...]** → Ele transforma outros agentes em ferramentas (AgentTool) que podem ser chamadas dinamicamente.
* **root\_agent = coordinator** → Define o orquestrador como o ponto de entrada principal do sistema.

Assim, o usuário interage **somente** com o HelpDeskCoordinator, que decide internamente se a pergunta deve ir para o **sauter\_agent** (buscas no site) ou para o **bigquery\_agent** (consultas no banco).

Agora vamos testar os bots criados

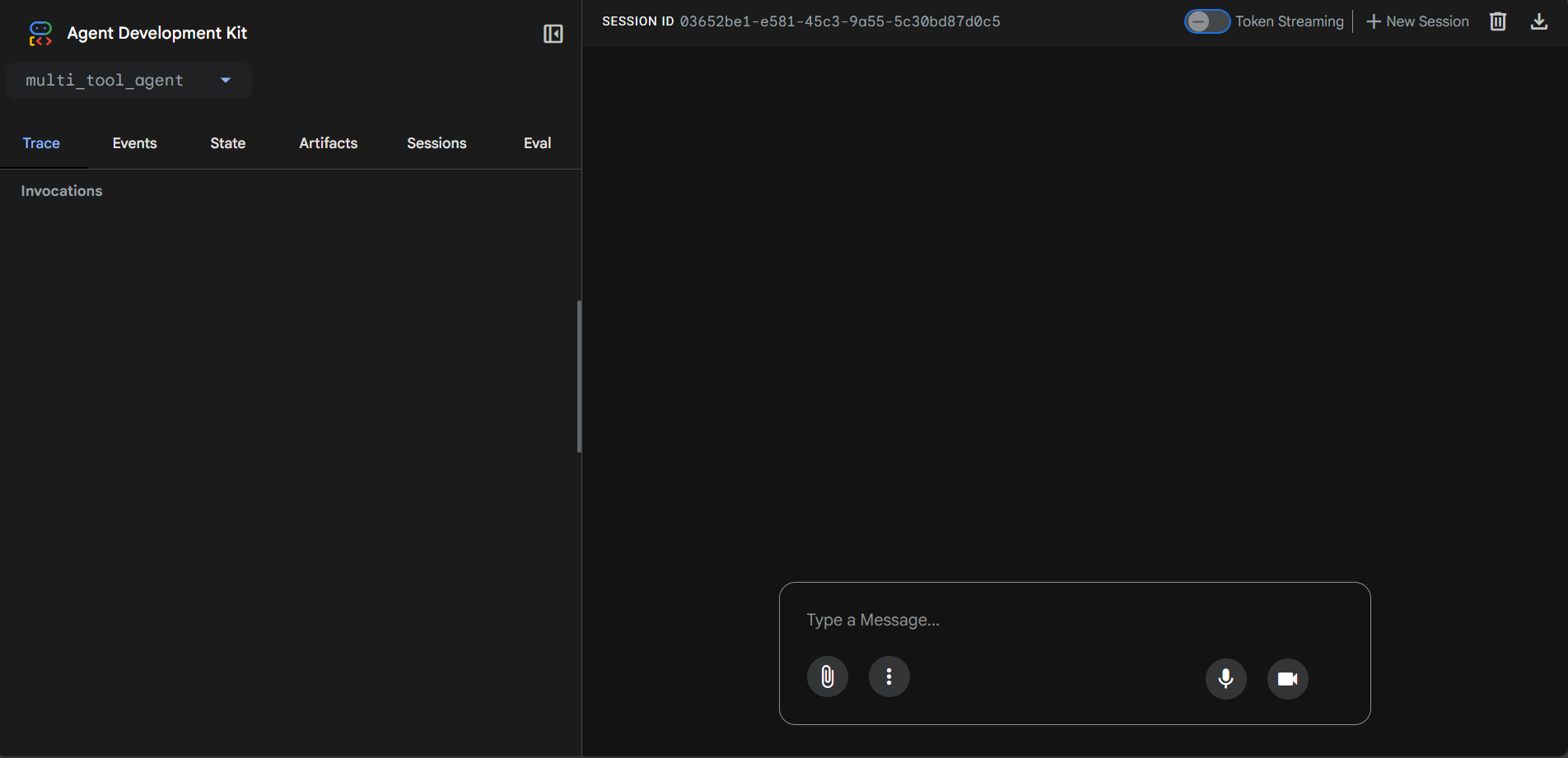
No terminal = coloque o comando **ADK WEB**

****

Algo mais ou menos assim aparecerá.

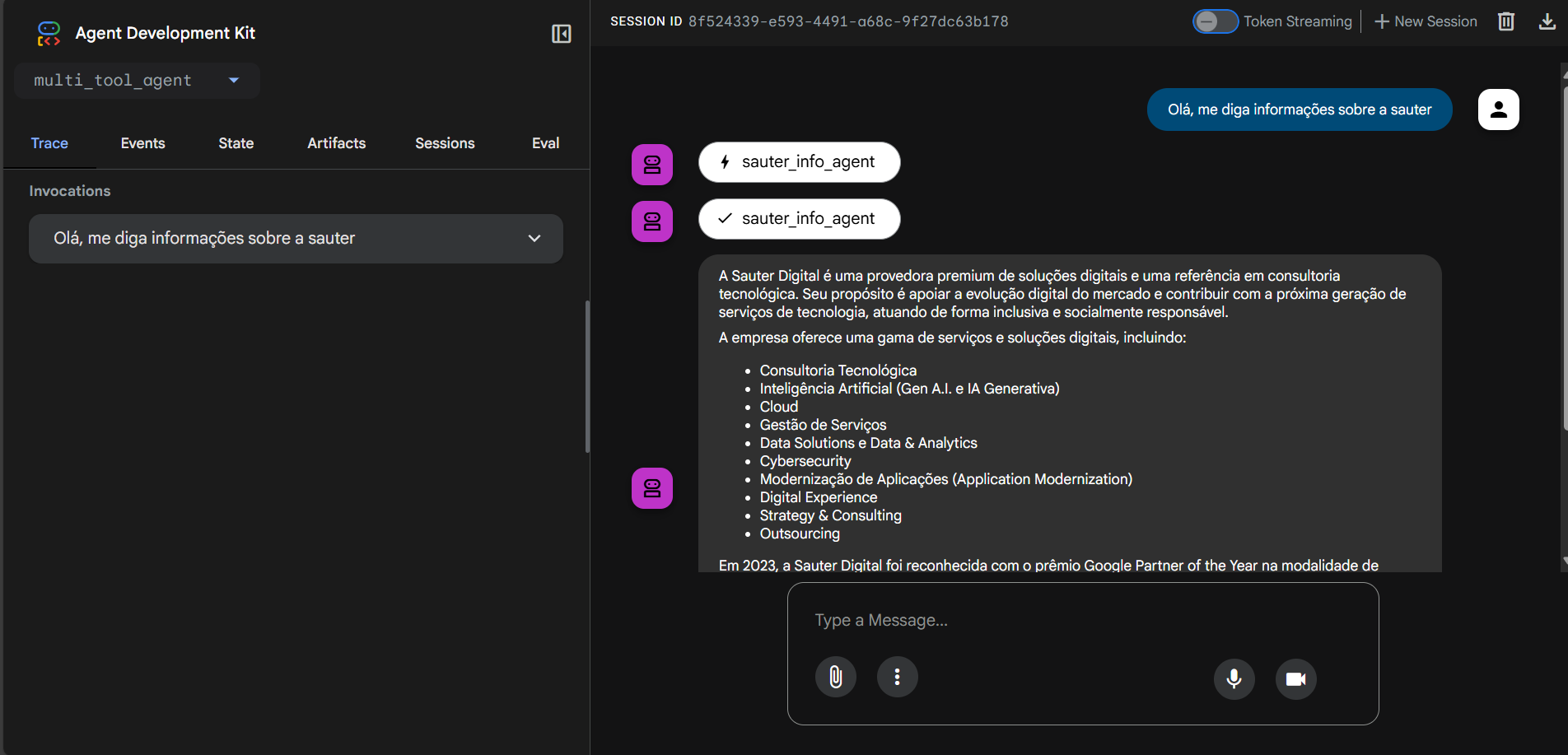
Clique no link: [http://xxxx.x.x.:xxxx](about:blank)

Isso abrirá a interface gráfica do agente



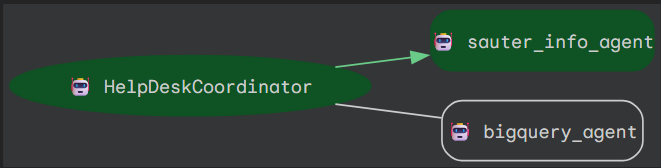
Vamos iniciar com um prompt simples,

Perguntaremos algo sobre a sauter.

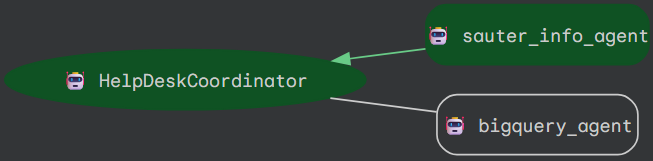


Aqui podemos ver a resposta do agente.

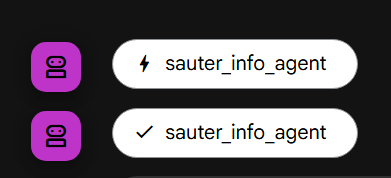
* **Fluxo de dados:**

****

Aqui podemos ver que o agente orquestrador manda manda a pergunta para o agente da sauter\_info.

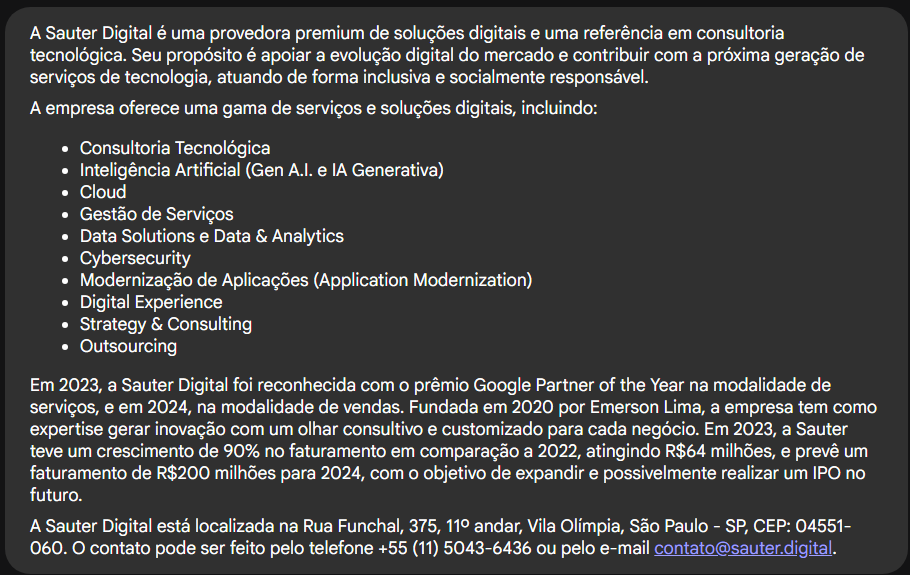


O agente da sauter\_info, por sua vez, retorna com a resposta para o agente orquestrador, que analisa e manda a resposta para o usuário.

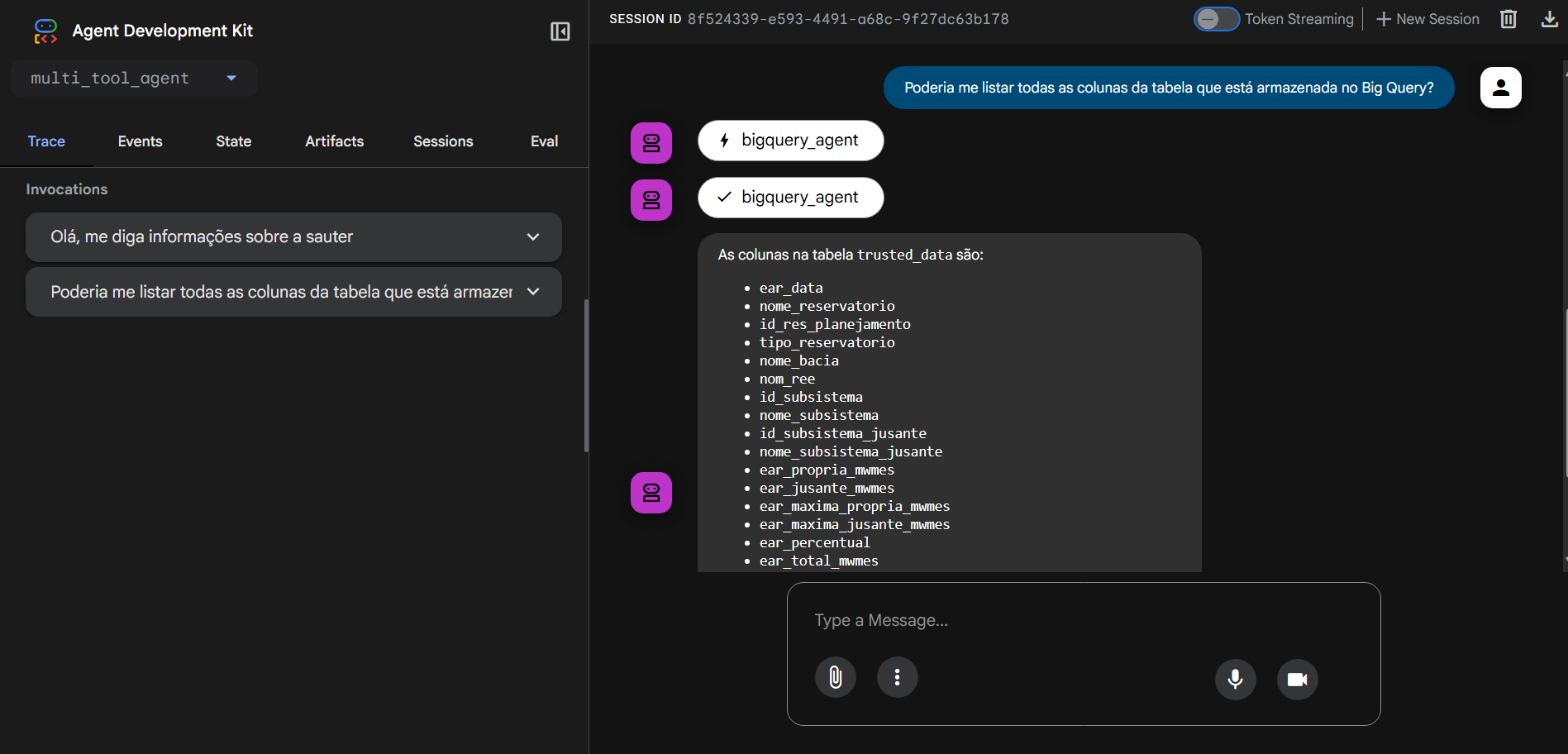


Essas funções são chamadas para passar a informação para o agente que irá respondelas

Resposta:

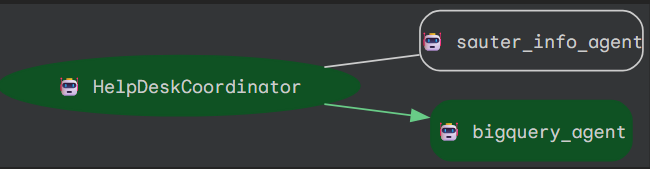


Então, vamos testar outra pergunta, dessa vez no agente de consulta do banco de dados:

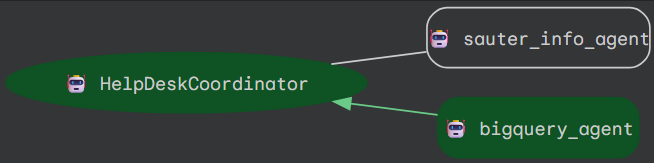


Como podemos ver, segue a mesma estrutura, chamando a função de transferencia de agente, dessa vez, para o agente de buscas no BQ.

O fluxo segue o mesmo modelo:



O agente orquestrador manda a pergunta para o agente responsável, nesse caso, o bigquery\_agent.



O agente bigquery retorna com a resposta para ser avaliada pelo agente orquestrador, caso esteja tudo ok, a resposta é enviada ao usuário.

Dessa forma o agente orquestrador serve tanto para delegar as perguntas para cada agente específico, como, também, supervisionar as respostas dos agentes e garantir uma melhor assertividade.