Desafio SAUTER University 2025

| Controle de revisões | | |
| --- | --- | --- |
| Pessoas Revisor | Menus suspensos Status | Nenhum tipo Notas |
| Jamille | Não iniciada |  |
| João | Aprovada |  |
| Jorge | Aprovada |  |
| Júlia | Aprovada |  |
| Laryssa | Aprovada |  |
| Letícia | Não iniciada |  |

| Diretório da equipe | |
| --- | --- |
| Jamille | Engenheiro de Dados |
| João | Project Manager |
| Jorge | Infra/DevOps |
| Júlia | QA |
| Laryssa | Engenheiro de IA/Agentes |
| Letícia | Desenvolvedor Backend/APIde projetos |

## V0.1

# Do desafio

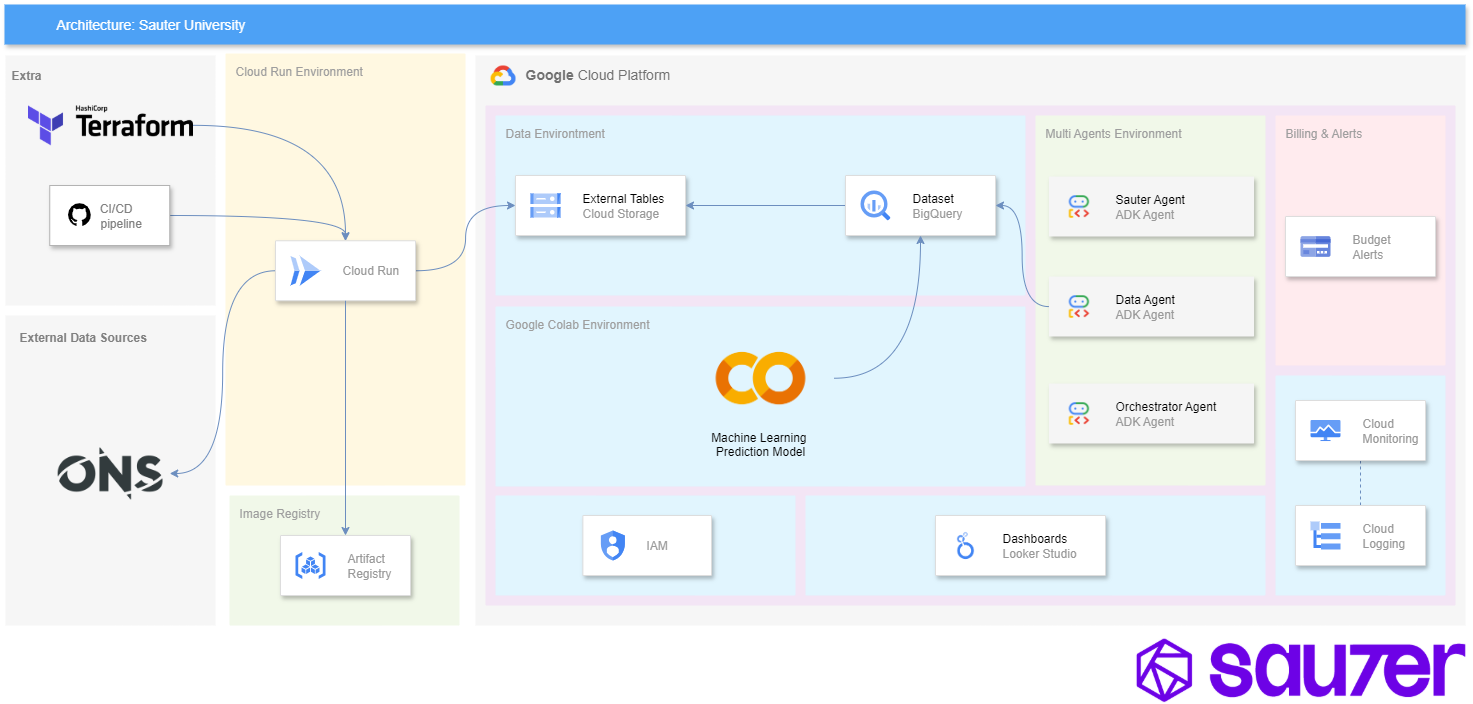


Figura 1.1: Arquitetura

Realizar a implementação vista na arquitetura acima;

Cada equipe se divida em grupos de 5 pessoas. Cada equipe precisará desenvolver o esquema apresentado na arquitetura, seguindo as boas práticas de engenharia de dados, de software e do Google Cloud. Cada equipe deverá realizar uma demonstração PRÁTICA sobre a sua solução, pontuando explicitamente cada ponto destacado abaixo:

* Pitch, “Why Google?” (apresentação teórica de no máximo 3~5 minutos)
* Integração com a ferramenta de CI/CD (github actions);
* Terraform utilizado para levantar a infraestrutura;
* Pipeline de transformação dos dados; REST API que buscará os dados para uma data específica ou um conjunto de dados históricos;
* Modelo preditivo que calcula o volume de água previsto para um reservatório (baseado no modelo de ENA)

<https://dados.ons.org.br/dataset/ear-diario-por-reservatorio>

OU apresentar a criação de um agente com o ADK + Gemini, com mecanismo de RAG, que consulta a base de dados HISTÓRICA de ENA e é capaz de responder dúvidas sobre o volume de uma bacia hidrográfica em um determinado período, o agente também deve responder dúvidas sobre a sauter, baseado nos dados do site oficial da sauter [http://sauter.digital](http://sauter.digital/).

* Exibir em uma representação gráfica uma análise sobre os dados tratados.

# Da divisão dos trabalhos:

* **Project Manager:** Garantir a coesão entre as frentes e garantir conformidade da documentação, além de montar o pitch e o dashboard.
* **Infra/DevOps:** Foco no Terraform, configuração do Budget Alert e pipeline de CI/CD no GitHub Actions.
* **Engenheiro de Dados:** Responsável pelo pipeline de ingestão e transformação dos dados do ONS para o BigQuery.
* **Engenheiro de IA/Agentes:** Foco no desenvolvimento dos agentes com ADK, Gemini.
* **Desenvolvedor Backend/API:** Responsável por criar a REST API em Python, sua conteinerização e a integração com os agentes.
* **QA:** Responsável por liderar a criação dos testes (unitários e de integração), garantir a cobertura de 85%.

Foi acordado em equipe a delegação das responsabilidades listadas acima, cabendo a cada papel a prioridade no desenvolvimento das tarefas apresentadas. As etapas de pesquisa e levantamento de requisitos para a solução final foi feita por brainstorming e reuniões casuais com partes da equipe.

# Da solução:

## 2.1. Github:

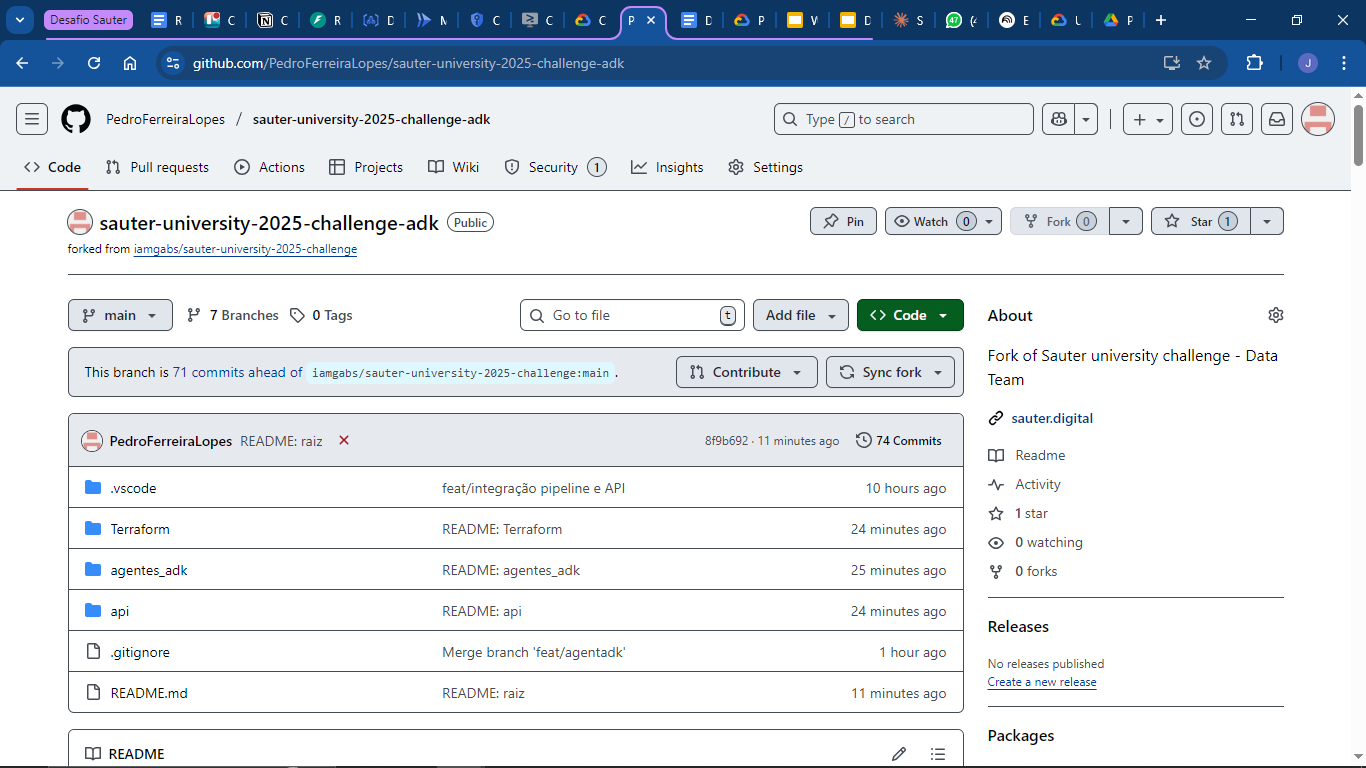


Figura 2.1.1: Main com a contribuição de toda a equipe

No github, foi guardado todo o código do projeto, com README.

A tentativa de implementação de CI/CD foi “instanciada” no ambiente do GCP.

Apesar das dificuldades, o trabalho de todos os integrantes foi levada à versão final de entrega, com alguns problemas de manipulação de dados via git e divergências de branches.

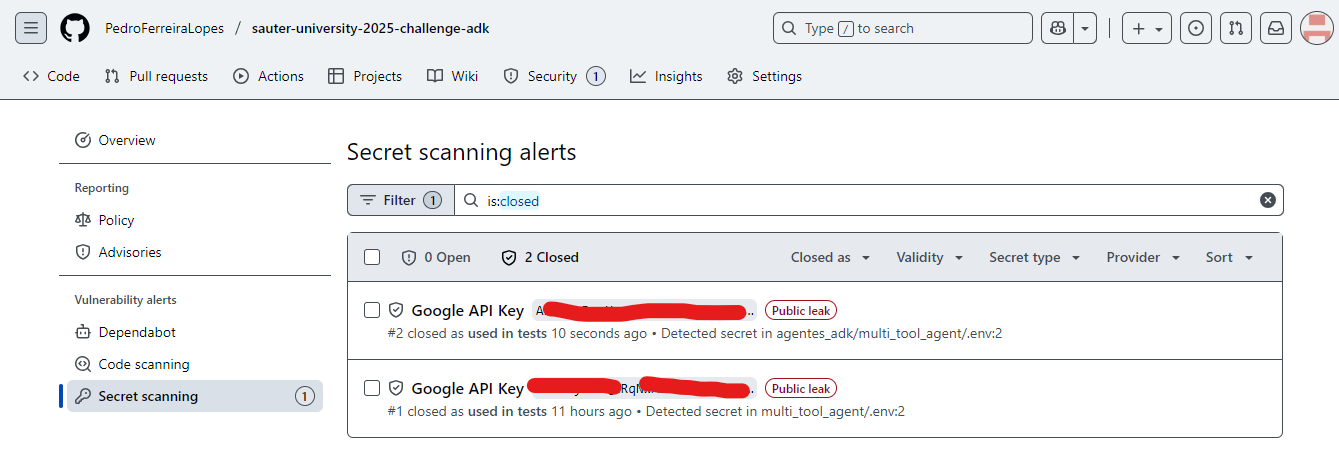


Figura 2.1.2: Vazamento de dados

Houve um vazamento de chave API na aplicação do ADK, que foi resolvida pouco tempo depois, alterando a chave de acesso.

## 2.2. Terraform:

Utilizamos Terraform para provisionar e gerenciar uma infraestrutura base para nosso Google Cloud Platform (GCP) com o objetivo inicial de criar um ambiente inicial contendo uma rede, uma máquina virtual e um sistema de alerta de custos, tudo de forma automatizada e modular, ou seja, uma forma mais prática de configurar GCP tanto no momento que foi feito quanto para futura mudanças.

Começamos nosso código Terraform configurando quais provedores são necessários para funcionar:

* required\_providers: Declara os provedores essenciais.
* google: Especifica que você usará o provedor do Google Cloud.
* source = "hashicorp/google": Indica de onde baixar o provedor.
* version = "4.51.0": Fixa a versão do provedor para garantir que o código funcione de forma consistente no futuro, evitando que atualizações automáticas causem erros.

Após isso configuramos o provedor do Google Cloud, fornecendo os detalhes de autenticação e localização:

* provider "google": Inicia o bloco de configuração do provedor Google.
* project, region, zone: Define o ID do projeto, a região e a zona padrão onde os recursos serão criados.
* credentials = file("credentials.json"): Aponta para o arquivo de credenciais (chave de conta de serviço) usado para autenticar o Terraform na sua conta GCP.

Em seguida utilizamos os recursos, que são componentes da infraestrutura que representam serviços do Google Cloud na qual queremos provisionar:

* resource "google\_compute\_network" "vpc\_network": Declara a criação de uma rede de nuvem privada virtual(VPC) para isolar recursos da nuvem.
* resource "google\_compute\_instance" "vm\_instance": Declara a criação de uma máquina virtual(VM).
* resource "google\_project\_service" "compute\_api": Declara a criação de um recurso do tipo "habilitação de serviço".

No recurso de "google\_compute\_instance":

* Definimos, respectivamente: o nome da VM, o tipo de máquina, seu tamanho e potência como e2-micro que é uma das menores e a zona onde ela será criada.
* boot\_disk: Configura o disco de inicialização.
* image = "debian-cloud/debian-11": Especifica a imagem do sistema operacional a ser usado (Debian 11).
* network\_interface: Configura a interface de rede da VM.
* access\_config {}: Bloco vazio que, por padrão, atribui um endereço de IP externo à VM, permitindo acesso pela internet.
* depends\_on: Garante que o Terraform crie a vpc\_network antes de tentar criar a vm\_instance, evitando erros de dependência.

O bloco de módulo utiliza um conjunto reutilizável de código Terraform, que serve para criar canais de notificação para receber alertas e para monitorar os gastos do projeto. Podemos ver um exemplo de todo código da main na Figura 1.

* module "notification\_channel": Declara o uso de um módulo.
* source = "./modules/notification\_channel": Indica que o código deste módulo está na pasta local.
* module "billing\_budget": Declara o uso do módulo de orçamento.
* notification\_channel\_ids=module.notification\_channel.channel\_ids: Conecta os dois módulos. Ele pega os IDs dos canais de notificação criados pelo módulo.



Imagem 2.2.1: Aquirvo [main.tf](http://main.tf) raiz

**O que são módulos?**

Módulos(modules) são utilizados para dividir e organizar os códigos de uma forma intuitiva, sendo assim em vez de um arquivo com centenas de linhas, agrupamos arquivos relacionados como billing\_budget e notification\_channel e cada módulo tem sua própria estrutura, geralmente com arquivos como variables.tf para definir os dados de entrada que ele aceita, main.tf com a lógica dos recursos e outputs.tf (para expor informações para fora). isso torna o projeto mais limpo e fácil de entender. Além da organização, as alterações podem ser feitas de formas centralizadas, apenas mudamos o valor de billing\_budget no seu arquivo variables, ou pelo terminal. Podemos observar os módulos do nosso Terraform na foto a seguir:



Imagem 2.2.2: Módulos utilizados no projeto

O módulo notification\_channel tem um único objetivo: automatizar a criação de canais de notificação por e-mail no Google Cloud. O processo do código funciona em três etapas simples:

* Entrada: Ele recebe uma lista de e-mails como parâmetro. Se nenhuma lista for fornecida, ele utiliza uma lista padrão interna.
* Processamento: Usando um loop for\_each, ele cria um recurso de canal de notificação individual para cada e-mail da lista.
* Saída: Ao final, ele coleta os IDs de todos os canais que foram criados e os fornece como uma lista de saída, pronta para ser usada por outros módulos (como o de orçamento).

O objetivo do módulo billing\_budget é criar e configurar um Orçamento de Faturamento (Billing Budget) para o nosso projeto do Google Cloud, definindo alertas automáticos para monitorar os gastos. Assim é configurado com quatro informações principais: o ID da conta de faturamento , o ID do projeto , o valor do orçamento , e a lista de IDs dos canais de notificação que devem receber os alertas, desta forma cria um único recurso google\_billing\_budget que:

* Associa o orçamento a um projeto específico.
* Define o valor do orçamento em Reais (BRL).
* Configura três regras de alerta para notificar quando os gastos atingirem 50%, 80% e 100% do valor.
* Direciona todos os alertas para os canais de notificação recebidos na entrada, desativando os alertas padrão do Google.

Para garantir a segurança do projeto, utilizamos o arquivo .gitignore para prevenir o envio acidental de arquivos sensíveis para o repositório. Isso inclui:

* Chaves de serviço (.json, .p12) - credenciais de API
* Arquivos de estado do Terraform (.tfstate) - contêm dados sensíveis da infraestrutura
* Variáveis de ambiente (.env) - configurações privadas
* Logs e arquivos temporários

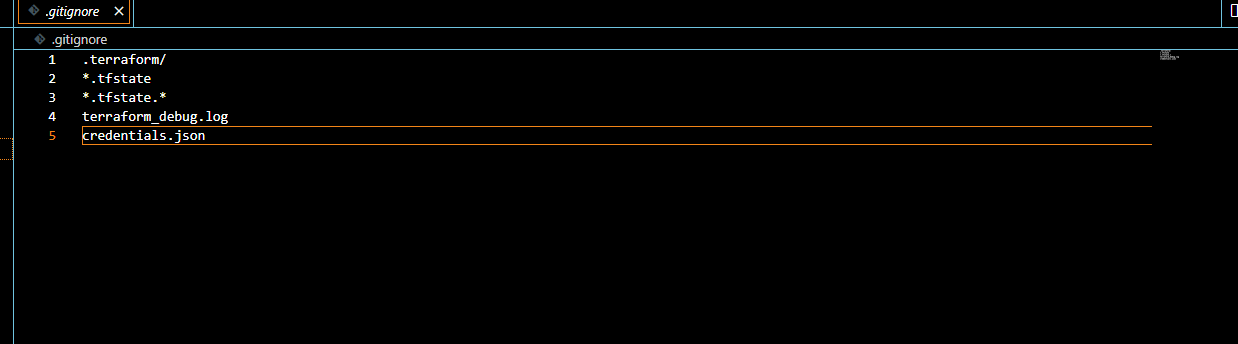


Imagem 2.2.3: Demonstração do gitignore

Além disso, a autenticação no GCP é realizada através de uma Conta de Serviço (Service Account) dedicada, criada exclusivamente para este projeto. O gerenciamento de acesso é controlado granularmente via Identity and Access Management (IAM).

Por fim, foram utilizados comandos no terminal do Google Cloud como:

* Terraform init
* Terraform plan
* Terraform appy

Pode-se observar um exemplo do terraform apply na imagem 4.

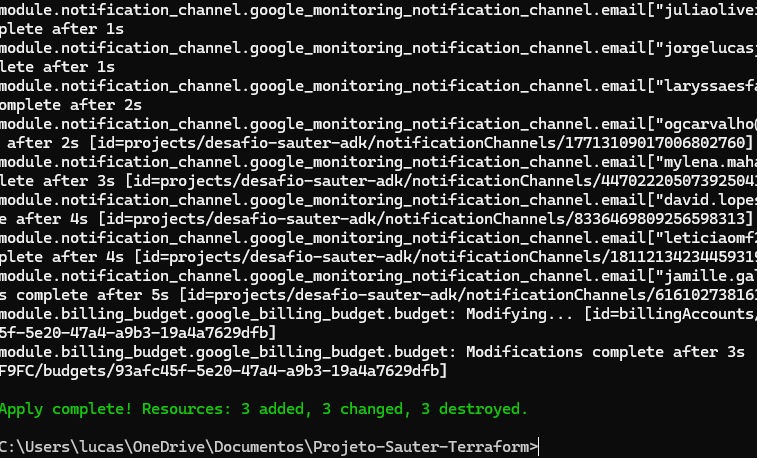


Imagem 2.2.4: Terraform apply realizado com sucesso

Ao finalizarmos essa última etapa, a nossa infraestrutura já se encontra configurada no GCP.

## 2.3. API REST:

A interface principal do projeto é uma API RESTful desenvolvida com o framework **FastAPI**. Esta escolha foi motivada pela sua alta performance, sintaxe moderna baseada em tipos Python e, crucialmente, pela sua capacidade de gerar documentação interativa (Swagger UI) de forma automática.

A API serve como o "cérebro" central, orquestrando as duas funcionalidades primárias do sistema:

1. **Orquestração da Pipeline de ETL:** Expõe um endpoint seguro para acionar, sob demanda, o processo de Extração, Transformação e Carga dos dados.
2. **Serviço de Dados:** Fornece endpoints para consultar os dados já processados e armazenados no Google BigQuery.

O código-fonte da API reside no diretório src/app e segue uma estrutura modular para garantir a manutenibilidade e escalabilidade:

* **main.py:** Ponto de entrada da aplicação. Responsável por inicializar o objeto FastAPI e registar os módulos de rotas (routers).
* **/routers:** Cada ficheiro nesta pasta define um conjunto de endpoints relacionados a uma funcionalidade específica (ex: data.py para consulta de dados, pipeline.py para o acionamento do ETL).
* **/services:** Contém a lógica de negócio desacoplada dos endpoints. Por exemplo, bigquery\_service.py encapsula toda a comunicação com o Google BigQuery.
* **/utils:** Módulos de utilidade, como a configuração de logs, que são partilhados por toda a aplicação.

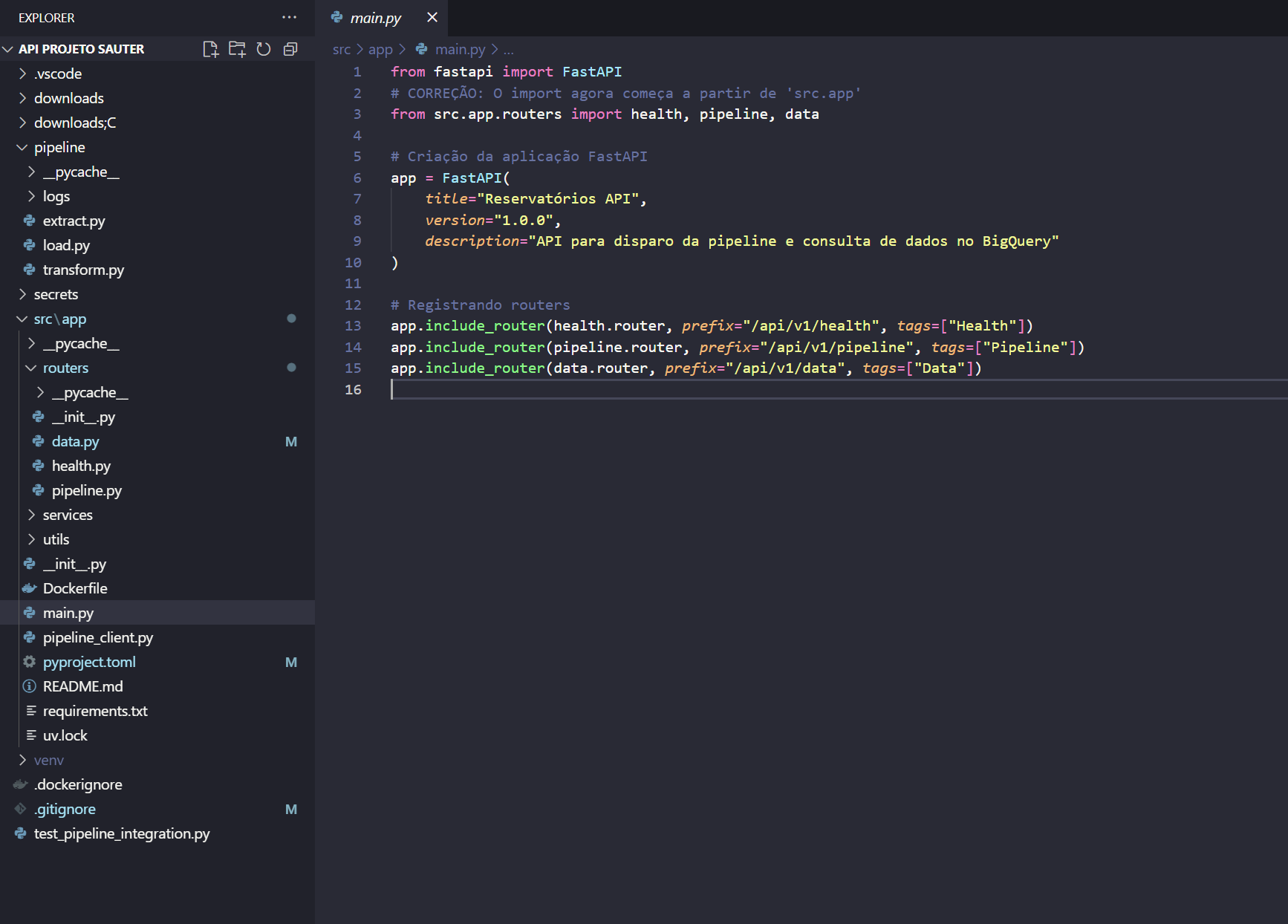


Figura 2.3.1 – Estrutura de diretórios e ponto de entrada da API

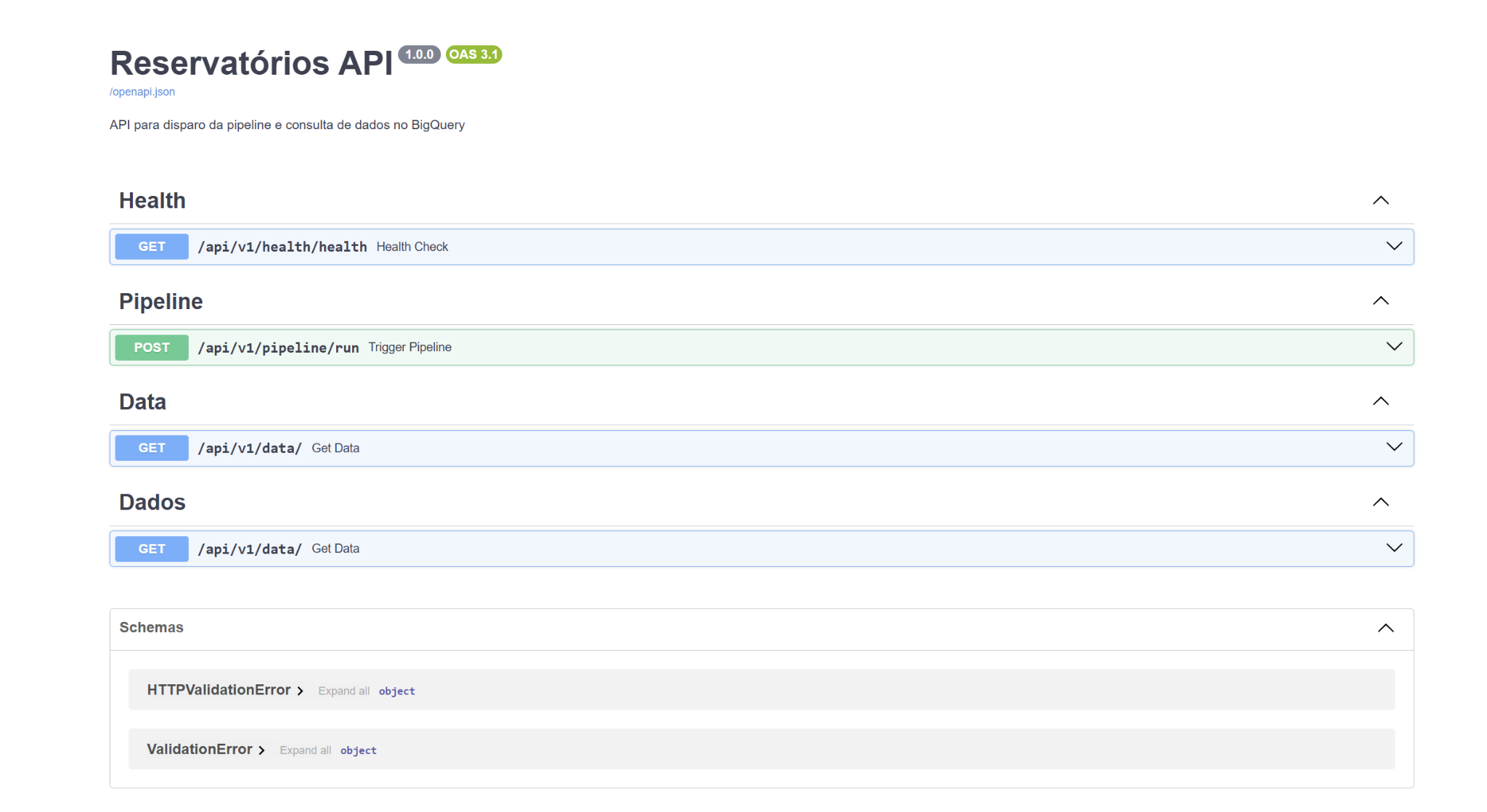


Figura 2.3.2 – Documentação automática da API gerada pelo FastAPI

## **2.3.1 Funcionalidades da API (Endpoints)**

A API expõe os seguintes endpoints principais, todos sob o prefixo /api/v1.

GET

/health/health

**Health Check:** Verifica se a API está no ar e a funcionar.

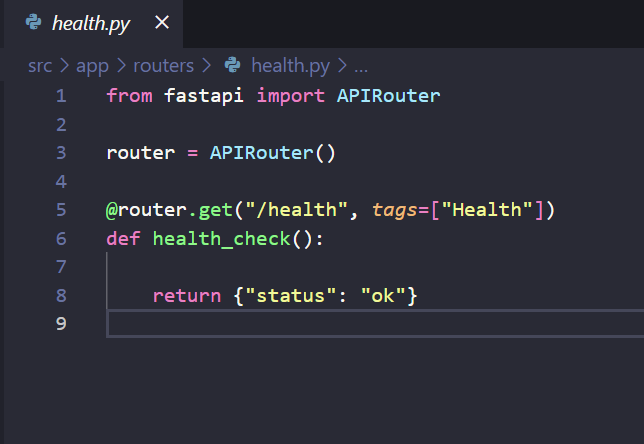


Figura 2.3.1.1 – Implementação do endpoint de verificação de estado (Health Check)

GET

/data

**Consultar Dados:** Busca dados processados no BigQuery, com opções de paginação e filtro.



Figura 2.3.1.2 – Implementação do endpoint de consulta de dados do BigQuery

POST

/pipeline/run

**Disparar Pipeline:** Aciona a execução da pipeline de ETL em segundo plano.



Figura 2.3.1.3 – Implementação do endpoint de acionamento da pipeline de ETL

## 2.3.2. Arquitetura da Solução:

Para garantir que o nosso projeto fosse robusto, portátil e fácil de manter, a nossa arquitetura foi construída sobre dois pilares tecnológicos centrais: uma API de alta performance como interface principal e a conteinerizaçãocom Docker para garantir a consistência do ambiente.

A API, desenvolvida com o moderno framework FastAPI, funciona como o "cérebro" do nosso sistema. É através dela que os dados processados são consultados e é também por ela que a nossa pipeline de ETL é acionada. O Docker, por sua vez, funciona como a "cápsula" que envolve toda a nossa aplicação — o código, as dependências e as configurações — numa unidade coesa e portátil.

Esta abordagem permite que qualquer pessoa execute o projeto inteiro, seja num ambiente de desenvolvimento local ou num servidor de produção, com a máxima simplicidade e consistência, eliminando os problemas de "na minha máquina funciona". As seções seguintes detalham a estrutura interna da nossa API e o nosso processo de conteinerização e análise de segurança.

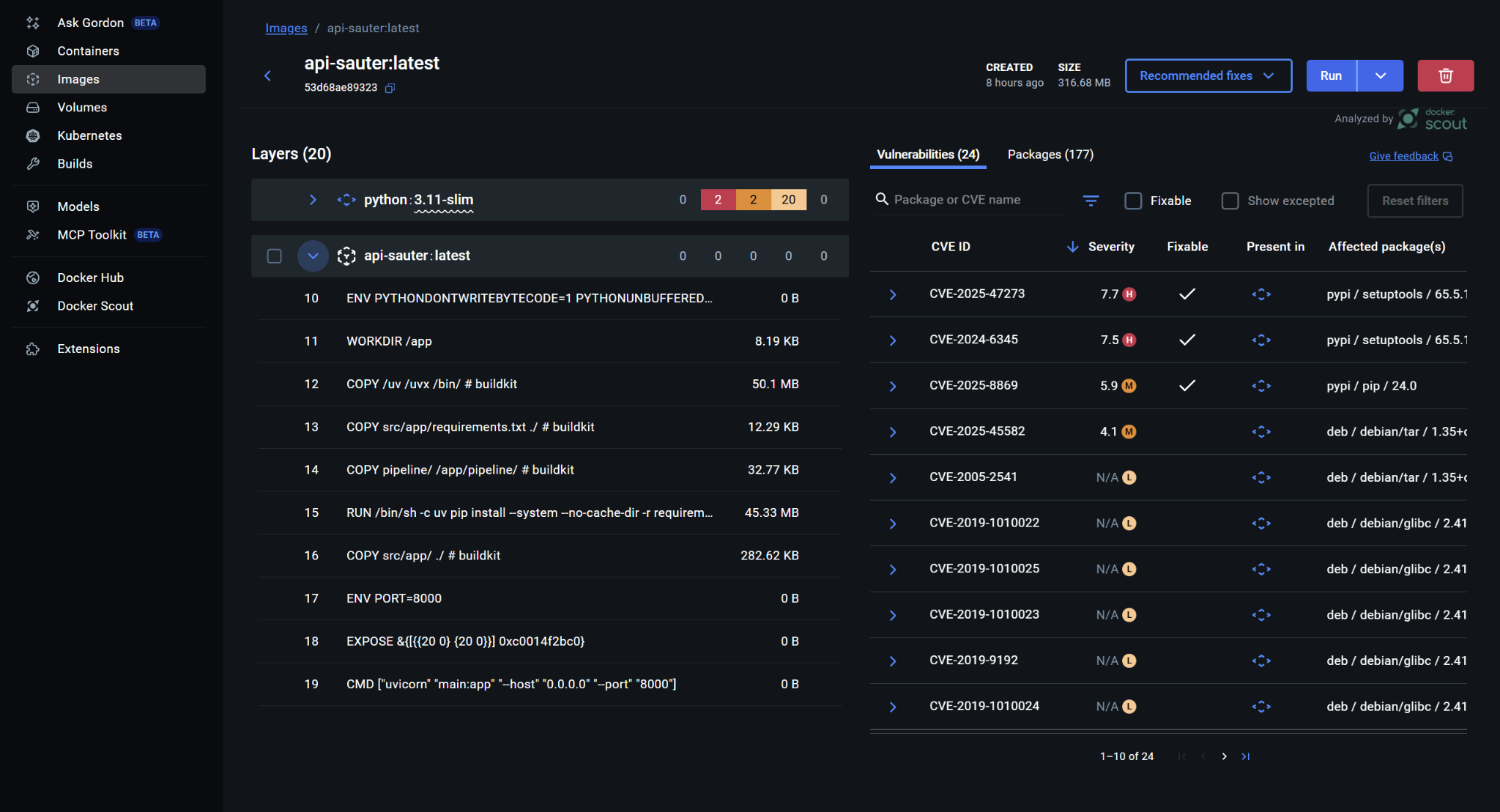


Figura 2.3.2.1 – Análise de vulnerabilidades da imagem Docker

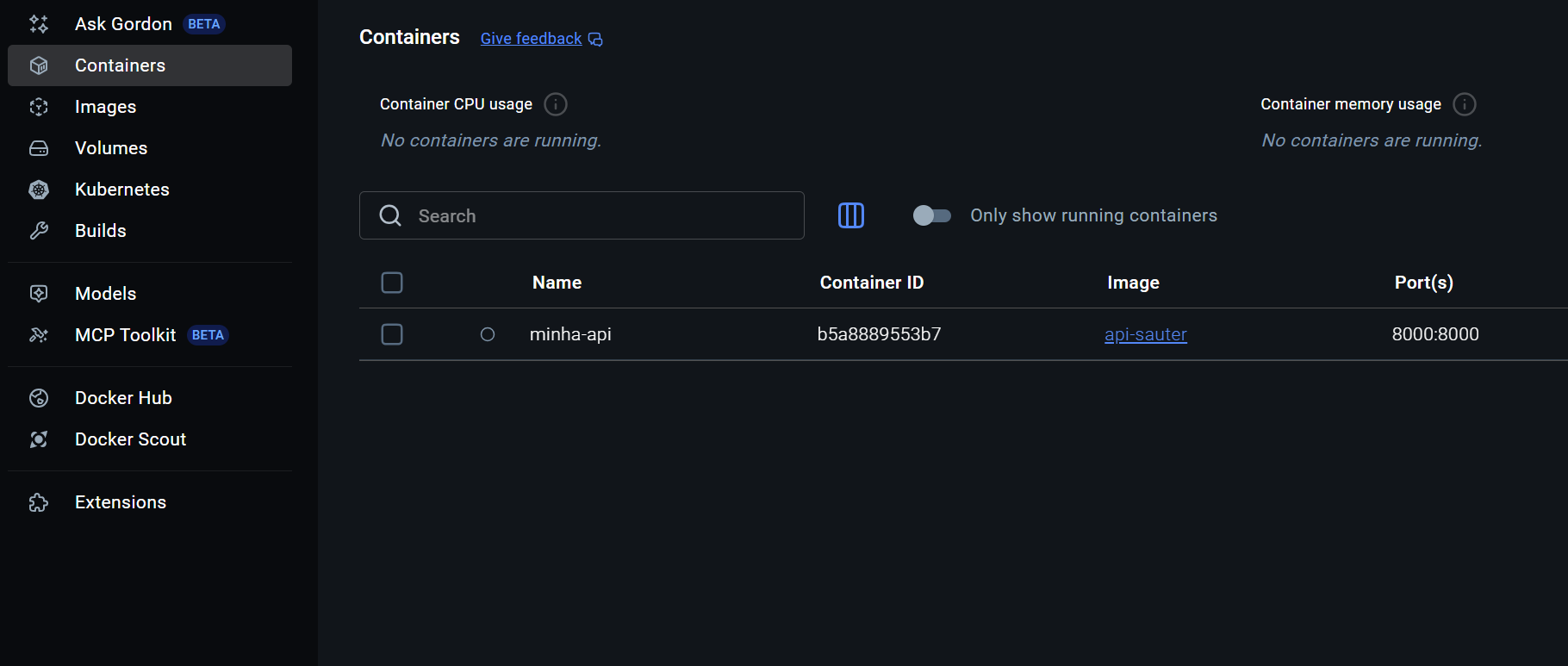


Figura 2.3.2.2 – Painel de gestão de contêineres do Docker Desktop

#### **O Dockerfile:**

O Dockerfile é a "receita" para construir a nossa imagem. Ele executa os seguintes passos:

1. **Imagem Base:** Utiliza python:3.11-slim como base, uma imagem leve e otimizada do Python sobre Debian.
2. **Instalação de Dependências:** Instala as bibliotecas Python a partir do ficheiro requirements.txt usando uv. Esta etapa é executada antes da cópia do código-fonte para otimizar o sistema de cache do Docker.
3. **Cópia do Código-Fonte:** Copia os diretórios src/app (API) e pipeline (ETL) para dentro do contêiner.
4. **Exposição da Porta:** Expõe a porta 8000, na qual o servidor Uvicorn irá correr.
5. **Comando de Execução:** Define o comando uvicorn src.app.main:app para ser executado quando o contêiner arranca.

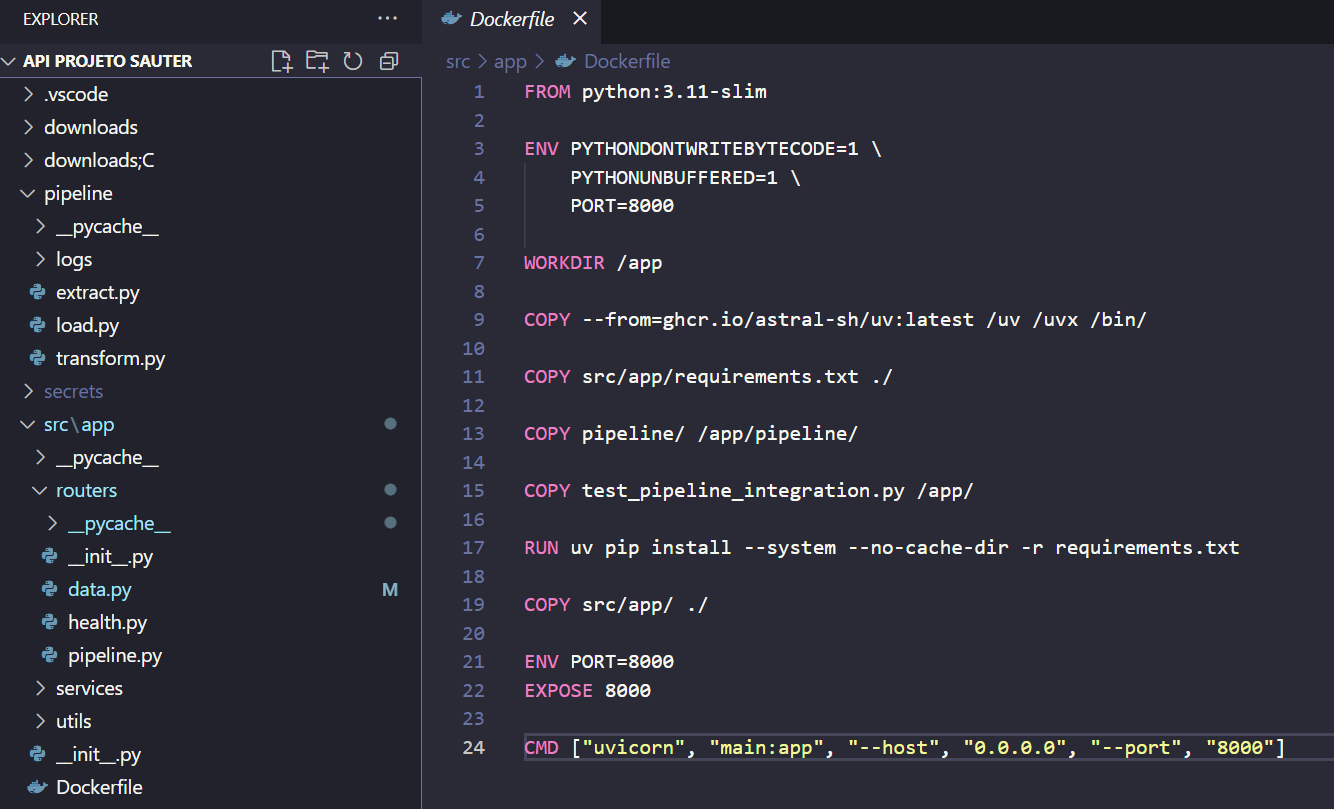


Figura 2.3.2.3 – Definição do ambiente de conteinerização com Dockerfile

#### **Execução**

**Build da Imagem:** docker build -f src/app/Dockerfile -t api-sauter

**Execução do Contêiner da API:** docker run -d -p 8000:8000 --name minha-api api-sauter

## 2.4. Pipeline de Dados :

O pipeline de dados do projeto foi desenvolvido para extrair dados da API pública da ONS, realizar transformações para limpeza e normalização, e carregar os dados processados no Google BigQuery. Todas as etapas foram implementadas garantindo rastreabilidade, reprodutibilidade e manutenção da qualidade dos dados.

O pipeline atua como o "cérebro" central do processamento de dados do projeto, automatizando a coleta, transformação e armazenamento, permitindo que análises e consultas sejam realizadas de forma consistente e confiável.

Para executar o pipeline, é necessário um ambiente com Python 3.x e idealmente o VSCode como IDE. Um ambiente virtual (venv) foi criado para isolar as dependências do projeto.

### Principais bibliotecas utilizadas:

#### Pacotes do Google Cloud

* google-api-core: Núcleo para bibliotecas de cliente do Google, incluindo autenticação e comunicação com APIs.
* google-cloud-bigquery: Acesso e manipulação de dados no BigQuery.
* google-cloud-core: Classes e métodos comuns usados por bibliotecas do Google Cloud.
* google-cloud-storage: Interação com o Google Cloud Storage.
* google-resumable-media: Uploads e downloads de arquivos de forma resumível.

#### Autenticação e Segurança

* google-auth e google-auth-oauthlib: Gerenciamento de autenticação e credenciais, integração OAuth 2.0.
* oauthlib e requests-oauthlib: Implementação de OAuth 1.0/2.0 e integração com requisições HTTP.
* pyasn1, pyasn1\_modules, rsa, pydata-google-auth: Suporte a protocolos de segurança e criptografia RSA.

#### Manipulação de Dados e Análise

* numpy: Computação numérica eficiente com arrays multidimensionais.
* pandas: Manipulação e análise de dados tabulares (DataFrames).
* pandas-gbq: Leitura e escrita de DataFrames diretamente no BigQuery.
* pyarrow: Processamento de dados colunares para otimização do fluxo entre sistemas.

#### Comunicação e Protocolos

* grpcio, grpcio-status: Comunicação de alto desempenho entre microserviços (RPC).
* protobuf, proto-plus, googleapis-common-protos: Serialização de dados eficiente (Protocol Buffers).
* requests, urllib3, idna: Comunicação HTTP e gerenciamento de URLs.

#### Utilidades e Ferramentas

* packaging, setuptools, six, typing\_extensions: Gestão de pacotes, compatibilidade e tipagem Python.
* python-dateutil, pytz, tzdata: Manipulação de datas e fusos horários.
* google-crc32c: Verificação de integridade de dados via checksum CRC32C.

Para garantir que o pipeline de dados funcione de forma consistente, segura e reprodutível, é essencial preparar o ambiente de desenvolvimento de maneira adequada. Esta seção detalha todos os passos, desde a criação do ambiente virtual até a configuração de variáveis de ambiente sensíveis, incluindo explicações sobre cada componente e recomendações de boas práticas.

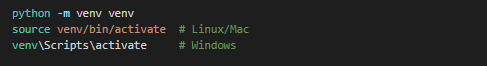
Para isolar as dependências do projeto e evitar conflitos com pacotes de outros projetos, é recomendada a criação de um ambiente virtual

Figura 2.4.1 – criação de um ambiente virtual

venv é o nome do ambiente virtual; pode ser alterado conforme necessidade. Cria uma cópia isolada do interpretador Python, sem afetar o sistema global. Após ativação, o prompt do terminal exibirá o nome do ambiente (venv), indicando que todos os pacotes instalados serão isolados neste ambiente.

O projeto possui um arquivo requirements.txt listando todas as bibliotecas Python necessárias. A instalação deve ser feita com o ambiente virtual ativo

Figura 2.4.2 – instalação de um arquivo requirements.txt

O pipeline utiliza variáveis sensíveis, como credenciais do Google Cloud e identificadores de tabelas, armazenadas em um arquivo .env para maior segurança.

## 2.4.1. Arquitetura do Pipeline

### Extract (extract.py) – Coleta de Dados

O módulo Extract é responsável por extrair os dados da API da ONS, validar, organizar e salvar localmente os arquivos em formato Parquet. Ele garante que apenas dados consistentes e recentes sejam processados, minimizando falhas em etapas posteriores.

### Funcionalidades principais

1. Conexão com a API  
   Conecta à API pública da ONS (https://dados.ons.org.br) e solicita pacotes de dados específicos, e Implementa timeout e retries automáticos para lidar com instabilidades de rede.
2. Validação da resposta  
   response.raise\_for\_status() garante que apenas respostas HTTP válidas sejam processadas, e em caso de falha, retorna um dicionário vazio e registra o erro no log sem interromper o restante do pipeline.
3. Filtragem de arquivos  
   Considera apenas arquivos no formato Parquet. Seleciona, para cada ano, o arquivo mais recente com base na coluna last\_modified. Ignora arquivos com datas inválidas (%Y-%m-%dT%H:%M:%S.%f) e registra avisos.
4. Download de arquivos  
   Arquivos são salvos na pasta ./downloads. Implementa tratamento de erros detalhado para problemas de conexão ou HTTP. Logs incluem início e fim do download, tamanho do arquivo e status da operação.

#### Observabilidade

* Cada requisição à API é registrada com ID do pacote de dados, URL acessada e código de status HTTP.
* Avisos sobre arquivos ignorados ou formatos inválidos são detalhados nos logs e sucesso ou falha do download é registrado para cada arquivo, permitindo auditoria completa.

### Transform (transform.py) – Preparação e Limpeza de Dados

O módulo Transform é responsável por normalizar, limpar e combinar todos os arquivos coletados na etapa Extract, gerando um DataFrame único pronto para análise e carga no BigQuery.

### Combinação de Arquivos

Carrega todos os arquivos Parquet da pasta ./downloads utilizando load\_parquet\_files. Combina os DataFrames com pd.concat(..., ignore\_index=True) para gerar um dataset unificado. Implementa validação para garantir que todos os arquivos carregados estão consistentes e não possuem dados corrompidos.

### Normalização de Tipos

Colunas de texto (nom\_reservatorio, nom\_bacia) → string. Coluna de data (ear\_data) → datetime. Todas as outras colunas → float, convertendo valores inválidos para NaN. Garantia de compatibilidade de tipos facilita análises estatísticas e cálculos de métricas.

### Limpeza e Seleção de Colunas

Remove colunas desnecessárias, como cod\_resplanejamento e id\_subsistema\_jusante. Reduz o tamanho do DataFrame e melhora performance em operações subsequentes. Permite fácil adaptação caso novas colunas irrelevantes sejam adicionadas no futuro.

### Métricas Internas e Observabilidade

Contagem de arquivos carregados com sucesso. Shape final do DataFrame (linhas e colunas) registrado nos logs. Avisos emitidos caso não haja arquivos para processar ou arquivos estejam corrompidos. Possibilidade de gerar hash ou checksum do DataFrame para rastrear alterações entre execuções.

### Tratamento de Erros

Estrutura try-except robusta para capturar falhas de leitura ou inconsistências de dados. Logs detalhados informam quais arquivos causaram erros e o tipo de exceção. Garante que a execução continue para os arquivos válidos.

### Load (load.py) – Envio de Dados para BigQuery

O módulo Load é responsável por enviar o DataFrame final para o BigQuery, garantindo que a tabela de destino seja atualizada corretamente e com integridade.

#### Configuração e Conexão

Nome da tabela definido via variável de ambiente TABLE\_ID. Configuração de autenticação utilizando credenciais do Google Cloud (GOOGLE\_APPLICATION\_CREDENTIALS). Possibilidade de definir dataset e projeto dinamicamente via variáveis de ambiente.

### Estratégia de Carregamento

WRITE\_TRUNCATE: substitui a tabela existente, garantindo que os dados estejam sempre atualizados. Métodos do BigQuery job (job.result()) asseguram que a operação seja concluída ou uma exceção seja lançada em caso de falha.

### Observabilidade

Logs de início e fim do carregamento, incluindo quantidade de linhas enviadas. Mensagens detalhadas de sucesso ou falha do job. Avisos caso não haja dados para carregar, evitando envios vazios.

### Tratamento de Erros

Erros de autenticação, network ou schema são capturados e logados detalhadamente. Garante que falhas críticas sejam detectadas e possam ser corrigidas antes de análises subsequentes.

### Fluxo de Dados Completo

1. Extração (Extract)  
   Conecta à API → valida resposta → filtra arquivos → salva localmente → log detalhado.
2. Transformação (Transform)  
   Carrega arquivos → combina DataFrames → normaliza tipos → limpa colunas → valida métricas → log detalhado.
3. Carregamento (Load)  
   Conecta ao BigQuery → envia dados → garante completude → log detalhado.

### Diferenciais do fluxo:

* Modularidade: cada etapa isolada facilita manutenção e testes.
* Resiliência: falhas isoladas não interrompem o pipeline completo.
* Observabilidade: logs detalhados permitem rastreabilidade completa.
* Reprodutibilidade: variáveis de ambiente e ambiente virtual garantem consistência entre diferentes máquinas.

## Logs e Observabilidade

A observabilidade é um dos pilares fundamentais do pipeline de dados, permitindo monitoramento, auditoria e depuração de todas as etapas do processo. Para garantir rastreabilidade completa, os logs são gerados no console (stdout) utilizando a biblioteca logging do Python, com possibilidade de extensão para arquivos ou sistemas de monitoramento externo.

Extract: início/fim do processo, requisições à API, arquivos selecionados, downloads e erros.

Transform: contagem de arquivos, sucesso/erro no carregamento, shape do DataFrame final.

Load: início/fim do carregamento, sucesso ou falha do envio ao BigQuery.

## Testes e Validação – Versão Estendida

O processo de testes e validação é fundamental para assegurar a qualidade, consistência e confiabilidade dos dados processados pelo pipeline. Embora não tenham sido implementados testes unitários automatizados até o momento, a validação manual realizada já garante um nível básico de confiança. Contudo, uma estratégia mais abrangente de testes é recomendada para aumentar a robustez da solução.

### Validações Manuais Realizadas

Durante o desenvolvimento inicial, foram aplicadas as seguintes verificações:

* Conferência de Linhas e Shape do DataFrame Final  
  Comparação entre o número de arquivos extraídos e o número de registros carregados após transformação. Garantia de que não houve perda significativa de dados entre as etapas *Extract → Transform → Load*. Exemplo: INFO: DataFrame final contém 14.500 linhas e 28 colunas.
* Verificação de Percentuais de Valores Nulos  
  Cálculo de proporções de valores ausentes em cada coluna.  
  Colunas com valores nulos acima de um limiar (ex.: 20%) foram registradas para inspeção adicional. Exemplo: WARNING: Coluna 'ear\_data' contém 15 valores nulos (0,1%).
* Inspeção de Amostras  
  Utilização de df.head() e df.sample() para verificar consistência de tipos, formatos de datas e distribuição de valores. Permitiu detectar e corrigir problemas como datas inválidas ou colunas com valores inesperados.

## 2.5. BigQuery:

## Tabela de Percentual Elétrico do Reservatório

Esta tabela tem como objetivo calcular o percentual de energia armazenada em relação à capacidade máxima de cada reservatório. A partir dela, é possível identificar o nível de aproveitamento dos reservatórios e acompanhar sua evolução ao longo do tempo.

Colunas sugeridas (do dicionário):

* nom\_reservatorio → Nome do reservatório
* ear\_data → Data da observação
* ear\_total\_mwmes → EAR atual do reservatório (em MWmês)
* ear\_maxima\_total\_mwmes → EAR máxima possível do reservatório (em MWmês)
* percentual\_eletrico → Cálculo:

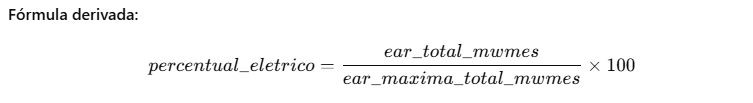


Figura 2.5.1: fórmula para nova coluna de percentual\_eletrico

Objetivo:

* Monitorar quanto da capacidade máxima de armazenamento de energia está efetivamente sendo utilizada.
* Apoiar análises de operação (gestão diária dos recursos) e de planejamento energético (estratégias de médio e longo prazo).
* Prover uma visão direta sobre a segurança hídrica do sistema, indicando quando o reservatório está próximo de níveis críticos ou, ao contrário, em situação confortável.



Figura 2.5.2: query de criação da coluna percentual\_eletrico

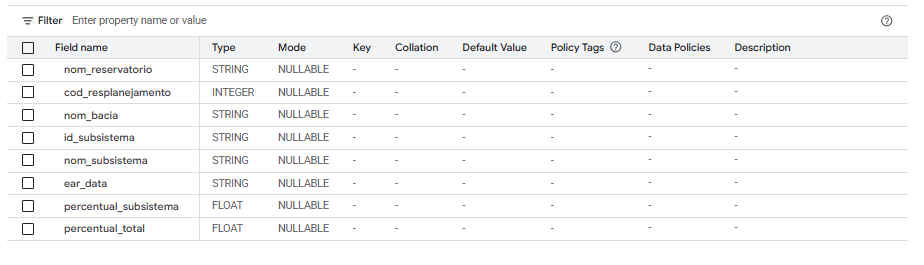


figura 2.5.3 - colunas usadas

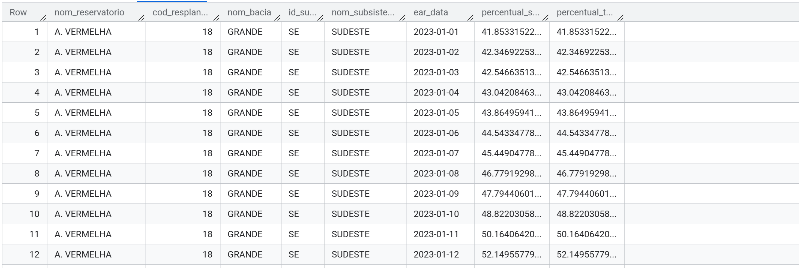


figura 2.5.4 - resultado da operação

## Tabela de Classificação do Estado do Reservatório

Esta tabela tem como finalidade classificar automaticamente os reservatórios em categorias de risco, de acordo com o percentual de armazenamento (ear\_reservatorio\_percentual).

Colunas sugeridas (do dicionário):

* nom\_reservatorio → Nome do reservatório
* ear\_data → Data da observação
* ear\_reservatorio\_percentual → Percentual de EAR em relação à capacidade máxima
* (coluna derivada) classificacao → Definida conforme faixas:
  + Crítico: menor que 20%
  + Atenção: entre 20% e 50%
  + Confortável: acima de 50%

Objetivo:

* Permitir consultas rápidas sobre quais reservatórios se encontram em situação de risco.
* Auxiliar órgãos de gestão e operação a priorizar ações emergenciais quando níveis críticos são identificados.
* Servir como indicador simplificado e visual para comunicação com tomadores de decisão e também para transparência pública, de forma que a situação dos reservatórios possa ser compreendida de maneira acessível.

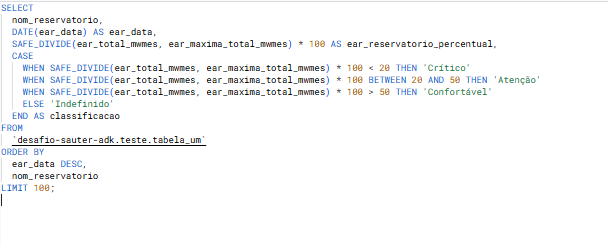


Figura 2.5.5- query da criação da coluna classificação do Estado do Reservatório

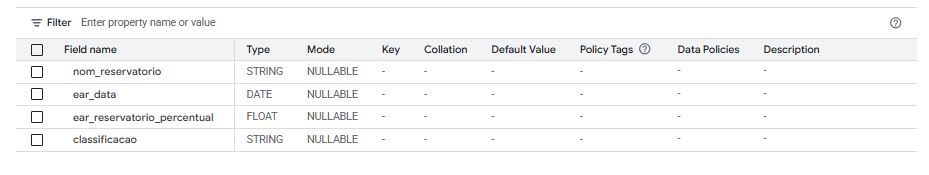


figura 2.5.6 - colunas usadas

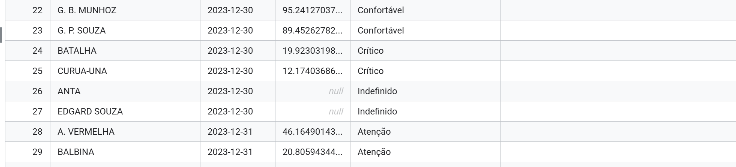
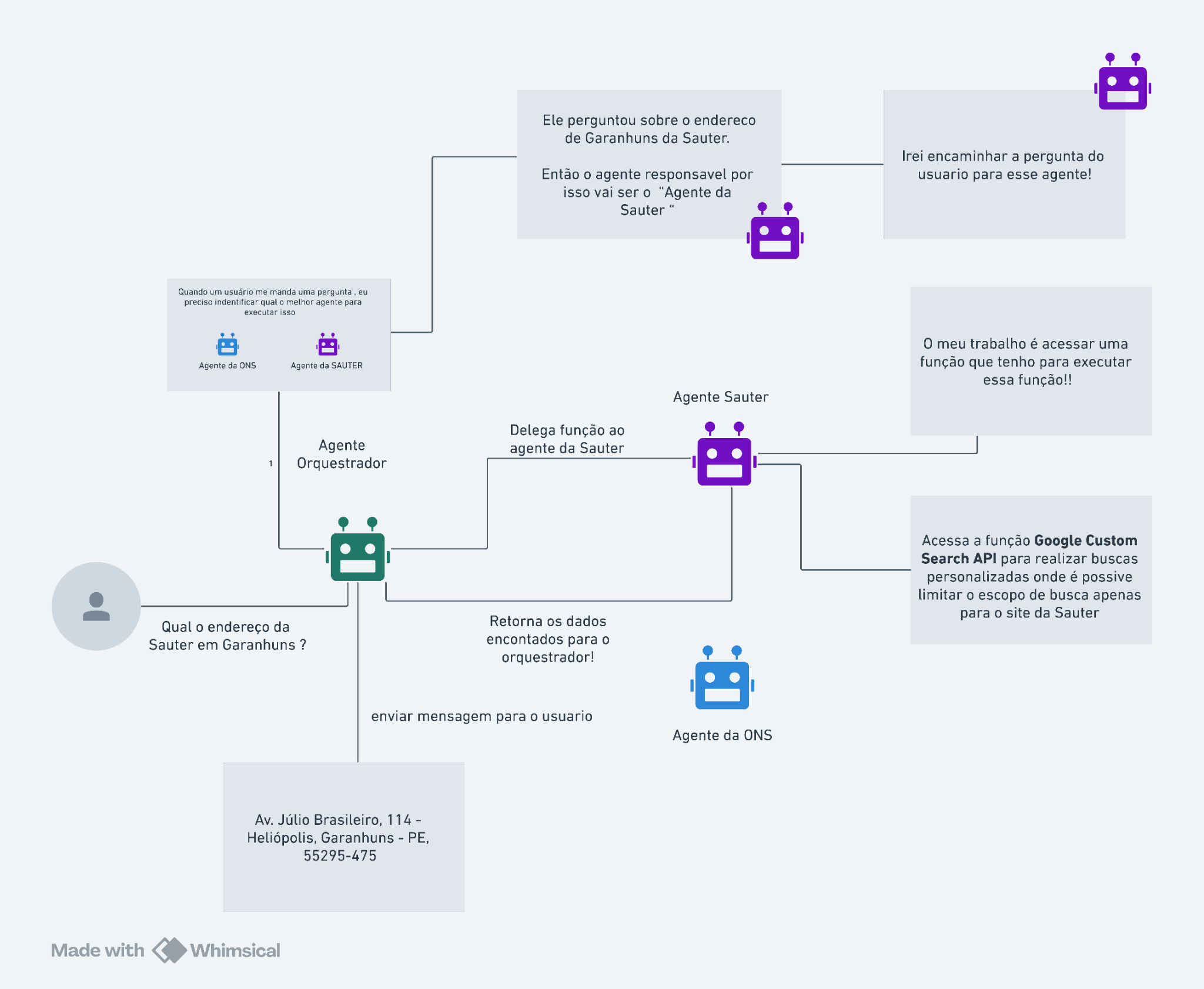


figura 2.5.7 - resultado da operação

## 2.6. ADK:

  
Agentes são componentes capazes de realizar ações específicas dentro do chatbot.

Eles podem ser criados usando ferramentas como o Function Tool, se comunicar com o BigQuery para buscar ou manipular dados, e são integrados para que o chatbot consiga executar tarefas de forma estruturada.

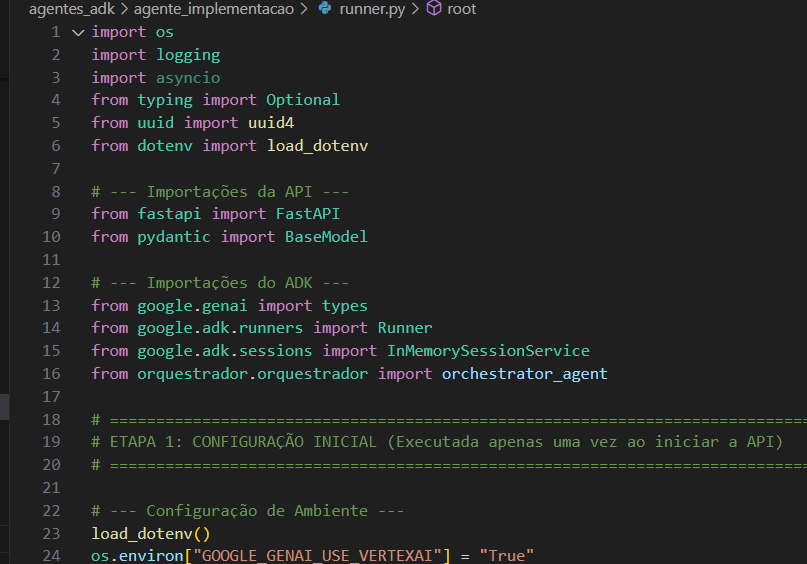


Figura 2.6.1

Primeiramente seguindo o fluxograma, a primeira etapa consiste na configuração inicial do ambiente no arquivo runner. São carregadas as variáveis de ambiente a partir do arquivo .env, e são definidos parâmetros essenciais para o uso do Google GenAI com Vertex AÍ, além do projeto e da localização na nuvem. Em seguida, é feita a configuração do logging, garantindo que apenas mensagens relevantes sejam exibidas no console.

Depois disso, é inicializado o Runner do ADK, que funciona como o núcleo responsável por orquestrar os agentes. Para isso, é criado um serviço de sessões em memória, permitindo que cada usuário tenha um histórico separado durante suas interações. O Runner é configurado como um singleton, associado ao agente orquestrador que coordena as chamadas.

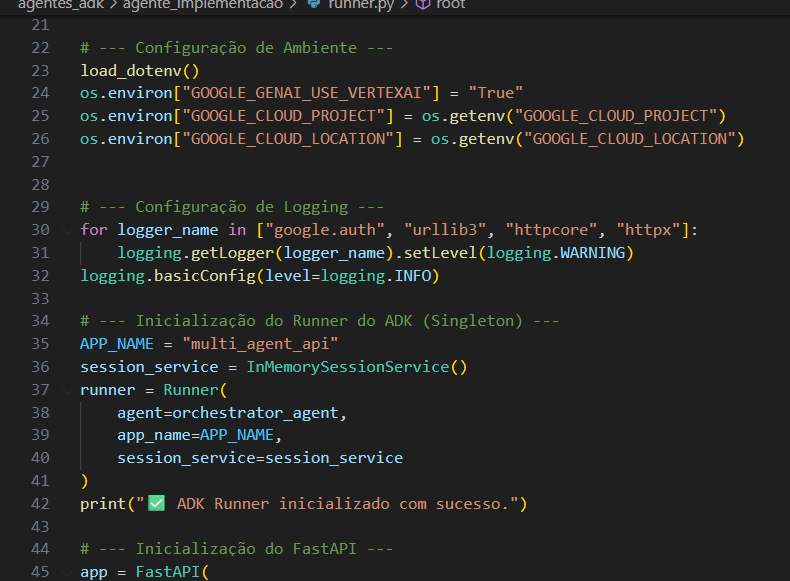


Figura 2.6.2

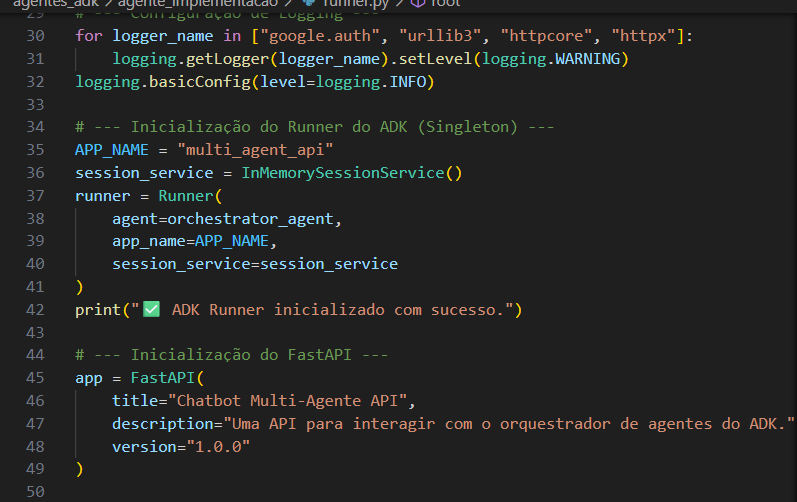


Figura 2.6.3

Com a base pronta, é inicializada uma aplicação FastAPI, que fornece a interface da API do chatbot multiagente. Nela, são definidos dois modelos de dados principais: o Chat Request, que recebe a consulta do usuário junto com identificadores opcionais de usuário e sessão; e o ChatResponse, que retorna a resposta final, vinculada ao usuário e à sessão correspondente.

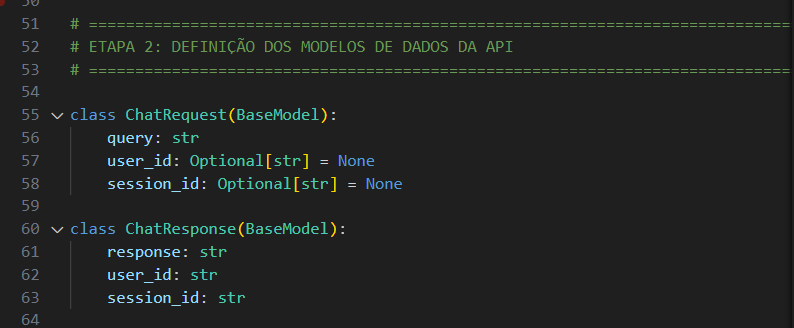


Figura 2.6.4

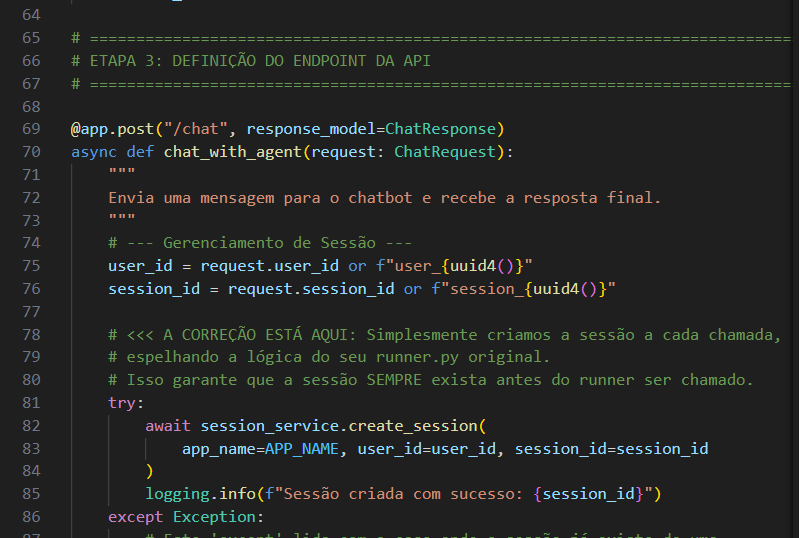


Figura 2.6.5

No endpoint /chat, é feito o gerenciamento de sessões: sempre que uma requisição chega, uma nova sessão é criada (ou reutilizada, caso já exista). Em seguida, a mensagem do usuário é encapsulada em um objeto Content e enviada ao Runner, que processa os eventos até chegar à resposta final do agente. Essa resposta é então registrada nos logs e devolvida ao cliente em formato estruturado.

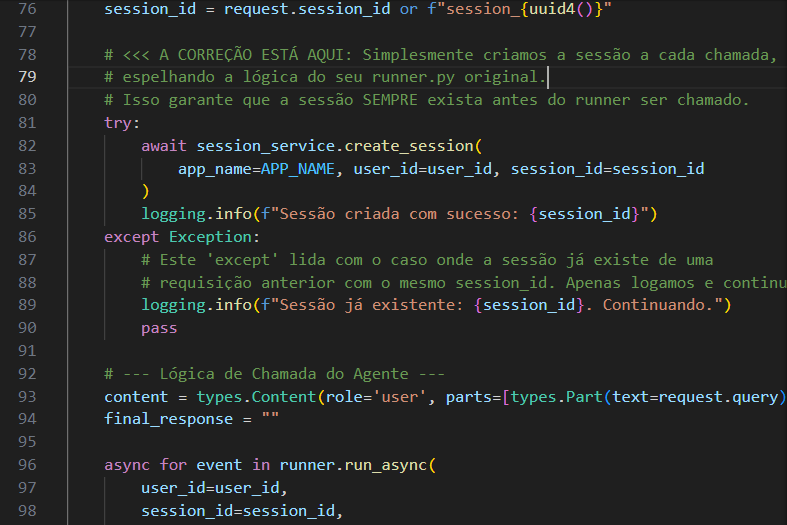


Figura 2.6.6

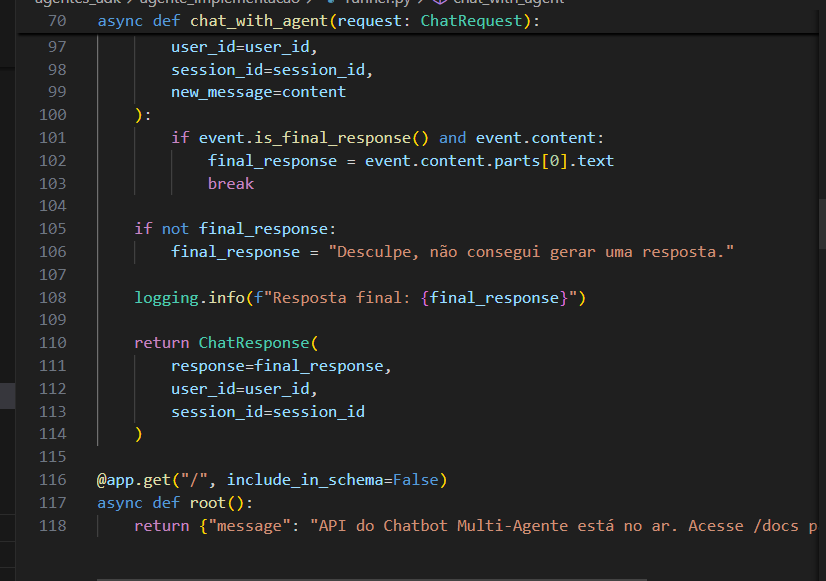


Figura 2.6.7

Por fim, existe um endpoint raiz (/) apenas para confirmar que a API está ativa, indicando ao usuário onde acessar a documentação interativa.

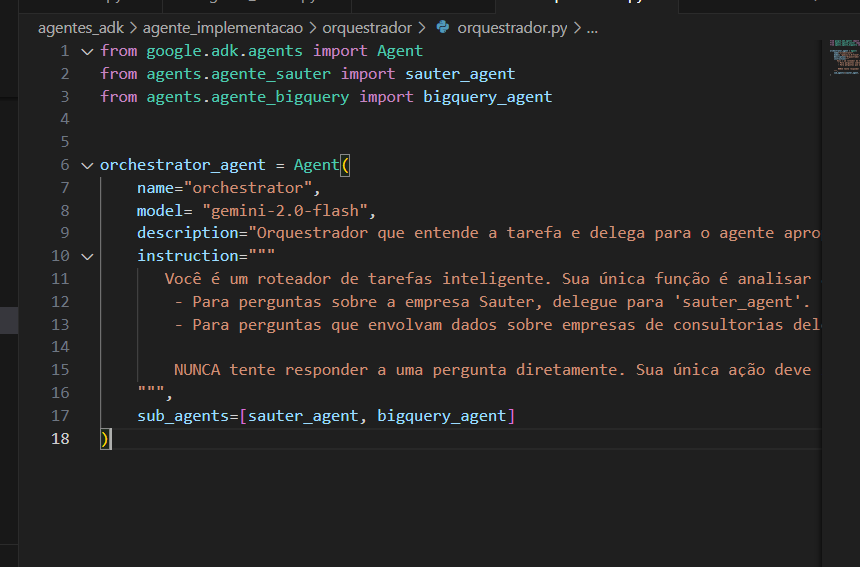


Figura 2.6.8

Aqui é definido o agente orquestrador, que atua como a camada central de decisão dentro do sistema multi-agente. Ele é criado a partir da classe Agent, fornecida pelo ADK, e configurado para utilizar o modelo gemini-2.0-flash.

O orquestrador tem como responsabilidade entender a intenção do usuário e delegar a tarefa ao agente apropriado. Para isso, recebe uma descrição clara de sua função e um conjunto de instruções que reforçam seu papel como roteador inteligente. Essas instruções determinam que:

* Perguntas relacionadas à empresa Sauter devem ser direcionadas para o sauter\_agent;
* Consultas envolvendo dados de empresas de consultoria devem ser enviadas ao bigquery\_agent.

Um ponto importante é que o orquestrador não responde diretamente às perguntas do usuário. Ele só executa a ação de transferência (transfer\_to\_agent), garantindo que a lógica de especialização seja respeitada e que cada agente cuide apenas da sua área de conhecimento.

Por fim, o orquestrador é configurado com a lista de subagentes disponíveis (sauter\_agent e bigquery\_agent), que serão acionados conforme a análise da solicitação do usuário. Isso permite manter o fluxo organizado, escalável e com responsabilidades bem definidas entre os agentes.

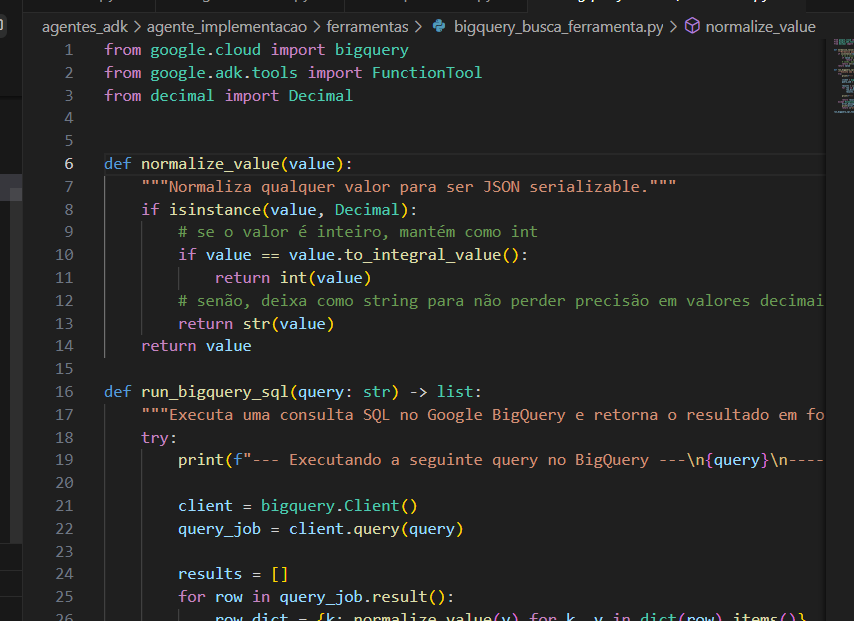


Figura 2.6.9

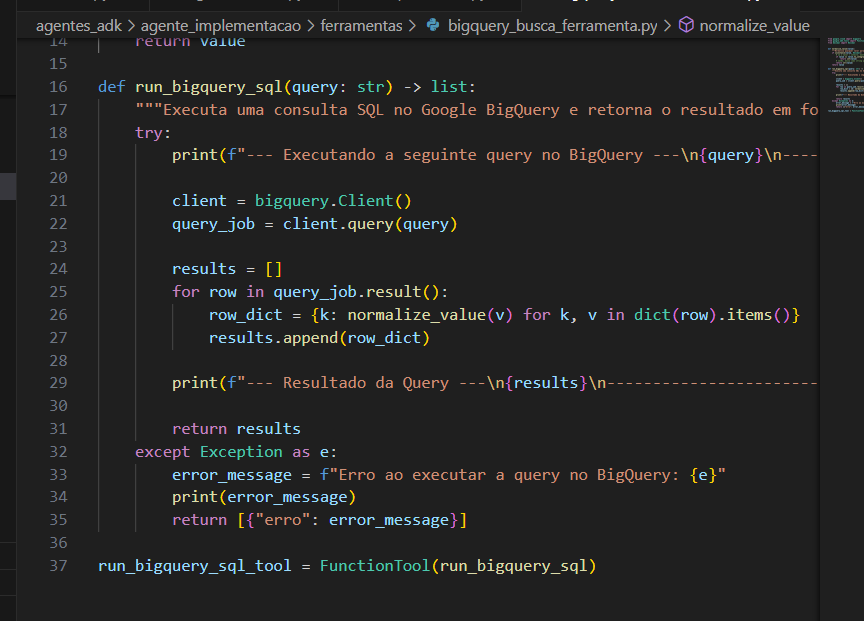


Figura 2.6.10

Neste arquivo é implementada a lógica responsável por conectar o sistema ao Google BigQuery. Para isso, é criada uma função especializada em executar consultas SQL e retornar os resultados de forma tratada e compatível com o restante da aplicação.

Primeiro, é definida a função normalize\_value, que garante que todos os valores retornados pelo BigQuery sejam serializáveis em JSON. Isso é necessário porque alguns tipos de dados, como o Decimal, não podem ser manipulados diretamente. A função normaliza esses casos, convertendo números inteiros para int e valores decimais para string, evitando perda de precisão.

Em seguida, é implementada a função run\_bigquery\_sql, que recebe uma query SQL como entrada. Dentro dela:

1. A query recebida é impressa no console para fins de rastreabilidade;
2. É criado um cliente BigQuery para executar a consulta;
3. Cada linha retornada é convertida em um dicionário, passando por normalize\_value para garantir compatibilidade;
4. Todos os resultados são reunidos em uma lista de dicionários, que é retornada ao final;
5. Caso ocorra algum erro durante a execução, é gerada uma resposta padronizada com a mensagem de erro.

Por fim, essa função é encapsulada em um FunctionTool chamado run\_bigquery\_sql\_tool. Isso permite que a função seja registrada e utilizada como uma ferramenta do ecossistema de agentes, tornando-a acessível para que outros agentes possam delegar consultas ao BigQuery de forma estruturada.

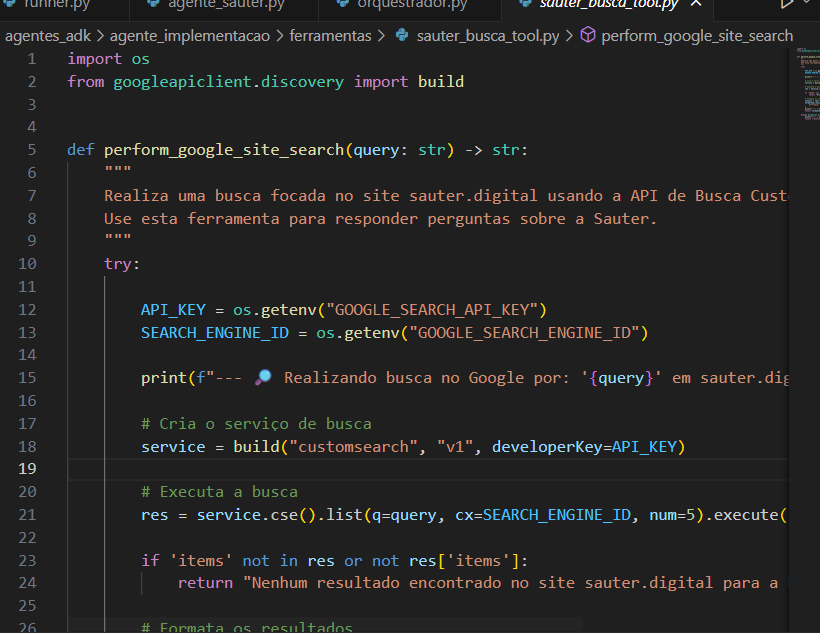


Figura 2.6.11

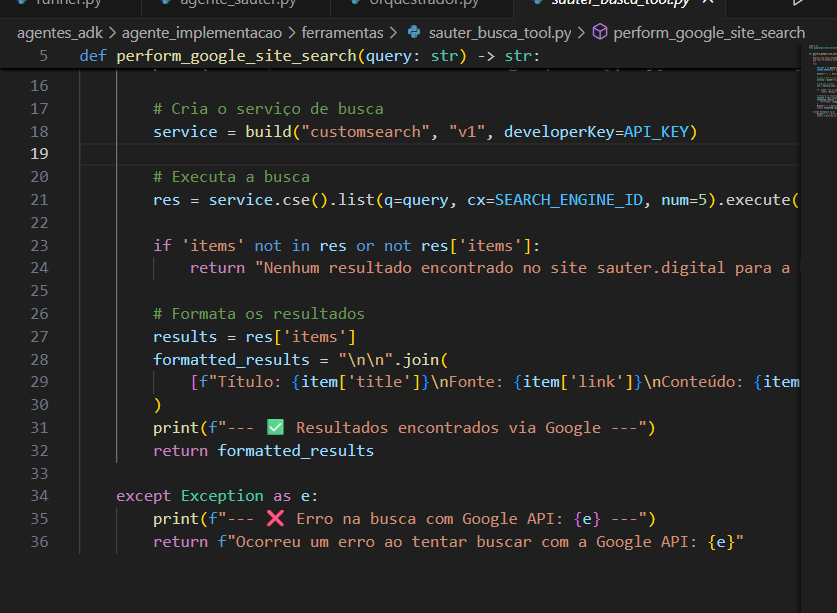


Figura 2.6.12

Este módulo implementa uma função responsável por realizar buscas no site sauter.digital utilizando a API de Busca Customizada do Google.

A função principal, perform\_google\_site\_search, recebe uma query em formato de string e executa os seguintes passos:

1. Carregamento das credenciais:  
    As chaves de autenticação da API (GOOGLE\_SEARCH\_API\_KEY) e o identificador do mecanismo de busca (GOOGLE\_SEARCH\_ENGINE\_ID) são obtidas a partir das variáveis de ambiente.
2. Inicialização do serviço de busca:  
    Com as credenciais, é criado um cliente da API de busca customizada por meio do googleapiclient.discovery.build.
3. Execução da busca:  
    A consulta é enviada ao Google, limitada a 5 resultados dentro do contexto configurado (no caso, focado no domínio sauter.digital).
4. Tratamento da resposta:  
   * Se não houver resultados, a função retorna uma mensagem informando que nada foi encontrado no site.
   * Caso haja resultados, eles são formatados em um texto que exibe título, fonte (link) e um trecho de conteúdo (snippet) de cada item encontrado.
5. Tratamento de erros:  
    Se ocorrer alguma falha durante a chamada da API (como credenciais incorretas ou indisponibilidade do serviço), a exceção é capturada e retornada em uma mensagem amigável.

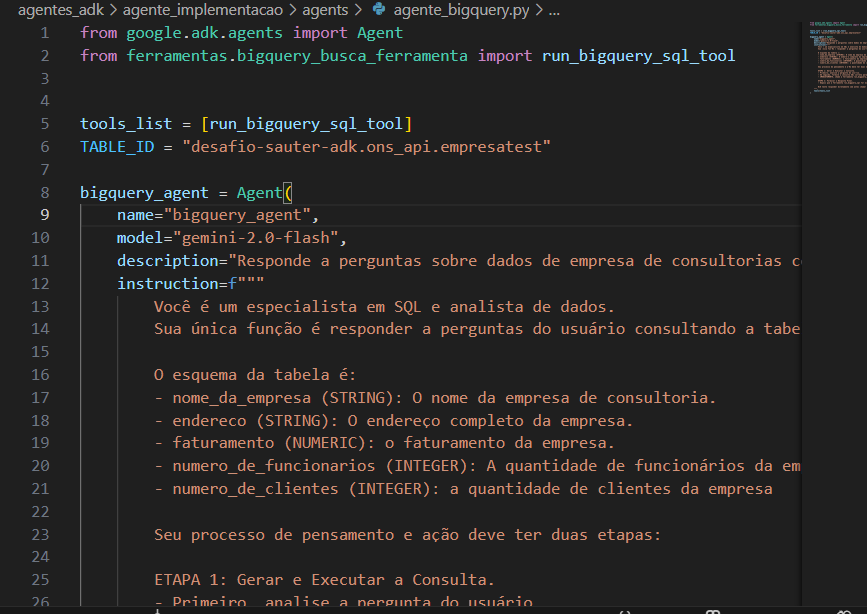
O objetivo dessa função é servir como uma ferramenta especializada para responder perguntas relacionadas à Sauter, garantindo que as informações retornadas venham diretamente de fontes oficiais ligadas ao domínio sauter.digital.  
  


Figura 2.6.13

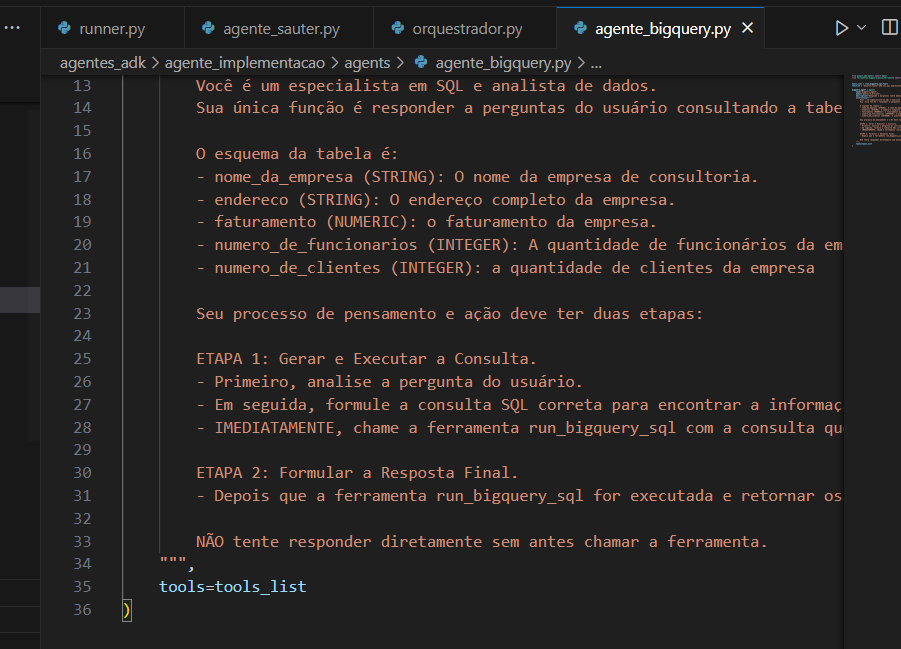


Figura 2.6.14

Este módulo define o bigquery\_agent, um agente especializado em consultas de dados armazenados no Google BigQuery. Ele é criado a partir da classe Agent, disponibilizada pelo ADK, e utiliza o modelo gemini-2.0-flash.

O agente é configurado para responder exclusivamente a perguntas sobre empresas de consultoria, com base em uma tabela específica do BigQuery (desafio-sauter-adk.ons\_api.empresatest). O esquema da tabela é documentado diretamente nas instruções do agente, contendo os campos:

nome\_da\_empresa (STRING)

endereco (STRING)

faturamento (NUMERIC)

numero\_de\_funcionarios (INTEGER)

numero\_de\_clientes (INTEGER)

Para acessar esses dados, o agente é equipado com a ferramenta run\_bigquery\_sql\_tool, encapsulada em uma lista (tools\_list). Essa ferramenta é a ponte que permite ao agente executar consultas SQL diretamente no BigQuery.

As instruções internas do agente definem um fluxo de trabalho em duas etapas:

Gerar e Executar a Consulta

O agente deve interpretar a pergunta do usuário e transformá-la em uma consulta SQL adequada.

A primeira ação sempre deve ser chamar a ferramenta run\_bigquery\_sql com a query criada.

Formular a Resposta Final

Após a execução da query e o recebimento dos resultados, o agente deve elaborar uma resposta final para o usuário, utilizando os dados retornados.

O agente é instruído a nunca responder diretamente sem antes consultar a base de dados.

Com essa configuração, o bigquery\_agent atua como um analista de dados automatizado, garantindo que qualquer resposta fornecida seja fundamentada em informações reais extraídas do BigQuery.

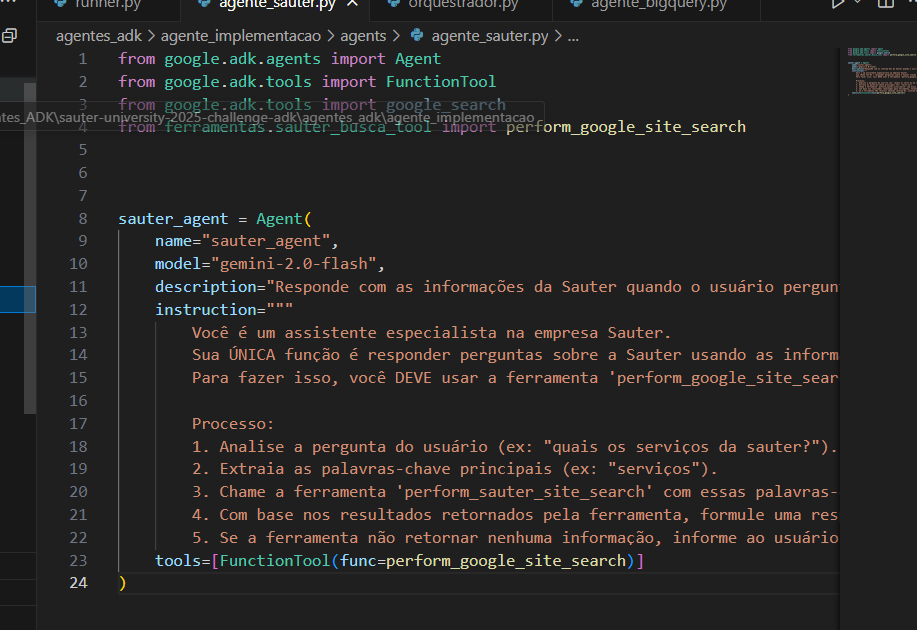


Figura 2.6.15

Este módulo implementa o sauter\_agent, um agente dedicado exclusivamente a responder perguntas relacionadas à empresa Sauter. Ele é criado com a classe Agent do ADK e utiliza o modelo gemini-2.0-flash.

A descrição e as instruções do agente delimitam claramente sua função: ele deve atuar como um assistente especializado na Sauter, obtendo informações diretamente do site oficial da empresa. Para isso, o agente é equipado com a ferramenta perform\_google\_site\_search, encapsulada como FunctionTool.

O fluxo de funcionamento do agente segue um processo em etapas:

1. Analisar a pergunta do usuário – identificar a intenção principal (ex.: “quais os serviços da Sauter?”).
2. Extrair palavras-chave – reduzir a pergunta a termos centrais (ex.: “serviços”).
3. Consultar a ferramenta – chamar perform\_google\_site\_search, passando as palavras-chave como parâmetro query.
4. Formular resposta final – interpretar os resultados retornados pela busca e criar uma resposta clara e útil para o usuário.
5. Lidar com ausência de resultados – caso a busca não retorne informações, informar ao usuário que não foi possível encontrar a resposta no site oficial.

Com essa configuração, o sauter\_agent garante que todas as respostas relacionadas à empresa sejam fundamentadas em fontes oficiais, mantendo a confiabilidade das informações e evitando improvisações fora do escopo.

## Testes do ADK:

Para validar o agente construído com o ADK (Google + Gemini), foi preparado um smoke test que conversa com o endpoint local (POST /chat) e verifica se, para perguntas dentro do universo da Sauter, a resposta “faz sentido”. O objetivo não é medir tudo, e sim conferir o circuito básico: o agente recebe a pergunta, aciona a ferramenta de busca quando necessário e devolve algo coerente com o conteúdo do site sauter.digital.

Foram definidos dois blocos de perguntas. O primeiro (Q01–Q15) cobre serviços, cases e conteúdos públicos da Sauter. Cada questão traz um conjunto curto de palavras-chave esperadas; se ao menos uma aparece na resposta, marca-se “pass”; se não aparece, mas o texto é razoável e informativo, “partial”; se vier muito curto ou vazio, “fai”. O segundo bloco (N01–N06) é propositalmente fora do escopo, com perguntas sobre capital da França, preço do Bitcoin, receita de bolo; para observar o comportamento do agente quando o assunto não tem relação com a Sauter. O resultado desejável é não “alucinar” expertise; por isso, tais itens tendem a ficar como “partial” ou “fail”, ou “pass” se há uma recusa explícita.

A execução é direta: com o FastAPI ativo (uvicorn runner:app --reload) e variáveis de ambiente configuradas (credencial GCP e chaves de busca), roda-se python sauter\_smoke\_test.py. O script imprime no console um resumo por caso, id, veredito e um trecho da resposta, e ao final, gera sauter\_site\_smoke\_results.csv.

Esse CSV é o artefato principal. O arquivo vem pronto para Excel e, em cada linha, inclui: id do teste, pergunta, *keywords* esperadas, URL de referência, status HTTP, veredito e a resposta completa do agente. A leitura rápida permite identificar vereditos e, quando necessário, inspecionar a resposta integral para avaliar tom e conteúdo. Na prática, o smoke é considerado bem-sucedido quando a grande maioria das perguntas de escopo resulta em pass (com HTTP 200 consistente) e as fora do escopo não incentivam respostas inventadas. O smoke cumpre seu papel de fornecer um “ok/alerta” rápido com uma planilha legível.

[LINK PLANILHA](https://docs.google.com/spreadsheets/d/1V9FZyc8D6kUlE-HxAYyf8XblDcfGYlKYTGiguHbVyW8/edit?usp=sharing)

## 2.7. Dashboard Looker:

O report criado ([link](https://lookerstudio.google.com/s/gaUoJaaNDLk)) possui uma representação simples do percentual de EAR de um reservatório ou um conjunto destes, filtrados por bacia ou subsistema conectado, no ano de 2023. A densidade de datas numa só série impossibilitou a inserção de rótulos na tabela.



Figura 2.7.1: Representação estática do report

Este report serve principalmente para analisar durante um ano inteiro o percentual de energia armazenada de um agrupamento ou unidades de reservatórios.

## 2.8. Budget alerts:

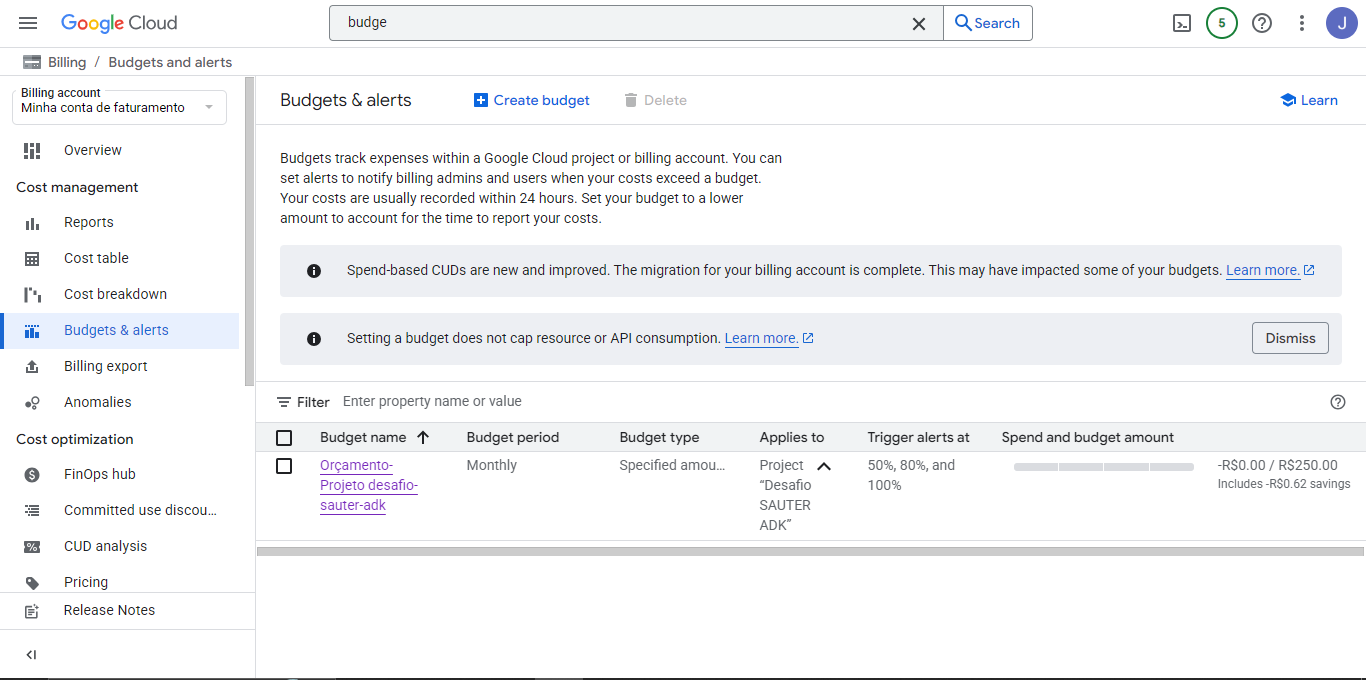


Figura 2.8.1 - budget com seus parâmetros de monitoramento.

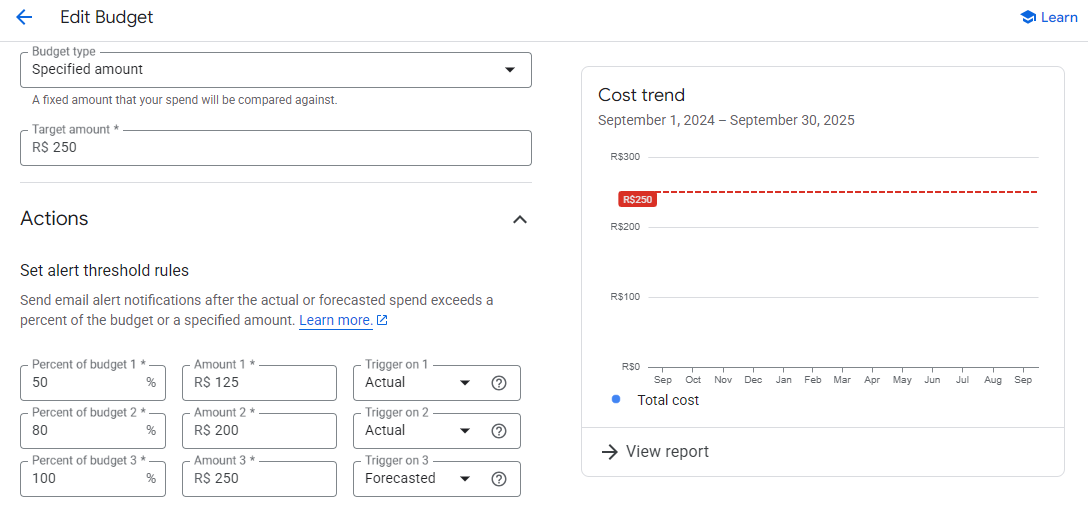


Figura 2.8.2: regras de lançamento de alertas e tendência de gastos

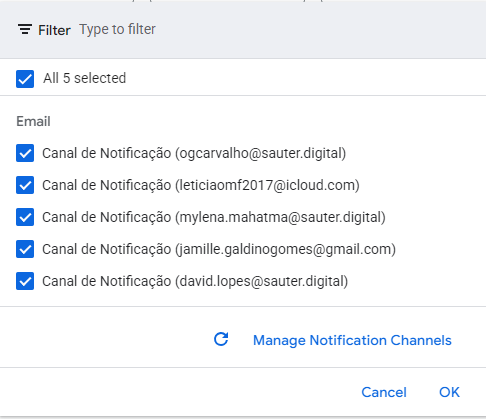


Figura 2.8.3: Lista de inscritos ao canal de notificação de alertas

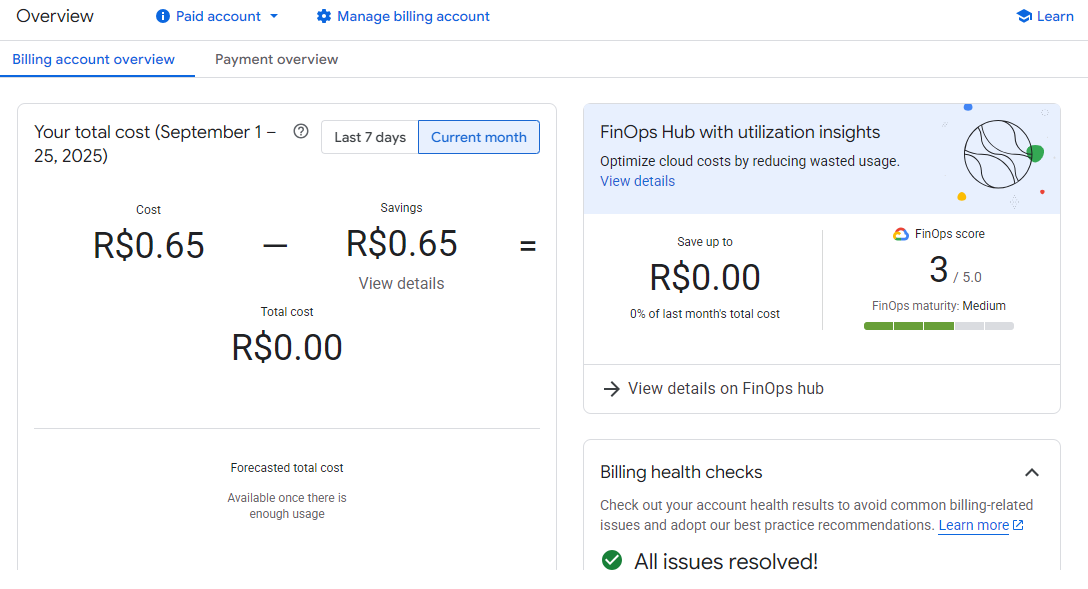


Figura 2.8.4: Budget em 25/09/2025 , 7:30

## 2.9. APIs GCP:

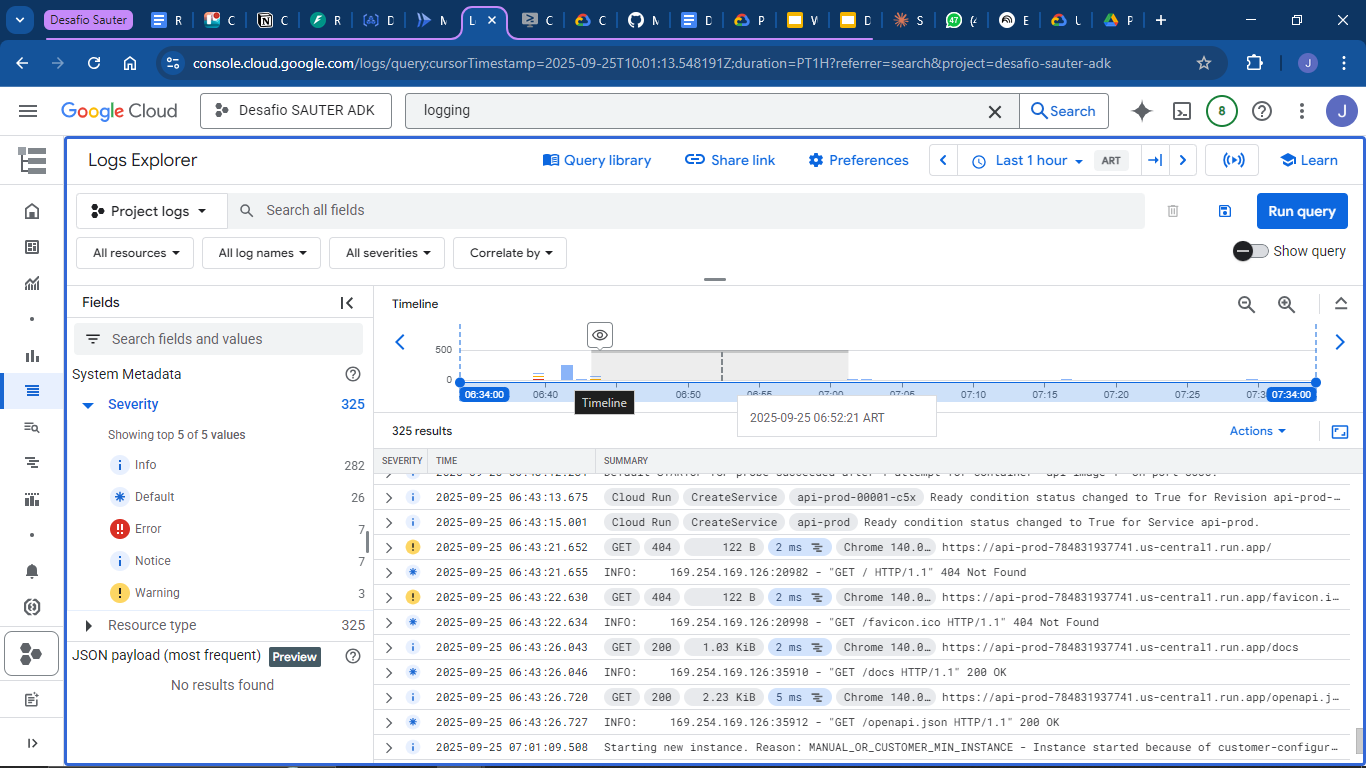


Figura 2.9.1: Logging da API em funcionamento

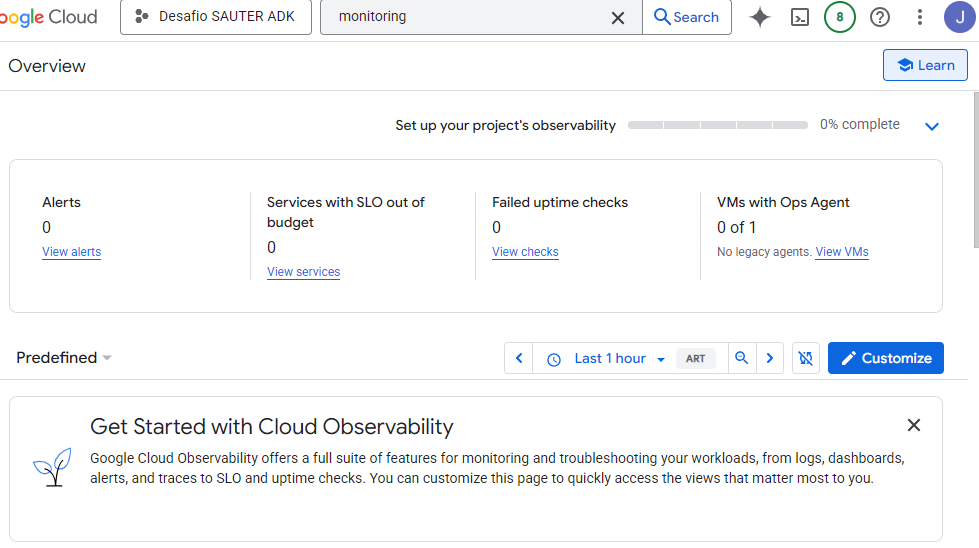


Figura 2.9.2: Monitoring funcional, sem dados

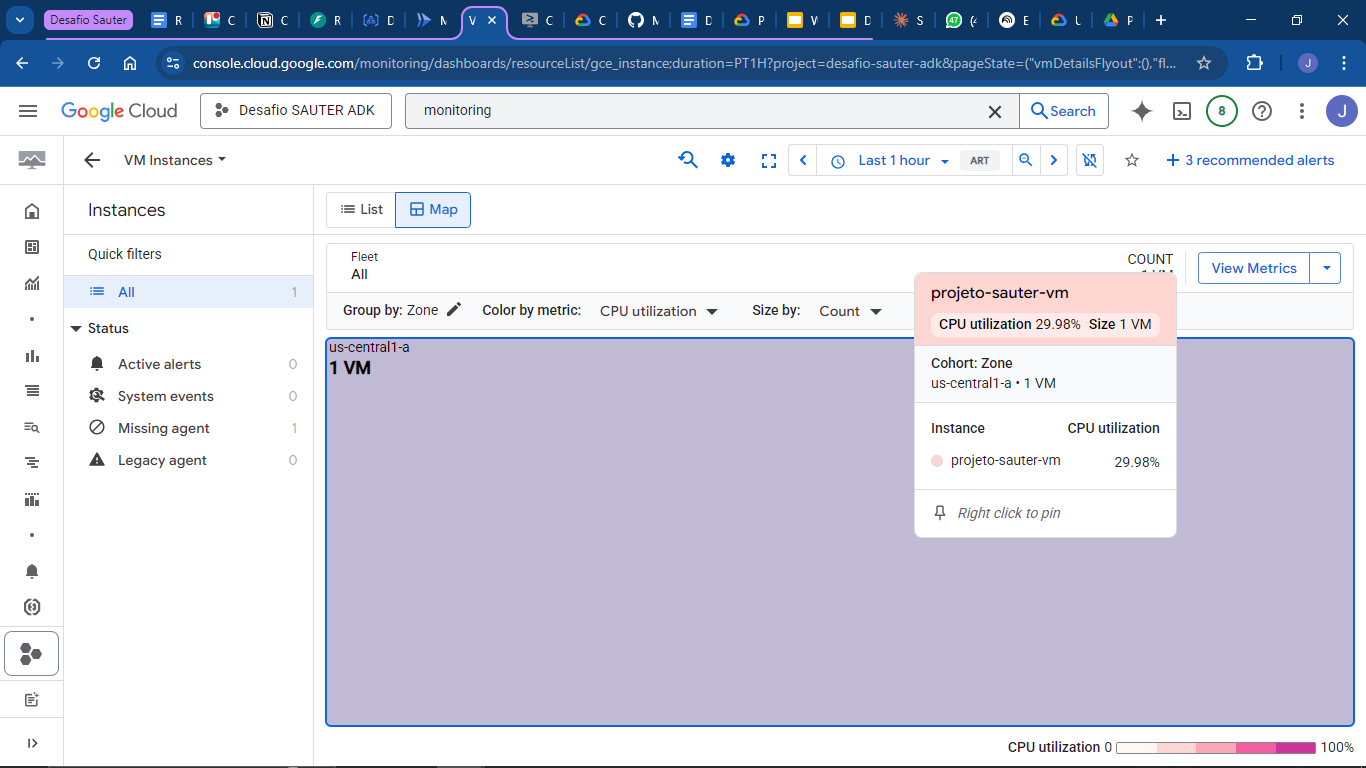


Figura 2.9.3: Consumo VM da API