

Guia de Implementação de Ingestão Recorrente no GCP com Cloud Run Jobs

Este documento descreve, de forma estruturada, como implementar um pipeline completo de ingestão de dados no Google Cloud Platform utilizando:

- Cloud Run Jobs
- Cloud Storage (Buckets)
- BigQuery
- Docker
- Agendamento (Scheduler)

O objetivo é criar um fluxo **automático, auditável e escalável** para ingestão diária e histórica de dados.

Visão Geral deste Fluxo

Antes de entrar nos comandos, é fundamental entender o desenho lógico deste pipeline.

O processo funciona em **quatro macro-etapas**:

1. Carga Diária

Uma **Job** é executada todos os dias e:

- Busca os dados do dia anterior (esta lógica vai depender do seu script)
- Gera um arquivo Parquet
- Salva esse arquivo em um bucket no Cloud Storage

2. Carga Histórica (One-shot)

Um segundo script:

- Executa apenas uma vez
- Busca todos os dias passados
- Gera milhares de arquivos Parquet
- Preenche o mesmo bucket

Esse script histórico **não é agendado**. Ele é usado apenas para trazer o histórico inicial.

3. Armazenamento por Data

Os arquivos ficam organizados no bucket por:

year=YYYY/month=MM/day=DD

Isso permite que o BigQuery faça leitura incremental eficiente.

4. Consumo no BigQuery

O BigQuery passa a:

- Ler diretamente os Parquets no bucket
- Através de uma **External Table**

Etapa 1 — Estrutura do Projeto

O primeiro passo é criar o repositório do projeto.

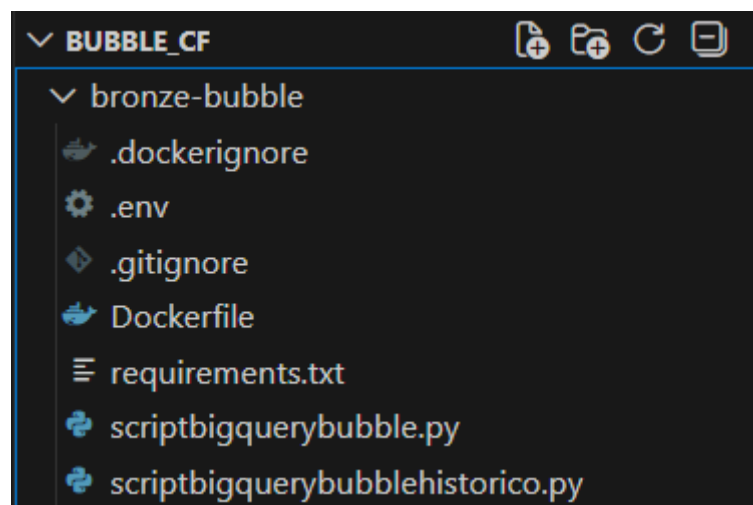
A estrutura mínima obrigatória é:

```
/seu-projeto
├── main.py      # Script principal de coleta
├── requirements.txt # Bibliotecas Python
└── Dockerfile   # Receita de empacotamento
```

Como o código será versionado no GitHub, você também deve criar:

- **.env** → credenciais e variáveis
- **.gitignore** → impedir envio das credenciais

No seu exemplo real, a estrutura ficou assim.



Etapa 2 — Teste Local

Antes de empacotar qualquer coisa:

1. Configure o `.env`
2. No `main.py`, use:

```
from dotenv import load_dotenv
load_dotenv()
```

3. Execute:

```
python nomedoarquivo.py
```

Só siga adiante se os dados estiverem sendo gravados corretamente no bucket.

Etapa 3 — Criar Repositório Docker no GCP

Se ainda não existir:

```
gcloud artifacts repositories create NOME-DO-REPO \
--repository-format=docker \
--location=us-east4 \
--description="Repositorio de coletores"
```

Esse repositório será onde suas imagens Docker ficarão.

Etapa 4 — Criar o Bucket de Dados

No console do GCP:

- Vá em **Cloud Storage** → **Buckets**
- Crie o bucket
- Use o mesmo nome que o script usa

O código e o bucket **devem bater exatamente**.

Etapa 5 — Build e Push da Imagem

Na pasta do projeto:

```
gcloud builds submit --tag  
us-east4-docker.pkg.dev/SEU-PROJECT-ID/NOME-DO-REPO/NOME-DA-IMAGEM:latest .
```

Isso faz:

- Build do Dockerfile
- Push da imagem para o Artifact Registry

Etapa 6 — Criar o Cloud Run Job

Agora criamos a Job que rodará esse código:

```
gcloud run jobs deploy nome-do-job --image  
us-east4-docker.pkg.dev/SEU-PROJECT-ID/NOME-DO-REPO/NOME-DA-IMAGEM:latest  
--region us-east4
```

Se precisar atualizar o código:

```
gcloud run jobs update nome-do-job --image  
us-east4-docker.pkg.dev/SEU-PROJECT-ID/NOME-DO-REPO/NOME-DA-IMAGEM:latest  
--region us-east4
```

Etapa 7 — Agendar a Job




No Console:

- Vá em **Cloud Run** → **Jobs**
- Clique na Job
- Vá em **Triggers**
- Clique em **Add Scheduler Trigger**
- Defina a frequência (ex: diário)

Job: joblancpedido Region: us-central1 Last updated: Dec 31, 2025, 2:55:07 AM

History Observability **Triggers** YAML

Triggers [+ Add scheduler trigger](#)

 joblancpedido-scheduler-trigger  

Schedule
0 */2 ***

Timezone
Etc/UTC

Region
us-central1

[View in Cloud Scheduler](#)

A partir daqui:

A Job Diária está ativa e salvando arquivos Parquet no bucket automaticamente.

Etapa 8 — Carga Histórica

Agora fazemos o bootstrap.

1. Copie o script da Job diária
2. Altere para:
 - Buscar da data inicial até hoje
3. Rode **localmente** (lembrando de usar o `load_dotenv()`)

Depois:

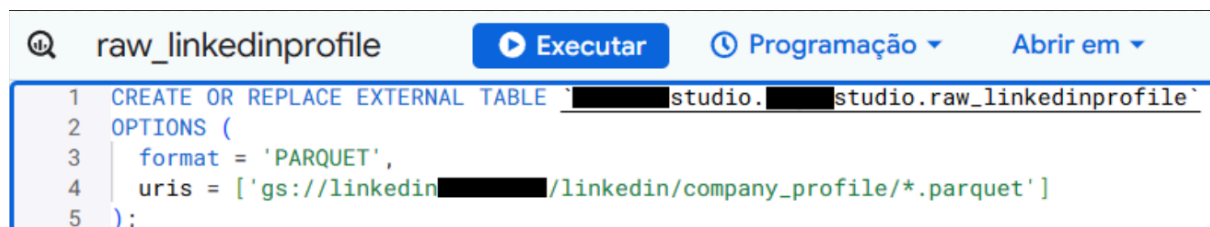
- Vá no bucket
- Verifique se múltiplos dias foram criados

Se sim, o histórico está carregado.

Etapa 9 — Conectar o BigQuery aos Parquets

Agora criamos uma **External Table**:

```
CREATE OR REPLACE EXTERNAL TABLE `fricarne-api.novaeng.raw_bubble`  
OPTIONS (  
  format = 'PARQUET',  
  uris = [  
  
'gs://seu-bucket-bronze/bubble/lancamento_pedidos/bubble/lancamento_pedidos/*.parquet'  
  ]  
)
```



O * indica: “Leia todos os anos, meses e dias automaticamente.”

Disclaimer – Erros de Autenticação (403) e Secrets no Cloud Run Job

Em ambientes reais de produção, é comum que **Cloud Run Jobs** eventualmente apresentem erros de autenticação, normalmente com mensagens do tipo:

403 – Permission denied

ou

Failed to access Secret

Mesmo quando:

- O Service Account está correto
- As permissões estão atribuídas
- O Secret existe

isso pode acontecer por problemas transitórios de IAM, Secret Manager ou resolução de identidade do Job.

Na prática, isso se manifesta como:

- O container inicia
- Mas falha ao tentar ler uma variável de ambiente vinda de um Secret
- E o Job aborta

Uma abordagem muito mais **estável operacionalmente** é **injetar os secrets diretamente no container do Cloud Run Job**, em vez de depender de resolução dinâmica via Secret Manager.

Isso é feito no próprio painel do Job:

Cloud Run → Jobs → Sua Job → Edit → Variáveis e secrets

Ali você pode definir:

Nome da variável	Valor
OMIE_APP_KEY	xxxxxx
OMIE_APP_SECRET	yyyyyy
DB_PASSWORD	zzzzzz

Essas variáveis ficam:

- Criptografadas
- Isoladas por Job
- Injetadas no container no startup

Na prática, o container passa a enxergar essas variáveis como se viessem de um `.env`.

Isso elimina:

- Dependência do Secret Manager em runtime
- Problemas de IAM intermitentes
- Falhas 403 imprevisíveis

Para pipelines de ingestão de dados, secrets que não mudam frequentemente podem ser configurados diretamente como variáveis no Cloud Run Job, pois isso elimina dependências em tempo de execução do Secret Manager, evita falhas intermitentes de IAM (como erros 403) e torna o comportamento do container mais previsível e estável em produção.

O uso de Secret Manager é mais importante em cenários em que é realmente necessário rotacionar segredos ou compartilhar credenciais entre múltiplos serviços.