Guia de Implementação de Transformações com DBT no GCP

Este documento descreve, de forma **estruturada, prática e padronizada**, como implementar a camada de **transformação de dados com DBT** no ecossistema GCP.

O foco é:

- Escalar desenvolvimento para o time
- Garantir padrão entre projetos
- Tratar DBT como **componente de produção**, não ferramenta local

Este guia **não ensina SQL** e **não cobre ingestão**.
Ele parte do pressuposto que os dados já existem no BigQuery (RAW / Bronze).

Visão Geral deste Fluxo

Antes de entrar em comandos, é fundamental entender **onde o DBT entra** no pipeline.

O fluxo completo funciona assim:

1. **Ingestão (fora do DBT)**
    - Cloud Run Jobs coletam dados
    - Arquivos Parquet são gravados no GCS
    - External Tables ou tabelas RAW são expostas no BigQuery
2. **Transformação (DBT)**
    - DBT lê dados RAW / Bronze
    - Aplica regras técnicas e de negócio
    - Cria tabelas Silver e Gold no BigQuery
3. **Consumo**
    - Dashboards
    - APIs
    - Modelos analíticos
    - IA / RAG

> **DBT não coleta dados, não chama API e não grava em GCS.**
> DBT apenas transforma dados **já disponíveis no BigQuery**.

Etapa 1 — Papel do DBT no Pipeline

O DBT é responsável por **três coisas apenas**:

Contrato de Dados

- Definir schemas
- Tipar colunas
- Garantir consistência entre execuções

Regras de Negócio

- Normalizações
- Flags
- Cálculos
- Métricas confiáveis

Confiabilidade

- Testes
- Detecção de erro antes do consumo
- Previsibilidade em produção

O DBT **não é**:

- Ferramenta exploratória
- Lugar de “SELECT *”
- Lugar de lógica improvisada

Se a transformação **quebra regra de negócio**, ela **não sobe**.

Etapa 2 — Estrutura Padrão de Projeto DBT

Todo projeto DBT **segue a mesma estrutura**.

Estrutura mínima obrigatória:

```
/meu_projeto_dbt
├── dbt_project.yml
├── profiles.yml
├── models/
│   ├── sources/
│   ├── silver/
│   └── gold/
├── macros/
├── tests/
└── docker/
```

models/sources

- Apenas definição de fontes
- Nenhuma transformação
- Espelha o RAW / Bronze

models/silver

- Limpeza
- Tipagem
- Normalização
- Deduplicação
- Colunas técnicas

models/gold

- Fatos
- Dimensões
- Métricas finais
- Dados prontos para consumo

> **Nunca existe lógica de negócio em `sources`**
> **Nunca existe dado sujo em `gold`**

Etapa 3 — Setup Local do Ambiente DBT

Antes de rodar qualquer coisa em produção, **o DBT deve funcionar localmente**.

1️⃣ Criar ambiente virtual

python -m venv dbt_env
source dbt_env/bin/activate

No Windows:

.\dbt_env\Scripts\activate

2️⃣ Instalar DBT para BigQuery

pip install dbt-bigquery

3️⃣ Inicializar o projeto

dbt init meu_projeto_dbt
cd meu_projeto_dbt

Durante o init:

- Escolha BigQuery
- Use o project-id correto
- Dataset default pode ser temporário

4️⃣ Autenticação com GCP

gcloud auth login
gcloud auth application-default login

Validações obrigatórias:

gcloud auth list
gcloud config get-value project

Se o projeto estiver errado:

gcloud config set project ID_DO_PROJETO

5️⃣ Teste de conexão

dbt debug

⚠️ **Só continue se o `dbt debug` retornar sucesso.**

Se falhar:

- Pare
- Corrija credenciais
- Corrija project-id
- Corrija dataset

Não avance com DBT quebrado localmente.

Etapa 4 — Definição de Sources (Contrato com o Bronze)

Antes de qualquer transformação, o DBT **precisa saber exatamente de onde os dados vêm**.

Toda tabela RAW / Bronze entra no DBT como source.

Estrutura obrigatória

```
models/
└── sources/
    └── fonte_x/
        └── fonte_x_sources.yml
```

Exemplo de source

```
version: 2

sources:
  - name: bubble
    database: meu_project_id
    schema: bronze_bubble
    tables:
      - name: lancamento_pedidos
        description: "Pedidos brutos vindos da API Bubble"
```

Boas Práticas

- Source **nunca transforma**
- Source **nunca filtra**
- Source **nunca cria regra**
- Source é **contrato**, não código

Se a tabela existe no BigQuery e não está em `sources`, ela **não deveria ser usada**.

Etapa 5 — Modelagem Silver (onde o dado vira confiável)

A camada Silver é a **mais importante do DBT**.

É aqui que:

- o dado deixa de ser cru
- o erro aparece
- a qualidade é imposta

Responsabilidades da Silver

- Tipagem explícita
- Normalização de campos
- Deduplicação
- Colunas técnicas
- Padronização de nomes

Estrutura

```
models/
└── silver/
    └── fonte_x/
        └── silver_fonte_x__entidade.sql
```

Exemplo de modelo Silver

```
with source as (

    select *
    from {{ source('bubble', 'lancamento_pedidos') }}

),

typed as (

    select
        cast(id as int64) as pedido_id,
        cast(valor as numeric) as valor_pedido,
        cast(data_criacao as timestamp) as dt_criacao,
        current_timestamp() as dt_ingest
    from source

),

final as (

    select
        pedido_id,
        valor_pedido,
```

```
        dt_criacao,
        dt_ingest
    from typed

)

select * from final
```

Boas Práticas

> **Na última CTE, TODAS as colunas devem estar explicitamente definidas.**
> Nada de `select *` no final.

Etapa 6 — Deduplicação

Deduplicação **mal feita** gera exatamente os mesmos problemas de incremental mal feito:

- duplicação silenciosa
- dados **aparentemente corretos**, mas errados
- perda de confiança no dado
- bugs difíceis de rastrear no Gold e nos dashboards

A **deduplicação não é efeito colateral**, é **regra explícita de modelagem**, especialmente na camada Silver.

Princípio de Deduplicação

Sempre que uma fonte puder reenviar registros, a Silver DEVE deduplicar.

Isso vale para:

- APIs
- arquivos reprocessados
- cargas históricas
- fontes que enviam “snapshots” do estado atual

Não importa se:

- a API “promete” não duplicar
- o fornecedor “garante” unicidade
- o dado “parece confiável”

Se existe reenvio, **existe deduplicação**.

Estratégia Padrão: Último Estado por Chave

A estratégia padrão é:

- **1 registro final por chave de negócio**
- mantendo **o mais recente** com base no timestamp de extração
- Deduplicar antes

Isso é feito com:

- `ROW_NUMBER()`
- `PARTITION BY chave_de_negocio`
- `ORDER BY dt_extract DESC`

Estrutura Conceitual da Deduplicação

O fluxo **sempre segue este padrão**:

1. Ler a source (RAW / Bronze)
2. Normalizar e tipar os dados

3. Criar uma CTE exclusiva de deduplicação
4. Filtrar explicitamente `row_num = 1`
5. Nunca propagar `row_num` para fora

Exemplo de Deduplicação

```
deduplicated AS (
  SELECT
    *,
    ROW_NUMBER() OVER (
      PARTITION BY solicitacao_id
      ORDER BY dt_extract_utc DESC
    ) AS row_num
  FROM renamed
  WHERE solicitacao_id IS NOT NULL
)
```

E **listando as colunas** no SELECT final:

```
SELECT
  solicitacao_id,
  empresa_id,
  data_prazo,
  data_hora_ultima_atualizacao,
  responsaveis_escritorio,
  responsaveis_empresa,
  dt_extract_utc
FROM deduplicated
WHERE row_num = 1
```

Sobre a Chave de Deduplicação

A chave usada no `PARTITION BY` deve ser:

- identificador lógico do negócio
- estável ao longo do tempo
- independente do processo de ingestão

Exemplos:

- `pedido_id`
- `solicitacao_id`
- `cliente_id + data_evento` (quando necessário)

Nunca usar:

- `run_id`
- `dt_ingest`
- timestamp técnico como única chave

Quando Deduplicar (Obrigatório)

- APIs que retornam histórico completo
- APIs que retornam "estado atual"
- Fontes que reenviam registros atualizados
- Cargas reprocessáveis
- Bases onde o `dt_extract` varia

Quando NÃO Deduplicar

- Eventos puramente append-only
- Logs imutáveis
- Streams com chave garantidamente única por evento

Mesmo assim, se houver dúvida, **deduplique**.

Erros Clássicos (e proibidos)

- Deduplicar no Gold
- Deduplicar no dashboard
- Deduplicar "na ingestão" sem deixar rastro
- Assumir unicidade sem teste
- Não documentar o critério de deduplicação

Deduplicar Antes de Tipar (ordem importa)

Sempre que possível, a deduplicação deve ocorrer **antes** da tipagem e das transformações mais pesadas da Silver.

Tipar, parsear datas, normalizar strings e criar arrays **em registros que serão descartados**:

- aumenta custo de processamento
- aumenta tempo de execução
- gera trabalho inútil no BigQuery

O papel da deduplicação inicial é **reduzir o volume de linhas** para que apenas **o registro sobrevivente** passe pelas transformações completas.

Não faz sentido tipar dado que vai morrer.

Quando a fonte permite:

1. deduplicar por chave + `dt_extract`
2. manter apenas o registro vencedor
3. aplicar tipagem e normalização **apenas nele**

Isso torna a Silver:

- mais barata
- mais rápida
- mais previsível

Boas Práticas

Silver sempre representa o último estado confiável do dado.

Se alguém perguntar: “Esse registro representa qual versão da informação?”

A resposta tem que ser: “A versão mais recente recebida da fonte, com base em `dt_extract_utc`.”

Se essa resposta não existir, **a modelagem está errada**.

Etapa 7 — Testes

Testes são o que permitem **escalar time sem caos**.

Sem testes:

- erros passam despercebidos
- duplicações chegam no Gold
- dashboards perdem credibilidade
- bugs aparecem tarde (e caros)

Com testes:

- erros quebram o pipeline cedo
- o time trabalha com segurança
- mudanças ficam previsíveis
- confiança no dado é mantida

Modelo sem testes robustos não é modelo pronto.

Papel dos Testes no DBT

Testes servem para:

- validar contratos de dados
- garantir unicidade após deduplicação
- impedir regressões silenciosas
- proteger modelos Gold
- permitir refactor sem medo

Teste **não é auditoria**, é **barreira de entrada**.

Testes Mínimos Obrigatórios

Todo modelo Silver **deve** ter, no mínimo:

- `not_null`

- `unique`
- `accepted_values` (quando aplicável)

Esses testes garantem:

- integridade de chaves
- consistência de status e flags
- previsibilidade em joins

Exemplo de `schema.yml`

```
version: 2

models:
  - name: silver_bubble__pedidos
    description: "Pedidos deduplicados e normalizados da fonte Bubble"
    columns:
      - name: pedido_id
        description: "Identificador único do pedido"
        tests:
          - not_null
          - unique
```

Se um campo representa chave de negócio, ele **obrigatoriamente** tem `not_null` + `unique`.

Testes de Domínio (quando aplicável)

Campos categóricos **devem ser protegidos**.

Exemplo:

```
- name: status
  tests:
    - accepted_values:
        values: ['aberto', 'em_andamento', 'encerrado']
```

Isso evita:

- variações inesperadas

- erros de digitação na fonte
- quebra de métricas no Gold

Outros Tipos de Testes (além do básico)

Além dos testes nativos do DBT, existem diversos testes prontos disponíveis em pacotes como `dbt_utils`, que ajudam a validar:

- relacionamentos (`relationships`)
- combinações de colunas
- ranges de valores
- consistência entre tabelas

Além disso, devemos criar testes customizados, sempre que:

- a regra de negócio não for genérica
- a validação não existir em pacotes prontos
- o erro for crítico para o consumo

Quando a regra é de negócio, o teste também é.

Testes prontos aceleram. Testes customizados protegem o que é específico do cliente.

Execução Real no Dia a Dia

Testes **não rodam uma vez**. Eles fazem parte do fluxo operacional.

```
dbt test --select path:models/bubble
dbt test --select path:models/bubble/silver
```

Regras:

- rodar testes após mudanças
- rodar testes antes de deploy
- rodar testes após ingestões novas

Como Encarar Falhas de Teste

Teste quebrando significa:

- o dado mudou
- a fonte quebrou contrato
- ou a modelagem está errada

Teste quebrado bloqueia avanço.

Etapa 8 — Camada Gold (dados prontos para decisão)

A Gold **não limpa dado**.
Ela **organiza decisão**.

Responsabilidades

- Fatos
- Dimensões
- Métricas
- Flags finais

Estrutura

```
models/
└── gold/
    ├── facts/
    └── dimensions/
```

Exemplo de fato

```
select
    pedido_id,
    valor_pedido,
    dt_criacao,
    extract(month from dt_criacao) as mes
from {{ ref('silver_bubble__pedidos') }}
```

Etapa 9 — DBT como Serviço (Docker + Cloud Run Jobs)

DBT **não roda só no notebook**.

Ele roda como:

- Job
- Serviço
- Pipeline oficial

Dockerfile padrão

```
FROM python:3.11-slim

WORKDIR /app

COPY . .

RUN pip install dbt-bigquery

ENTRYPOINT ["dbt"]
```

Build e Push

```
docker build -t dbt-FONTE:latest -f docker/FONTE/Dockerfile .

docker tag dbt-FONTE:latest
us-east1-docker.pkg.dev/berlim-studio/repo-dbt-FONTE/dbt-FONTE:latest

docker push us-east1-docker.pkg.dev/berlim-studio/repo-dbt-FONTE/dbt-FONTE:latest


docker push us-east1-docker.pkg.dev/PROJECT/repo/dbt-fonte:latest
```

Atualizar Job

```
gcloud run jobs update job-dbt-fonte \
--image us-east1-docker.pkg.dev/PROJECT/repo/dbt-fonte:latest \
--region us-east1
```

Etapa 10 — Execução Operacional

Esta etapa descreve **como o DBT é usado no dia a dia**, tanto localmente quanto em produção, e como lidar com execução, falhas e correções.

O foco aqui é **operar**, não desenvolver.

Comandos Reais Usados no Dia a Dia

Execução por fonte ou domínio específico:

- `dbt run --select path:models/googleads`

Execução direta de um modelo (ou conjunto pequeno):

- `dbt run -s silver_entregas`

Execução completa de testes após transformação:

- `dbt test`

Boas práticas:

- rodar `dbt run` sempre de forma **segmentada** quando possível
- evitar `dbt run` full sem necessidade
- testar antes de subir mudanças para produção

Execução em Produção (Cloud Run Job)

Em produção, o DBT é executado via **Cloud Run Job**, garantindo:

- ambiente controlado
- reprodutibilidade
- isolamento de dependências
- histórico de execuções

A execução segue sempre este ciclo:

1. Build da imagem Docker
2. Atualização da Job
3. Execução manual ou agendada
4. Análise de logs

Logs e Observabilidade

Quando algo falha, **o caminho de debug é sempre o mesmo**:

- **Cloud Run Logs**
    - erro de execução
    - falha de container
    - problema de ambiente
- **BigQuery Logs**
    - erro de SQL
    - problema de permissão
    - custo inesperado
- **DBT Logs**
    - modelo que quebrou
    - teste que falhou
    - dependência não resolvida

Nunca debugar “no escuro”. Sempre começar pelos logs.

Falhas Comuns em Execução

Alguns cenários esperados:

- teste quebrando após mudança na fonte
- erro de tipagem por dado inesperado

- falha de incremental mal configurado

Essas falhas **não são exceção**, são parte da operação.

O importante é:

- identificar rápido
- corrigir com critério
- reexecutar com segurança

Estratégia de Rollback

Rollback no DBT é **simples e controlado**, quando feito corretamente.

Opções padrão:

- **Rebuild da tabela**
    - drop + recriação
    - usado quando incremental ficou inconsistente
- **Ajuste no modelo**
    - correção de SQL
    - ajuste de regra de negócio
    - reforço de testes
- **Reexecução da Job**
    - mesma imagem
    - mesma lógica
    - resultado previsível

Rollback deve ser consciente, nunca impulsivo.

Mentalidade Operacional

Operar DBT em produção significa entender que:

- erro cedo é bom

- teste quebrando protege o dado
- rollback faz parte do fluxo
- previsibilidade é mais importante que velocidade

DBT não é “rodar comando”. DBT é **manter a confiança do dado ao longo do tempo**.
