



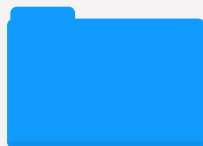
Apache  
**Airflow**®

Python Brasil 2025

Edson(edinho)



Porquê?  
Quando?  
Quando não?



Arquitetura



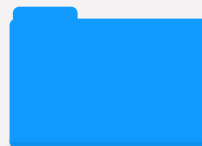
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



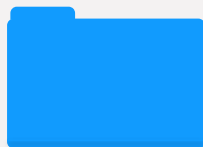
Zoologico de  
Operadores



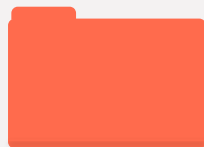
Padrões de  
Pipelines



Porquê?  
Quando?  
Quando não?



Arquitetura



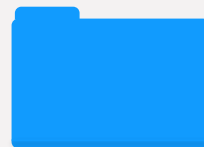
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores



Padrões de  
Pipelines



# Porquê?

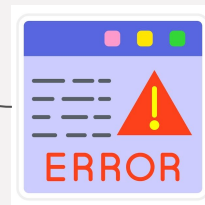
```
1 def pipeline():  
2     dados_brutos = download()  
3     dados_limpos = transformar(dados_brutos)  
4     with banco_de_dados.criar_sessao() as sessao:  
5         salvar_dados(sessao, dados_limpos)  
6         sessao.commit()
```

E  
T  
L



# Porquê?

```
1 def pipeline():  
2     dados_brutos = download()  
3     dados_limpos = transformar(dados_brutos)  
4     with banco_de_dados.criar_sessao() as sessao:  
5         salvar_dados(sessao, dados_limpos)  
6         sessao.commit()
```



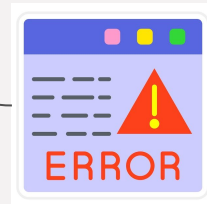
# Porquê?

```
1 def pipeline():  
2     dados_brutos = download()  
3     dados_limpos = transformar(dados_brutos)  
4     with banco_de_dados.criar_sessao() as sessao:  
5         salvar_dados(sessao, dados_limpos)  
6         sessao.commit()
```



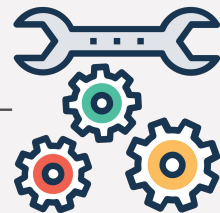
# Porquê?

```
1 def pipeline():
2     if dados_ja_foram_salvos():
3         dados_brutos = open('dados.csv').read()
4     else:
5         dados_brutos = download()
6         salvar_no_disco(dados_brutos, 'dados.csv')
7
8     dados_limpos = transformar(dados_brutos)
9     with banco_de_dados.criar_sessao() as sessao:
10         salvar_dados(sessao, dados_limpos)
11         sessao.commit()
```



# Porquê?

```
1 def pipeline(incremental: bool):  
2     dados_brutos = download(incremental)  
3     dados_limpos = transformar(dados_brutos)  
4     with banco_de_dados.criar_sessao() as sessao:  
5         salvar_dados(sessao, dados_limpos, incremental)  
6         sessao.commit()
```



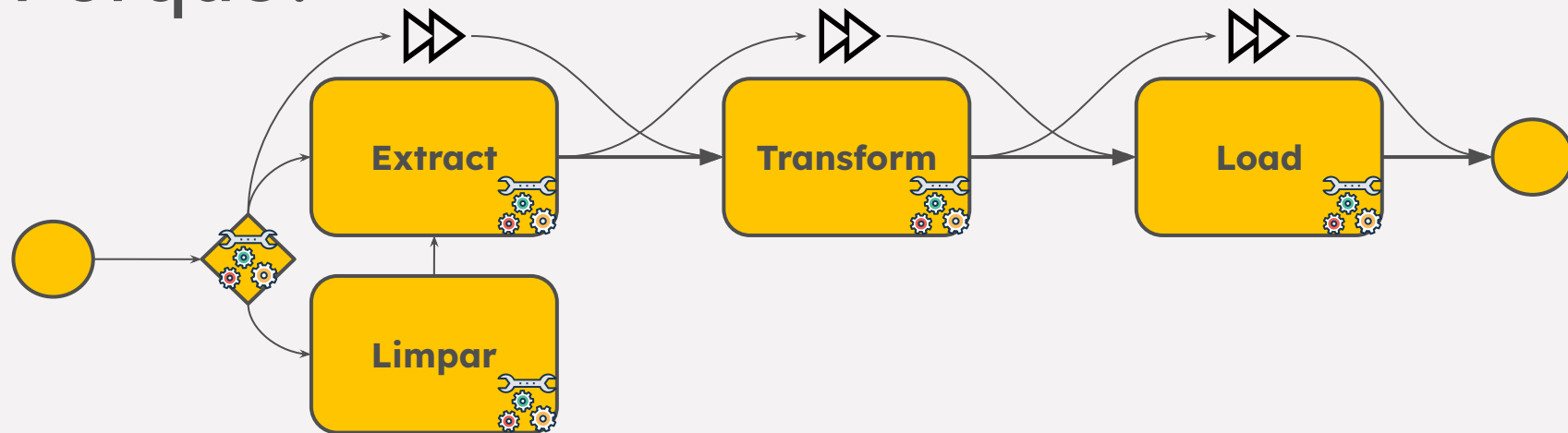
Configurações



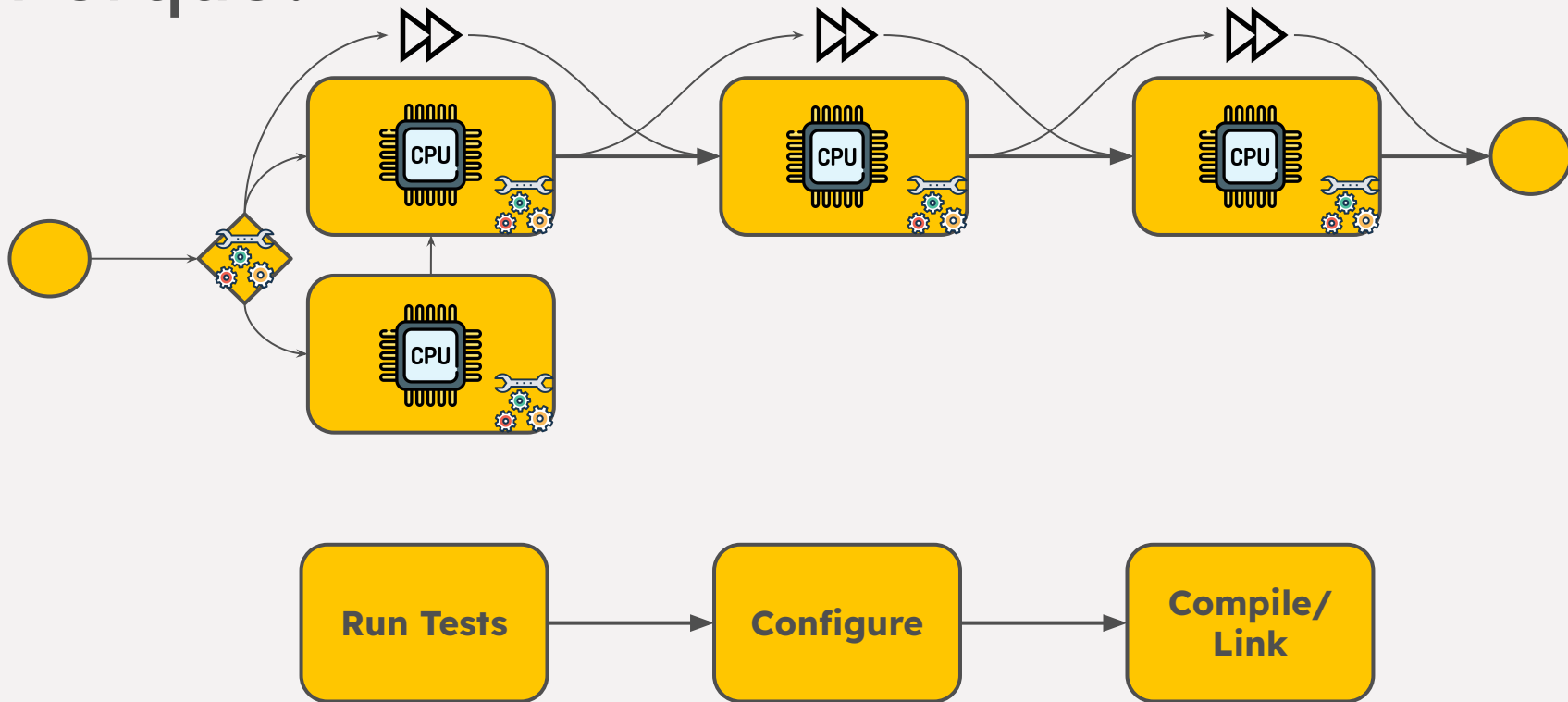
# Porquê?

```
1 def pipeline(incremental: bool, reset: bool):  
2     if reset and existe_dados():  
3         remover_dados_existentes()  
4         incremental = False  
5  
6     dados_brutos = download(incremental)  
7     dados_limpos = transformar(dados_brutos)  
8     with banco_de_dados.criar_sessao() as sessao:  
9         salvar_dados(sessao, dados_limpos, incremental)  
10        sessao.commit()
```

# Porquê?



# Porquê?



# Porquê?

No desenvolvimento de software, make é um utilitário que compila automaticamente programas e bibliotecas do arquivo fonte através da leitura de instruções contidas em arquivos denominados Makefiles, que especificam como obter o programa de destino. O make pode decidir por onde começar através de uma ordenação topológica. Ele também é capaz de resolver automaticamente as dependências do programa que se pretende compilar.

<https://pt.wikipedia.org/wiki/Make>

# Porquê?

No desenvolvimento de software, make é um utilitário que compila automaticamente programas e bibliotecas do arquivo fonte através da **leitura de instruções contidas em arquivos** denominados Makefiles, que especificam como obter o programa de destino. O make pode decidir por onde começar através de uma ordenação topológica. Ele também é capaz de resolver automaticamente as dependências do programa que se pretende compilar.

<https://pt.wikipedia.org/wiki/Make>

# Porquê?

No desenvolvimento de software, make é um utilitário que compila automaticamente programas e bibliotecas do arquivo fonte através da **leitura de instruções contidas em arquivos** denominados Makefiles, que especificam como obter o programa de destino. O **make pode decidir por onde começar** através de uma ordenação topológica. Ele também é capaz de resolver automaticamente as dependências do programa que se pretende compilar.

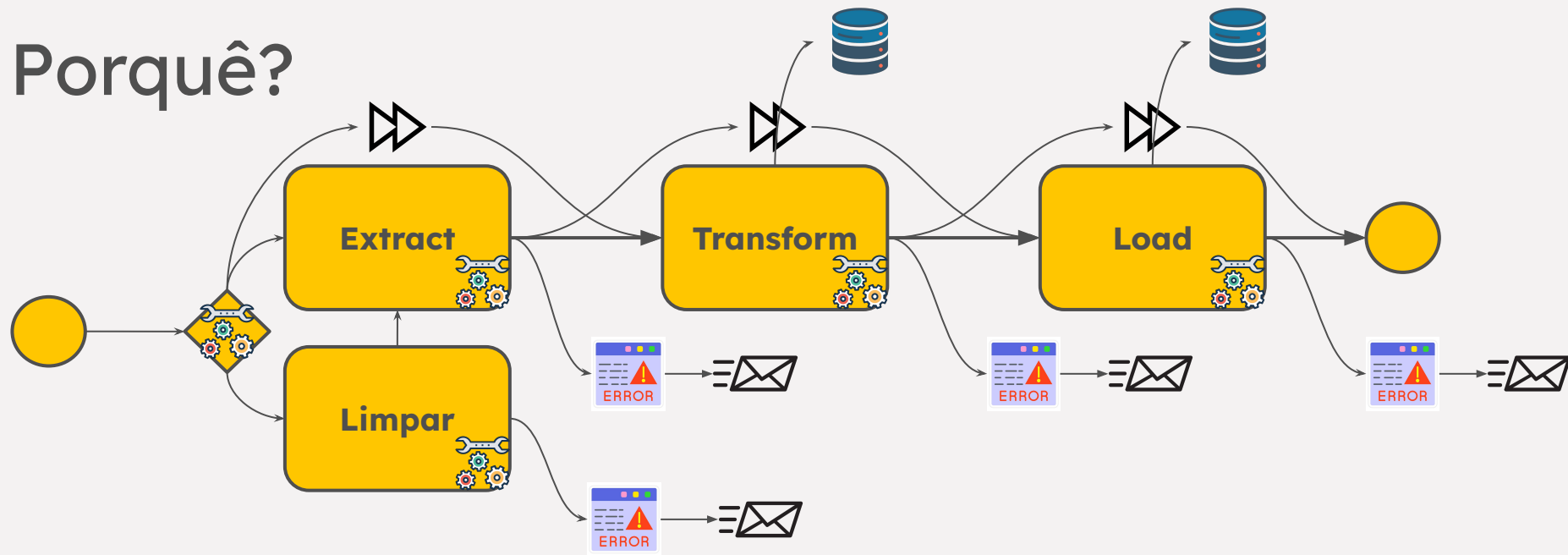
<https://pt.wikipedia.org/wiki/Make>

# Porquê?

No desenvolvimento de software, make é um utilitário que compila automaticamente programas e bibliotecas do arquivo fonte através da **leitura de instruções contidas em arquivos** denominados Makefiles, que especificam como obter o programa de destino. O **make pode decidir por onde começar** através de uma ordenação topológica. Ele também é capaz de **resolver automaticamente as dependências** do programa que se pretende compilar.

<https://pt.wikipedia.org/wiki/Make>

# Porquê?





# Porquê?



# Porquê?



**Luigi Task Status**

Task List Dependency Graph Workers Resources

Task Families: [Clear selection](#)

data\_management

PartisReport

- PartisReport.CombineReports
- PartisReport.DatedB0Report
- PartisReport.DownloadReport
- PartisReport.ExtractReport
- PartisReport.GeneratedFiles
- PartisReport.LoadTable
- PartisReport.LoadPartPlant
- PartisReport.LoadPartPlantB
- PartisReport.LoadParts
- PartisReport.DebateScrum

PFEF

BIOOMaterials

Others

- LoadTable

PENDING TASKS: 3

RUNNING TASKS: 1

BATCH RUNNING TASKS: 0

DONE TASKS: 134

FAILED TASKS: 3

UPSTREAM FAILURE: 1

DISABLED TASKS: 0

UPSTREAM DISABLED: 0

Displaying tasks of family **LoadTable**

Show 10 entries

	Name	Details	Priority	Time	Actions	
	RUNNING	LoadTable	table=FUP, date=2013-07-28	0	5/28/2013, 8:00:17 AM	

Showing 1 to 1 of 1 entries (filtered from 142 total entries)

Previous 1 Next

**Luigi Task Visualiser**

localhost:8082/static/visualiser/index.html#UserRecs(test=False, date=2013-07-24, re...

**Luigi Task Status** Active tasks

Task List Dependency Graph

TaskId(param1=va1,param2=va2) Show task details

UserRecs(test=False, date=2013-07-24, rec\_days=4, exp\_days=8, test\_users=False, force\_updates=False, build\_from\_scratch=True, index\_path=/spotify/discover/index, index\_version=None, FOLLOWS\_SCORE=5.0)

Dependency Graph

Legend: Failed (Red), Running (Blue), Pending (Yellow), Done (Green)

AggregateUserMatrices  
test=False  
date=2013-07-21  
index\_version=1363504343  
test\_users=False

# Porquê?



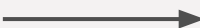
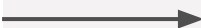
# Porquê?



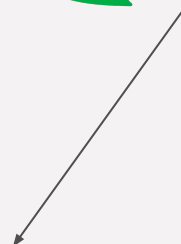
Apache  
**Airflow**<sup>®</sup>



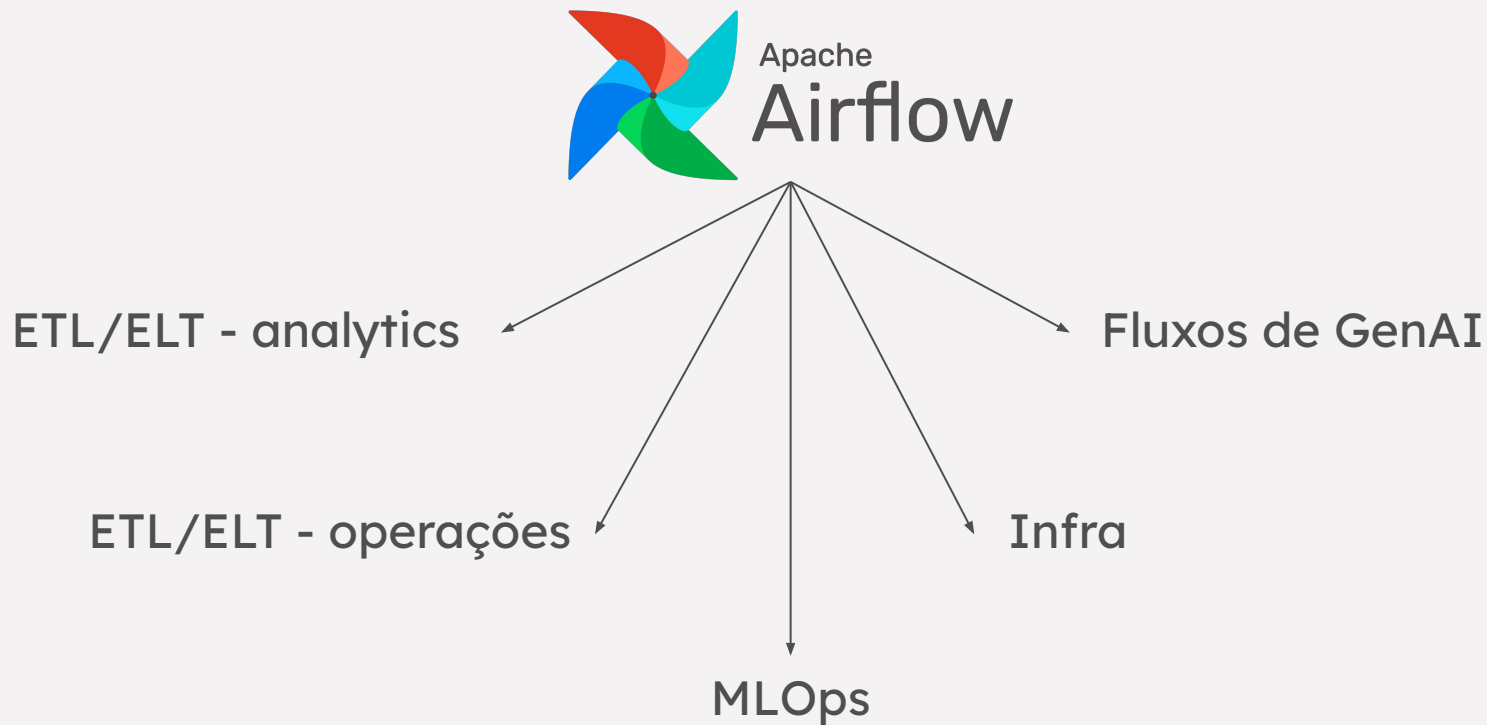
# Porquê?



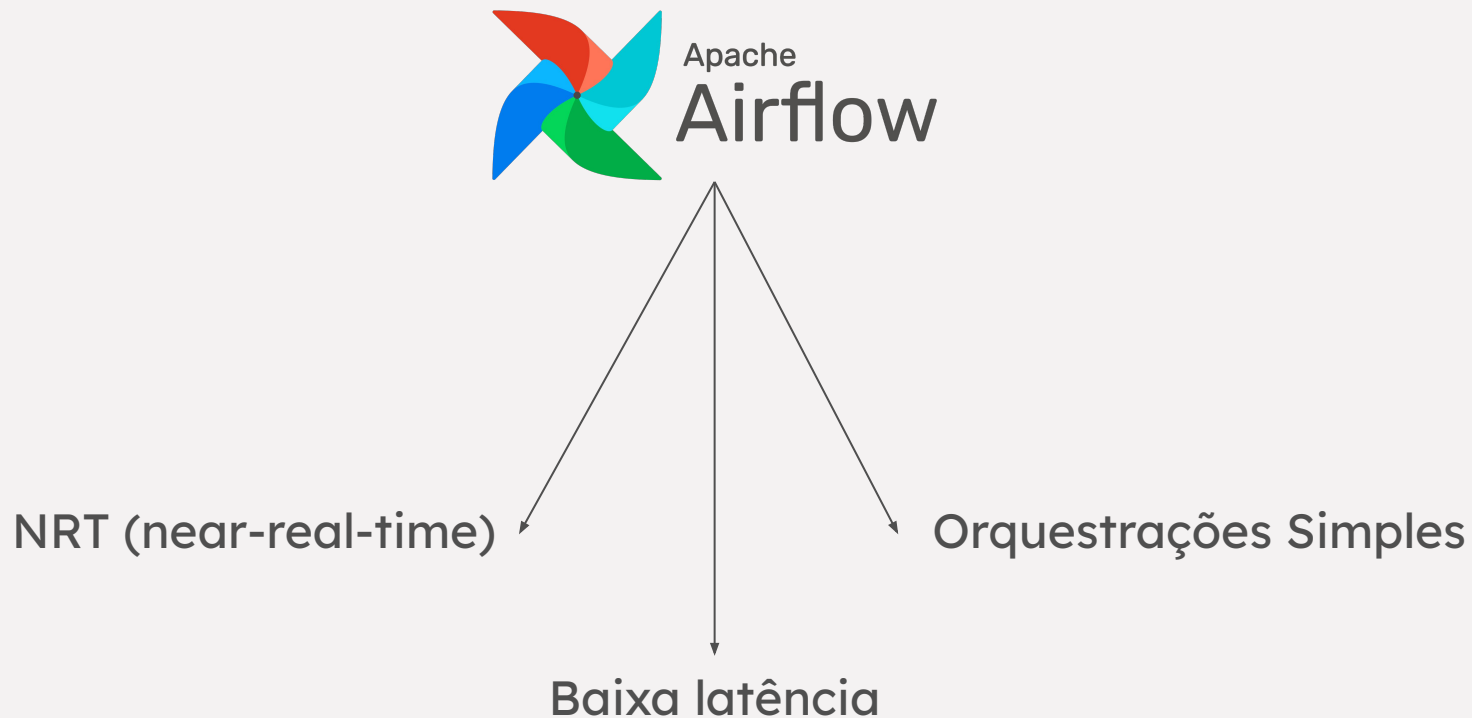
Apache  
Airflow



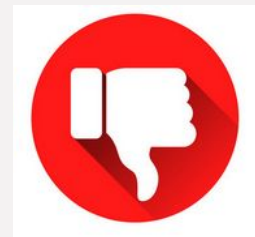
# Quando?



# Quando não?



# Quando?



ETL/ELT - analytics
ETL/ELT - operações
MLOps
Infra
Fluxos de GenAI

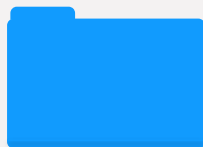
NRT (near-real-time)
Baixa latência
Orquestrações Simples



# Dúvidas?



Porquê?  
Quando?  
Quando não?



Arquitetura



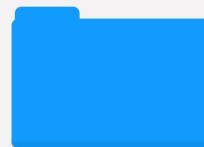
Criando um  
ambiente



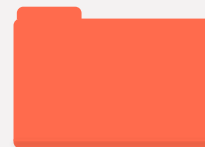
Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores

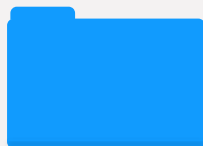


Padrões de  
Pipelines





Porquê?  
Quando?  
Quando não?



Arquitetura



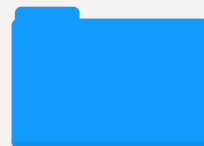
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores



Padrões de  
Pipelines



# Arquitetura



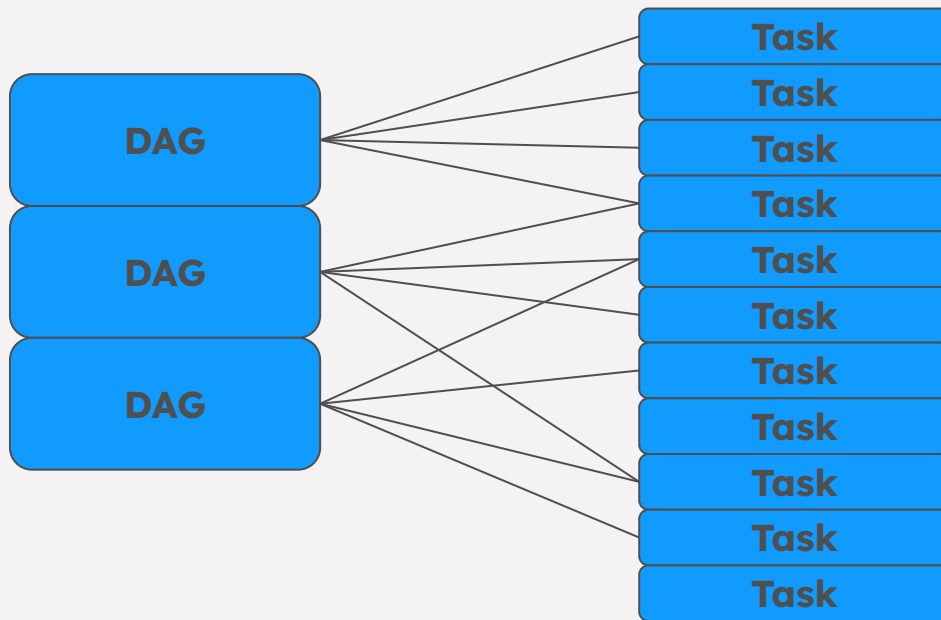
**DAG**

**Task**

**DagRun**

**TaskInstance**

# Arquitetura



# Arquitetura

The screenshot displays the Apache Airflow web interface. The left sidebar contains navigation links: Home, Dags (highlighted with a blue box and showing '80 Dags'), Assets, Browse, Admin, Docs, and User. The main content area is titled 'Dags' and includes a search bar, filters for status (All, Failed, Queued, Running, Success, Required Actions), and a 'Filter by tag' dropdown. The DAG list is sorted by 'Display Name (A-Z)'. The first four DAGs are highlighted with a blue box:

DAG Name	Latest Run	Next Run
Params Trigger UI example, params		
Params UI tutorial example, params, ui		
Sample DAG with Display Name example	2025-10-18 09:55:19	
asset1_producer		

# Arquitetura

[illegible]

# Arquitetura

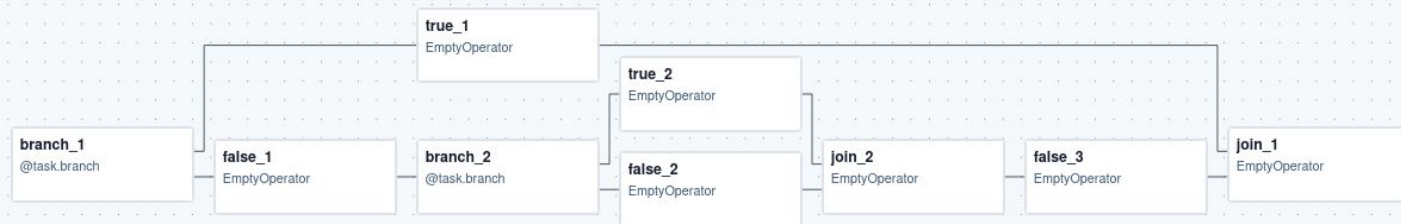
```
graph LR; task1["sample_task_1  
EmptyOperator"] --- task2["sample_task_2  
@task"]
```

**sample\_task\_1**  
EmptyOperator

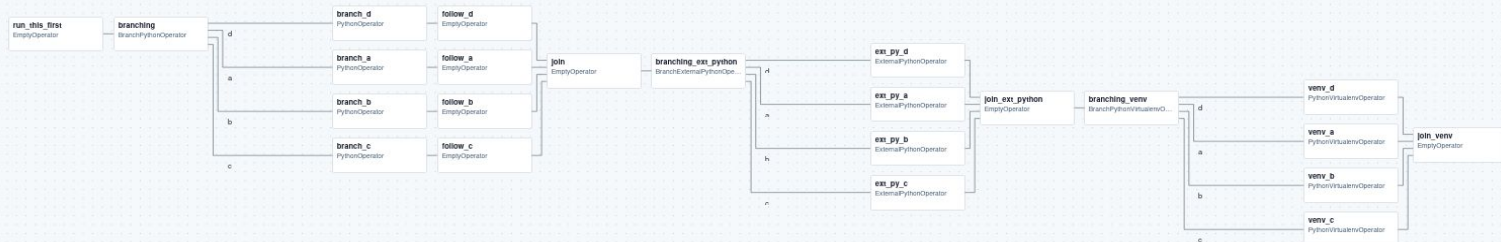
**sample\_task\_2**  
@task



# Arquitetura

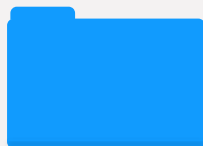


# Arquitetura





Porquê?  
Quando?  
Quando não?



Arquitetura



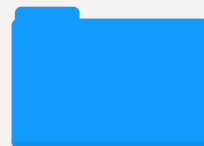
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores

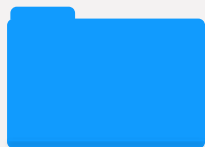


Padrões de  
Pipelines

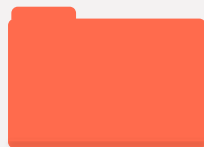




Porquê?  
Quando?  
Quando não?



Arquitetura



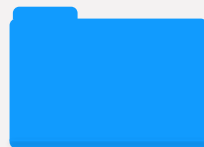
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores



Padrões de  
Pipelines



# Mão na massa...

```
$ git clone https://github.com/edinhodiluviano/airflow-pybr25.git
```

```
$ cd airflow-pybr25
```

```
$ ./venv-tool.sh
```

```
$ cp env_template .env
```

```
$ . activate
```

```
$ pip install uv
```

```
$ uv sync --frozen
```

# Mão na massa...

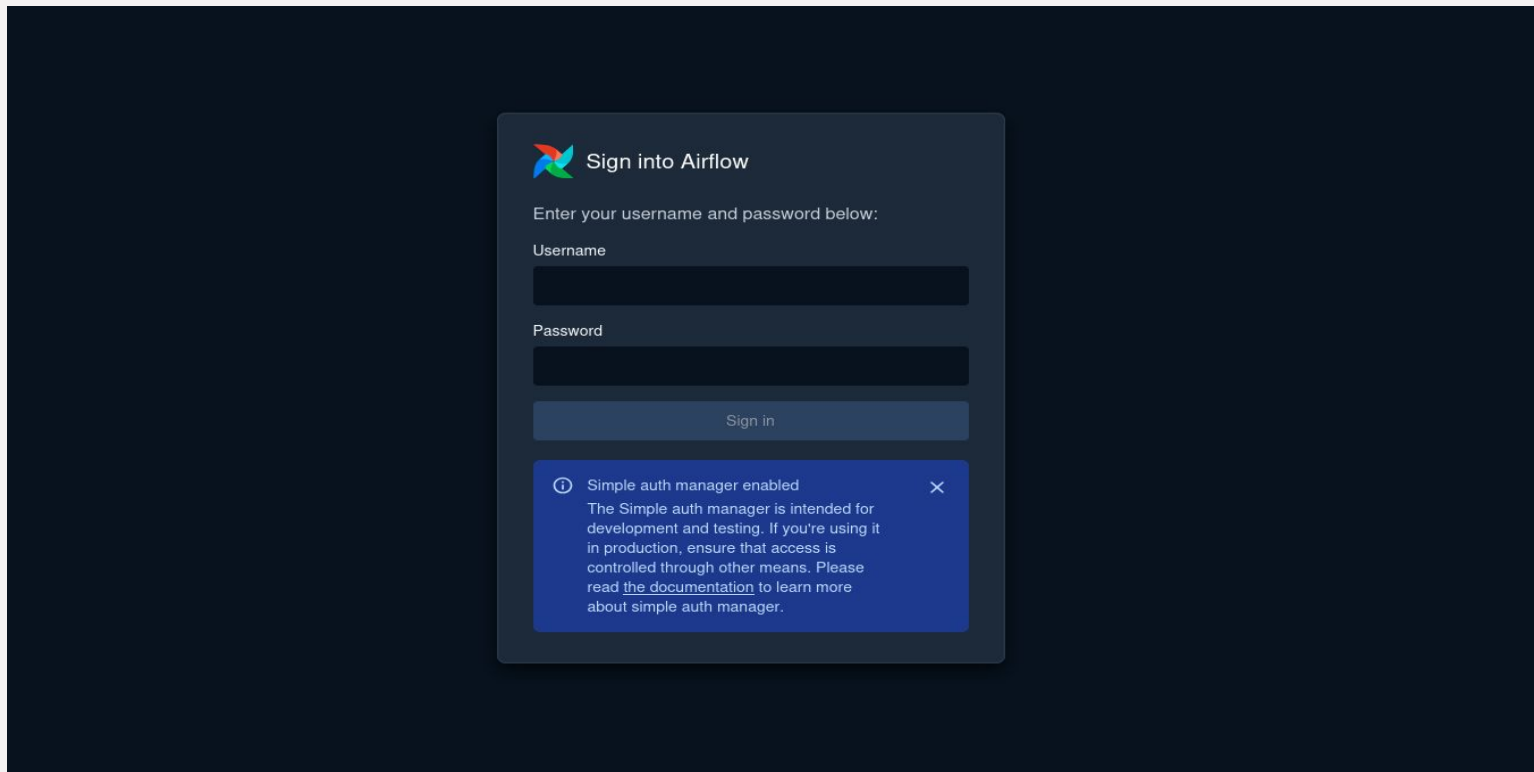
```
$ echo $AIRFLOW_HOME  
/tmp
```

```
$ airflow version  
3.1.0
```

# Mão na massa...

```
$ airflow standalone
```

# Mão na massa...

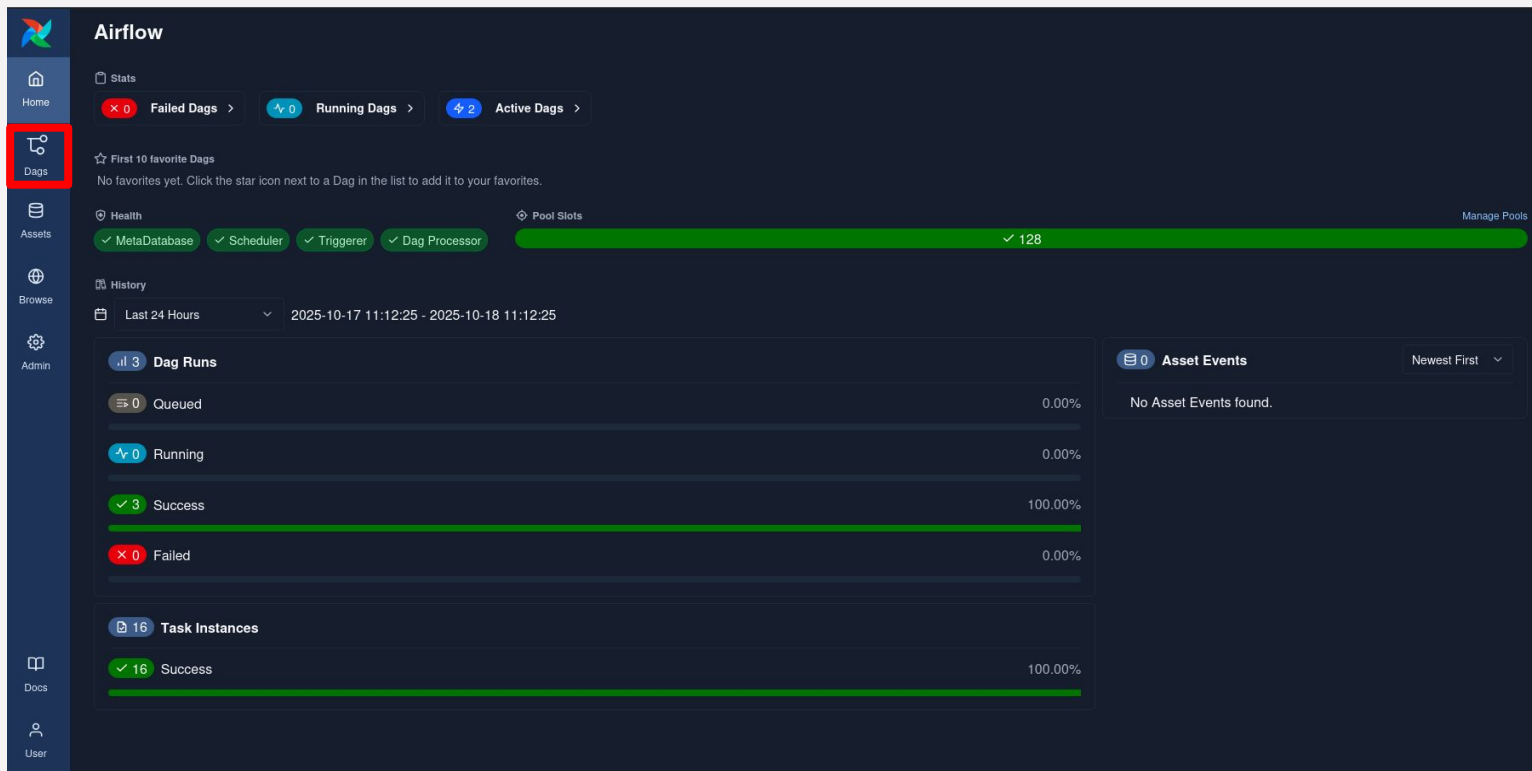




# Usuário e Senha

```
$ cat /tmp/simple_auth_manager_passwords.json.generated  
{"admin": "82WzERM6S7Fe9D3T"}
```

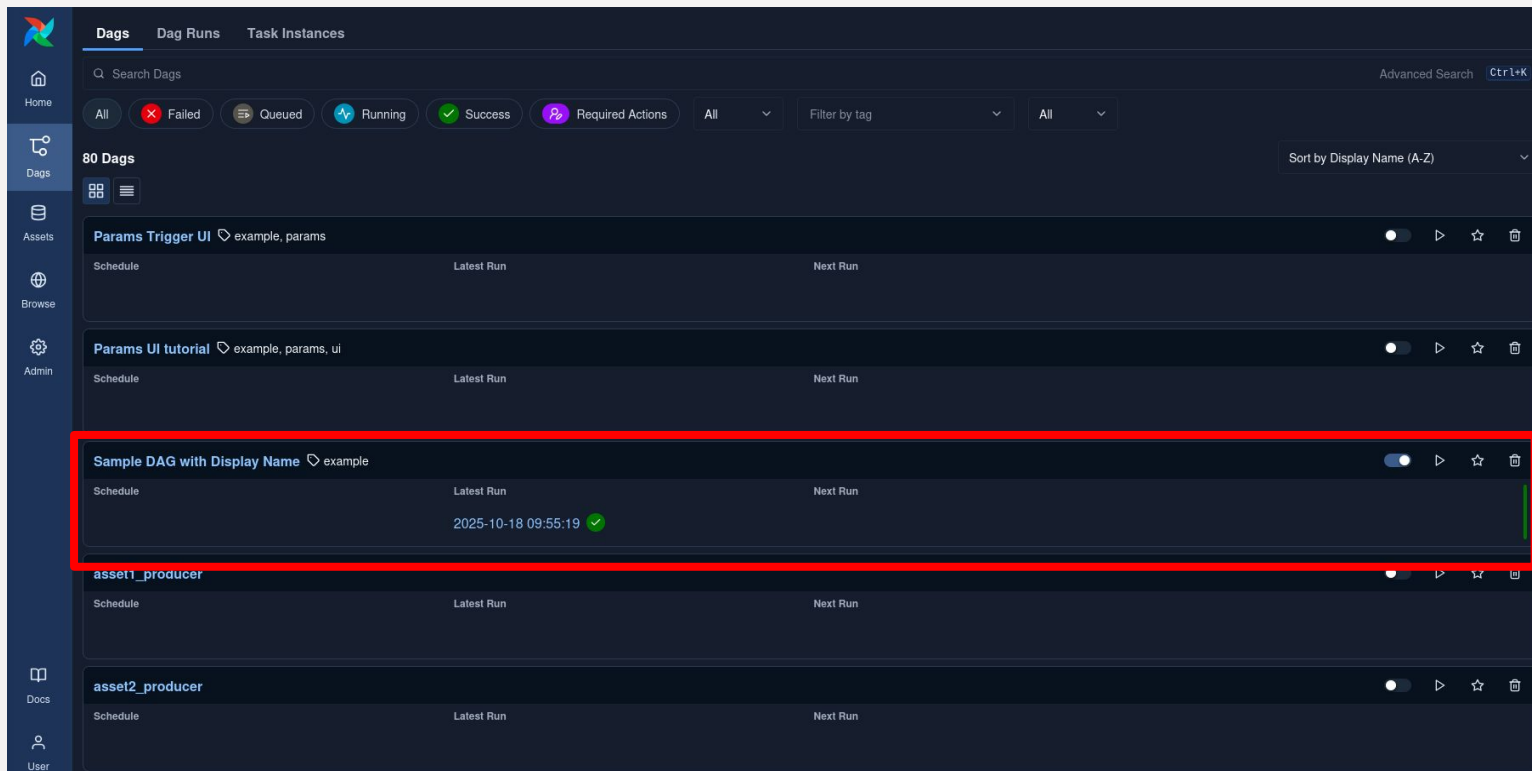
# Meu primeiro Airflow



The screenshot displays the Apache Airflow web interface. On the left sidebar, the 'Dags' tab is highlighted with a red box. The main content area shows the 'Airflow' dashboard with various sections:

- State:** Failed Dags (0), Running Dags (0), Active Dags (2).
- First 10 favorite Dags:** No favorites yet. Click the star icon next to a Dag in the list to add it to your favorites.
- Health:** MetaDatabase, Scheduler, Triggerer, Dag Processor. Pool Slots: 128.
- History:** Last 24 Hours, 2025-10-17 11:12:25 - 2025-10-18 11:12:25.
- Dag Runs:** 3 Queued (0.00%), 0 Running (0.00%), 3 Success (100.00%), 0 Failed (0.00%).
- Task Instances:** 16 Success (100.00%).
- Asset Events:** No Asset Events found.

# Meu primeiro Airflow



The screenshot displays the Apache Airflow web interface. The left sidebar contains navigation links: Home, Dags, Assets, Browse, Admin, Docs, and User. The main content area is titled 'Dags' and shows a list of DAGs. The 'Sample DAG with Display Name' is highlighted with a red box. Below the header, there are filters for 'All', 'Failed', 'Queued', 'Running', 'Success', and 'Required Actions'. The 'Sample DAG with Display Name' is listed with its latest run status as 'Success' on 2025-10-18 09:55:19. The interface also includes a search bar, a sort dropdown, and a table of DAGs with columns for Schedule, Latest Run, and Next Run.

DAG Name	Latest Run	Next Run
Params Trigger UI example, params		
Params UI tutorial example, params, ui		
Sample DAG with Display Name example	2025-10-18 09:55:19 <span>Success</span>	
asset1_producer		
asset2_producer		

# Meu primeiro Airflow

The screenshot displays the Apache Airflow web interface. On the left sidebar, the 'Dags' menu item is highlighted, and a red box highlights the 'DAG' icon. The main content area shows the details for the 'Sample DAG with Display Name'. A red box highlights the 'Trigger' button in the top right corner. The interface includes a table of DAGs, a table of runs, and a 'Last Dag Run' section with a bar chart showing 'Queued Duration' and 'Run Duration'.

**DAGs Table:**

DAG	Status
sample_task_1	✓
sample_task_2	✓

**Sample DAG with Display Name Details:**

- Schedule:** Latest Run: 2025-10-18 09:55:19 ✓
- Next Run:**
- Owner:** airflow
- Tags:** example
- Latest Dag Version:** v1

**Overview** | **Runs** | **Tasks** | **Calendar** | **Audit Log** | **Code** | **Details**

**Failed Tasks:** 0

**Failed Runs:** 0

**Last Dag Run:**

Duration: 3.22 (Run Duration), 1.00 (Queued Duration)

Run After: 2025-10-18 09:55:19

# Arquitetura



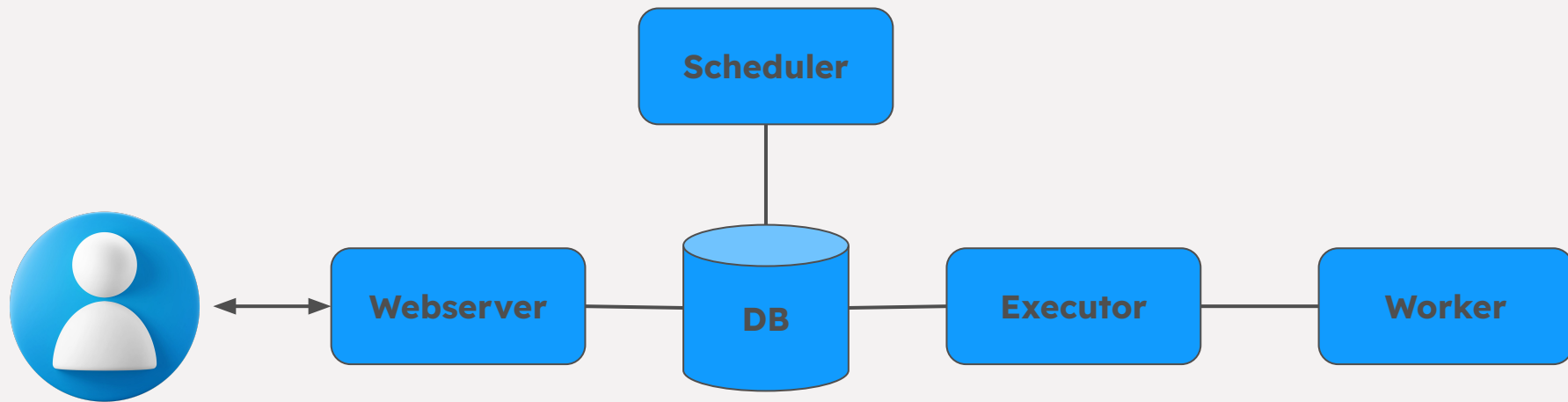
**DAG**

**Task**

**DagRun**

**TaskInstance**

# Arquitectura



# Arquitetura



**Trigger Dag - Sample DAG with Display Name** ×

**Single Run** ☒ **Backfill** ☐  
Trigger a single run of this Dag Run this Dag for a range of dates

**Advanced Options**

**Trigger**

**Last Dag Run**

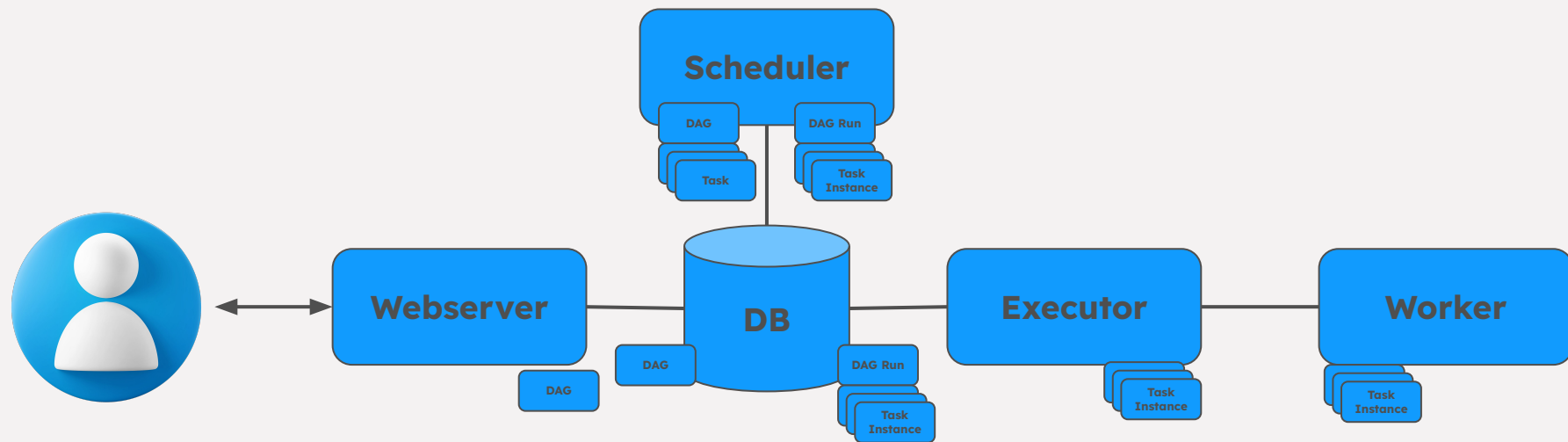
Duration

Queued Duration Run Duration

Task	Queued Duration	Run Duration
1	1.00	3.22
2		
3		
4		

**Worker**

# Arquitetura





# Arquitetura



**DAG**

**Task**

**DagRun**

**TaskInstance**

# Dúvidas?

# Mão na massa...

```
$ docker compose up --build
```

# Mão na massa...

The screenshot displays the Apache Airflow web interface. The left sidebar contains navigation links: Home, Days, Assets, Browse, Admin, Docs, and User. The main content area is titled "Airflow" and includes a "Stats" section with buttons for "Failed Dags" (0), "Running Dags" (0), and "Active Dags" (0). Below this is a section for "First 10 favorite Dags" with a note that no favorites are currently set. The "Health" section shows the status of various components: MetaDatabase (checked), Scheduler (checked), Triggerer (failed), and Dag Processor (checked). A "Pool Slots" bar indicates 128 slots are available. The "History" section shows a time range from 2025-10-18 10:16:04 to 2025-10-19 10:16:04. The "Dag Runs" section shows a table with columns for status and percentage, with all statuses (Queued, Running, Success, Failed) at 0%. The "Task Instances" section is currently empty. The "Asset Events" section shows "No Asset Events found."

**Airflow**

Stats

Failed Dags > Running Dags > Active Dags >

First 10 favorite Dags  
No favorites yet. Click the star icon next to a Dag in the list to add it to your favorites.

Health

Pool Slots

Manage Pools

✓ MetaDatabase ✓ Scheduler ✗ Triggerer ✓ Dag Processor ✓ 128

History

Last 24 Hours 2025-10-18 10:16:04 - 2025-10-19 10:16:04

Dag Runs

Status	Count	Percentage
Queued	0	0%
Running	0	0%
Success	0	0%
Failed	0	0%

Task Instances

Asset Events

Newest First

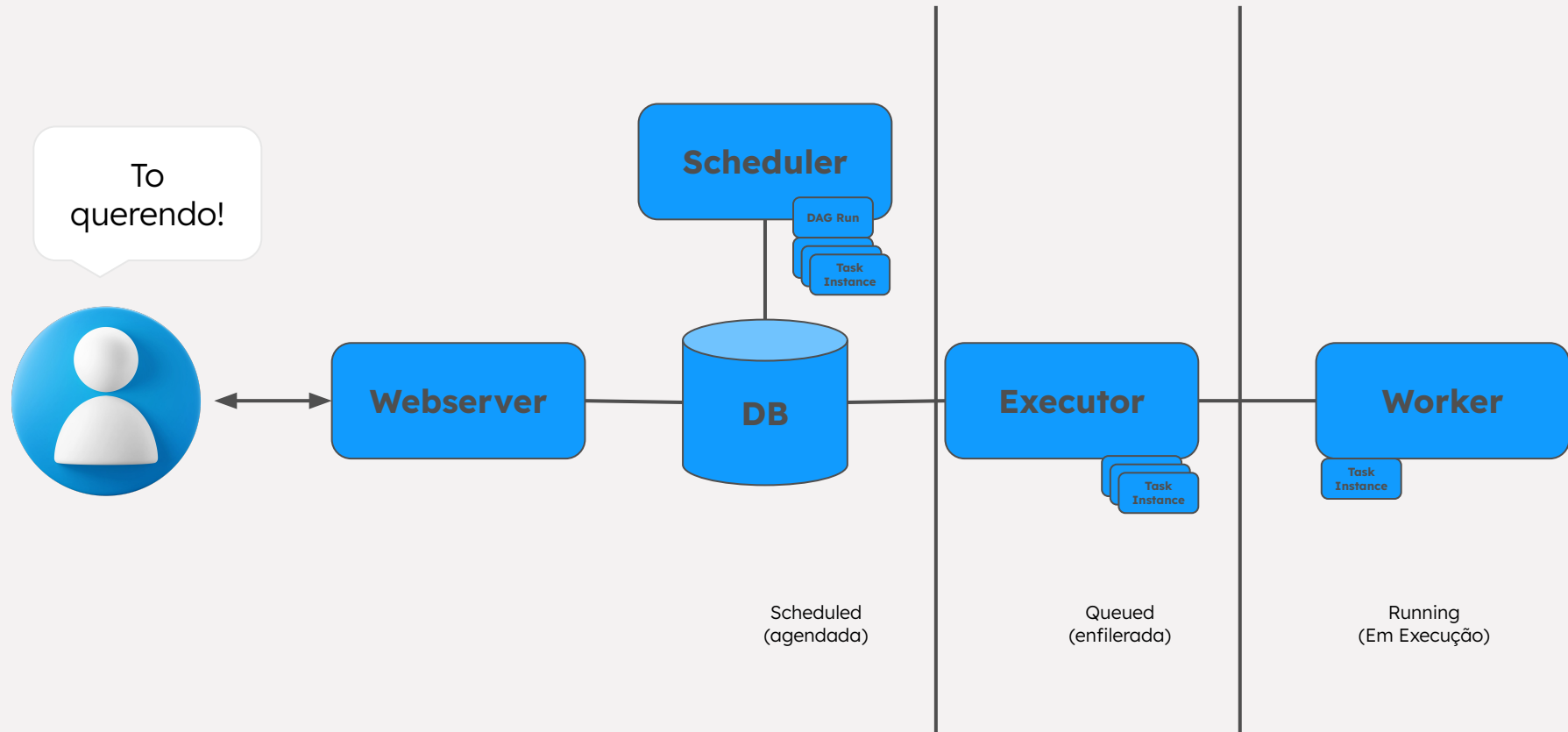
No Asset Events found.

# Mão na massa...

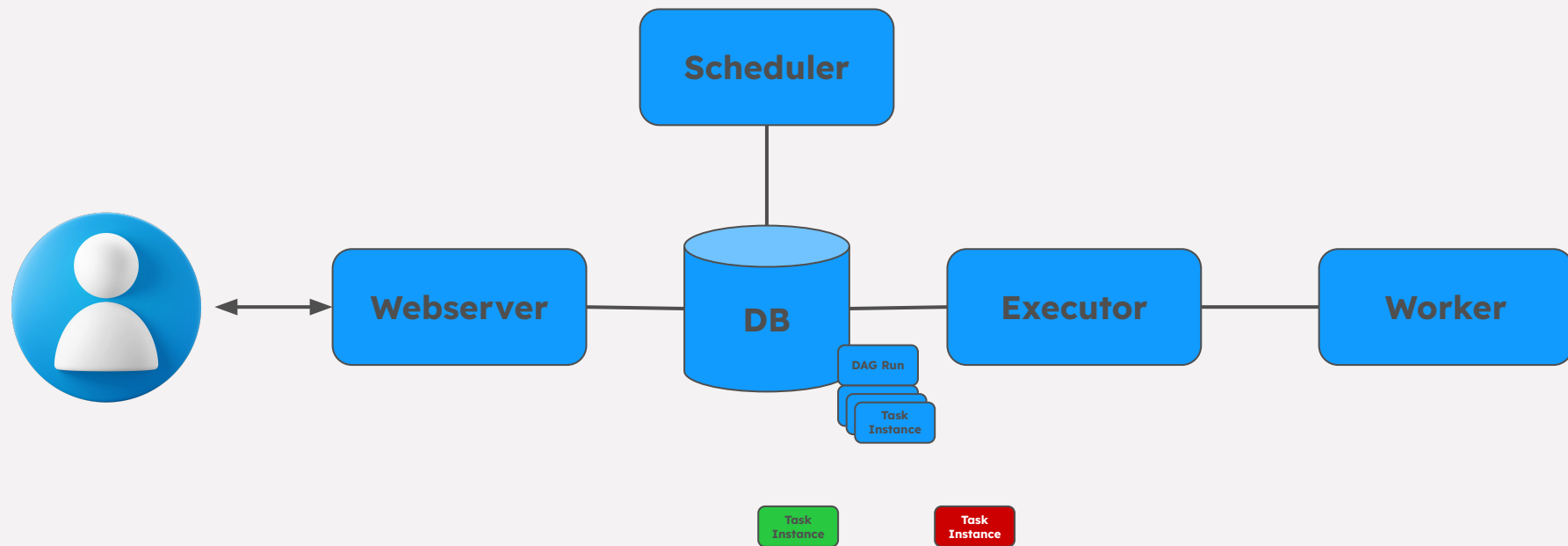
Executar a dag `hello_world_dag`

Verificar os logs

# Mão na massa...

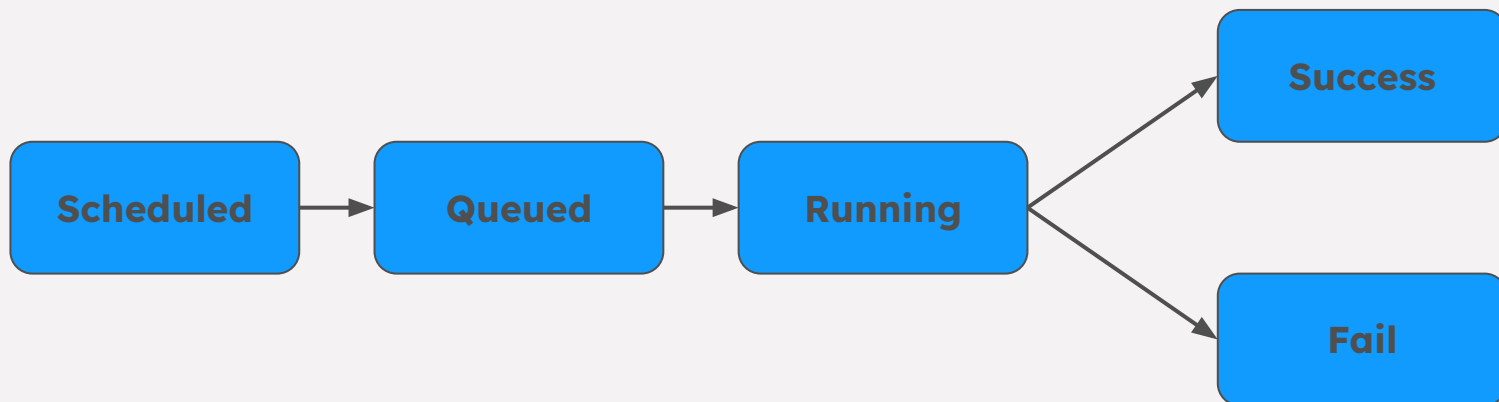


# Mão na massa...



# Task Lifecycle

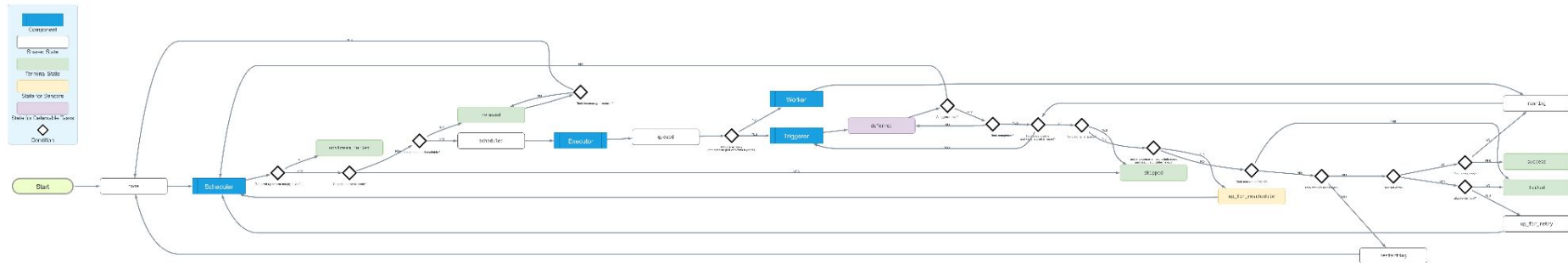
(ciclo de vida de uma tarefa)





# Task Lifecycle

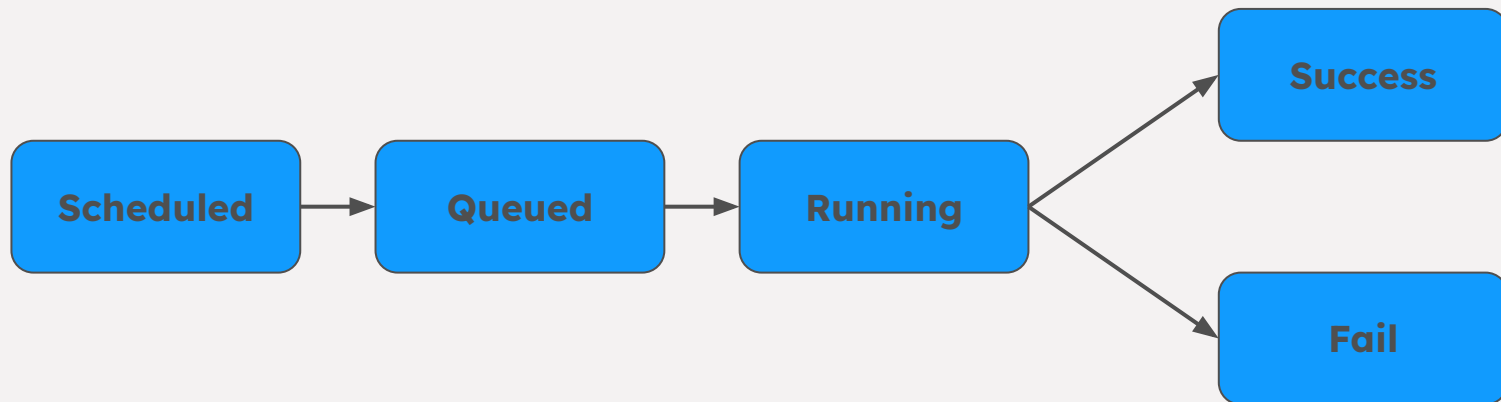
(ciclo de vida de uma tarefa)



<https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/tasks.html>

# Task Lifecycle

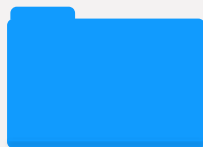
(ciclo de vida de uma tarefa)



# Dúvidas?



Porquê?  
Quando?  
Quando não?



Arquitetura



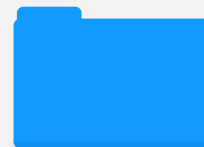
Criando um  
ambiente



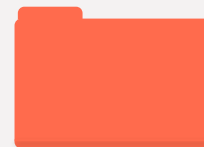
Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores

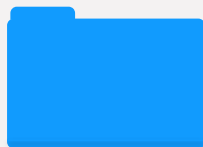


Padrões de  
Pipelines





Porquê?  
Quando?  
Quando não?



Arquitetura



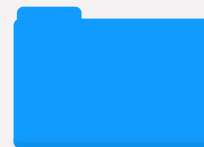
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores



Padrões de  
Pipelines



# Mão na massa...

Fazer uma dag onde uma tarefa só começa depois do final da outra

Em caso de dúvidas olhar os arquivos:

- `project_home/dags/dependencia.py`
- `project_home/dags/dependencia2.py`

# Mão na massa...

Processar os dados de vendas

“Esqueleto” do exercício no arquivo:

- `project_home/dags/processar_dados_de_vendas.py`

# Mão na massa... (dicas de pandas)

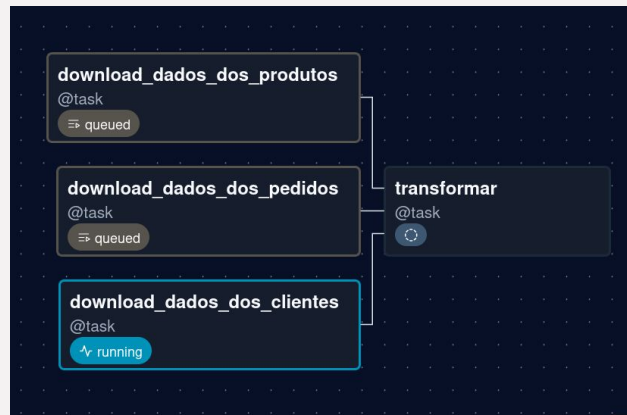
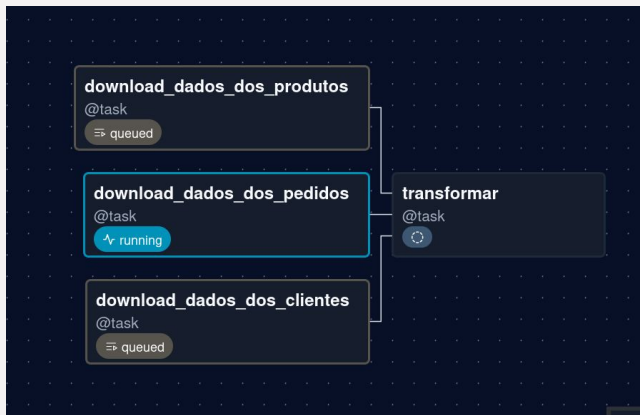
```
# fazer o download e salvar um arquivo
>>> resp = httpx.get(url)
>>> with open('nome_do_arquivo.csv.zip', 'wb') as f:
    f.write(resp.content)
```

```
# abrir um arquivo com o pandas
>>> df = pd.read_csv('nome_do_arquivo.csv.zip')
```

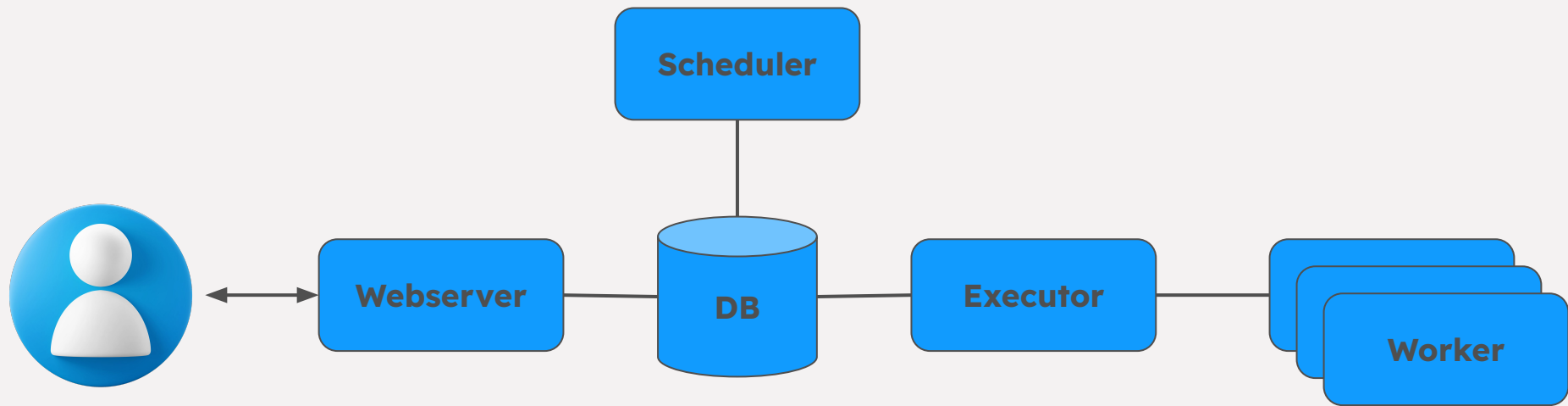
```
# fazer "join" de duas tabelas com o pandas
>>> resultado = pd.merge(
...     left=df_pedidos,
...     left_on='cliente_id',
...     right=df_clientes,
...     right_on='id',
... )
```



# Paralelização



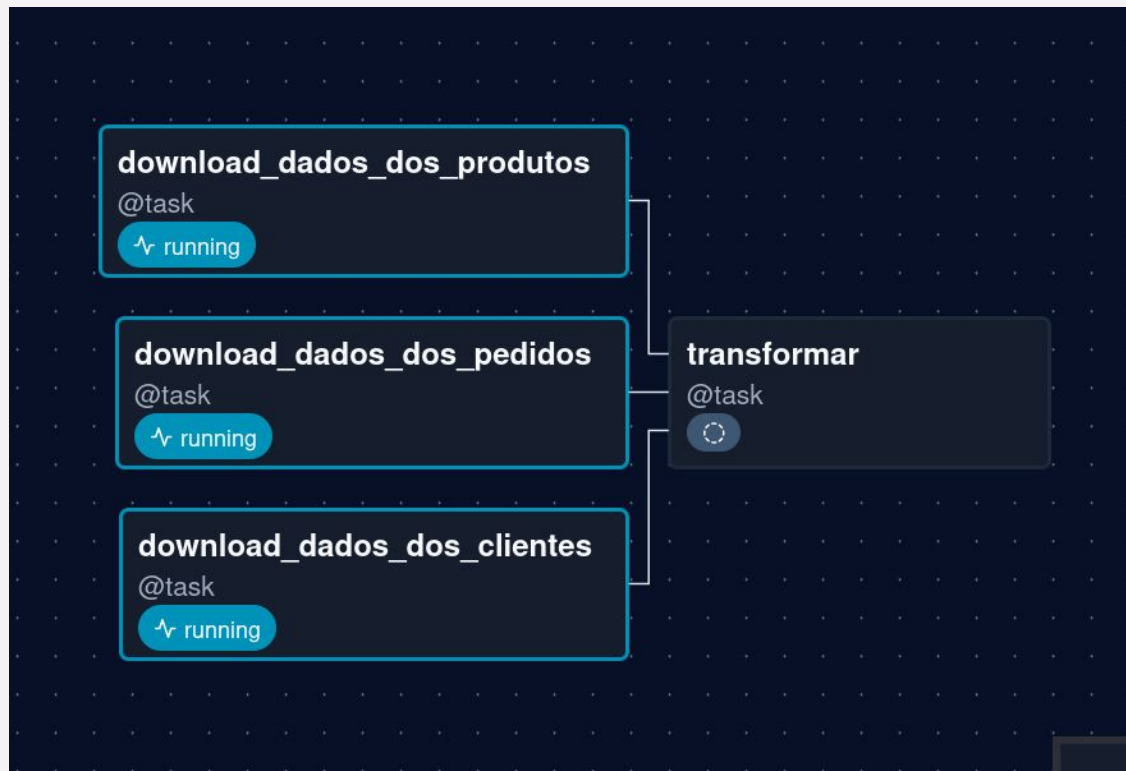
# Paralelização



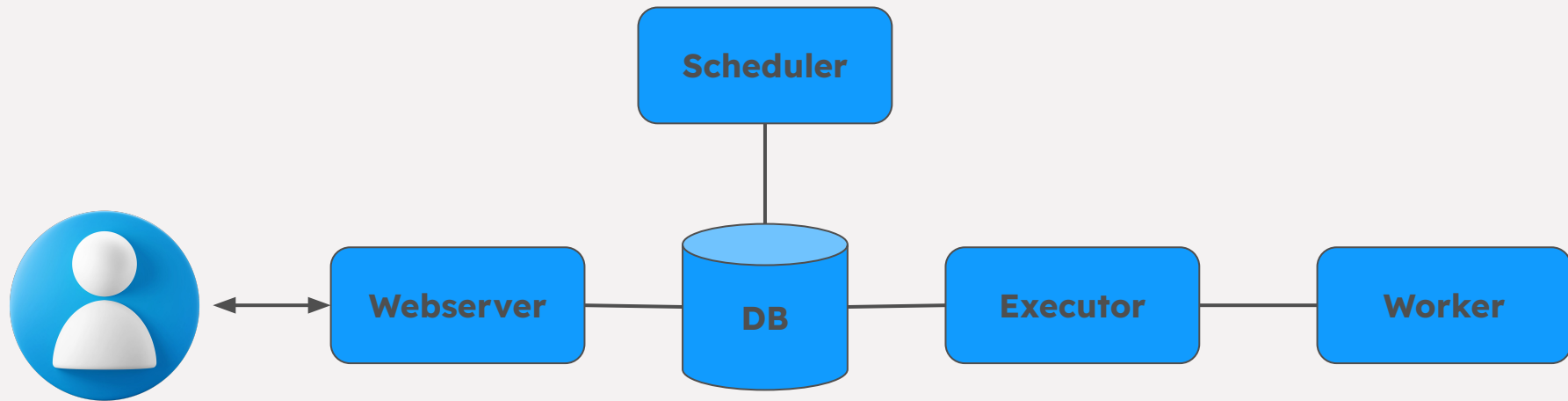
# Paralelização

```
$ docker compose scale worker=3
```

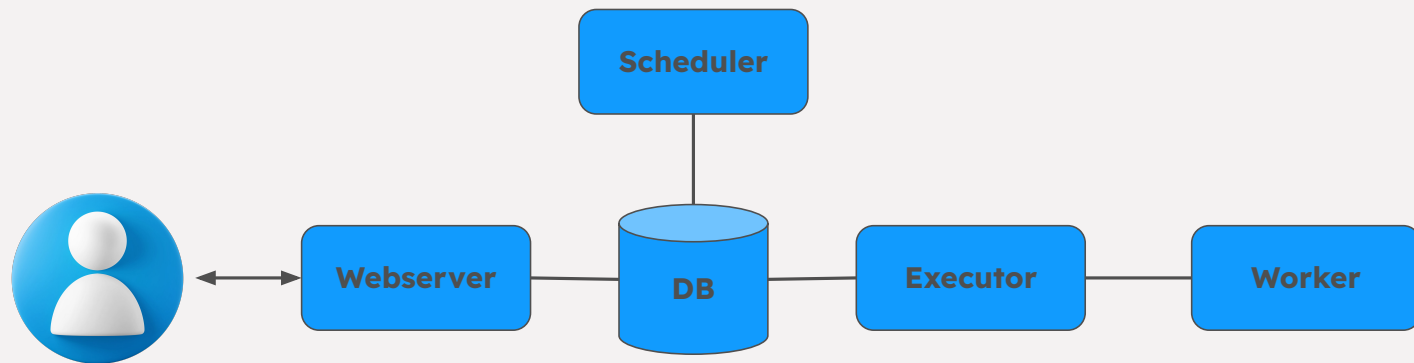
# Paralelização



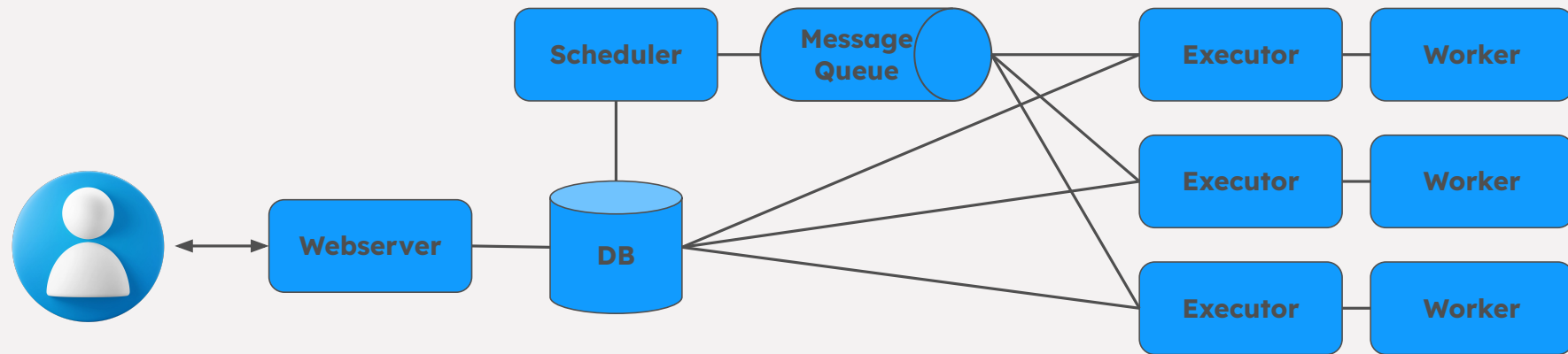
# Paralelização



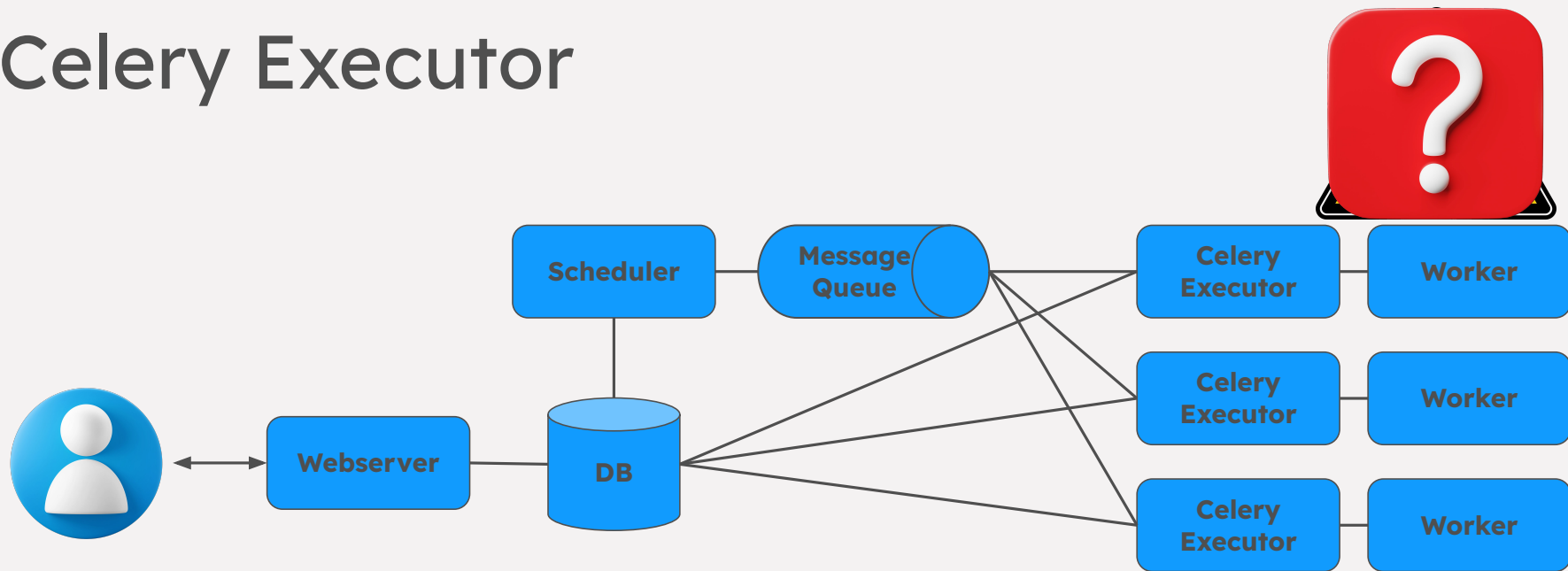
# Paralelização



# Paralelização

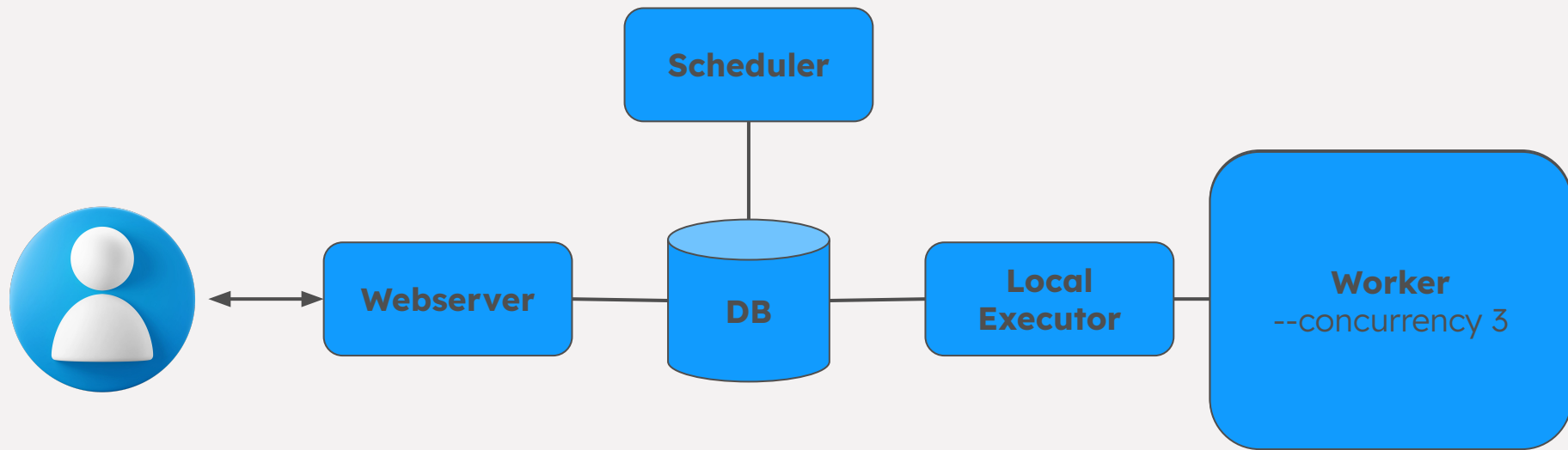


# Celery Executor

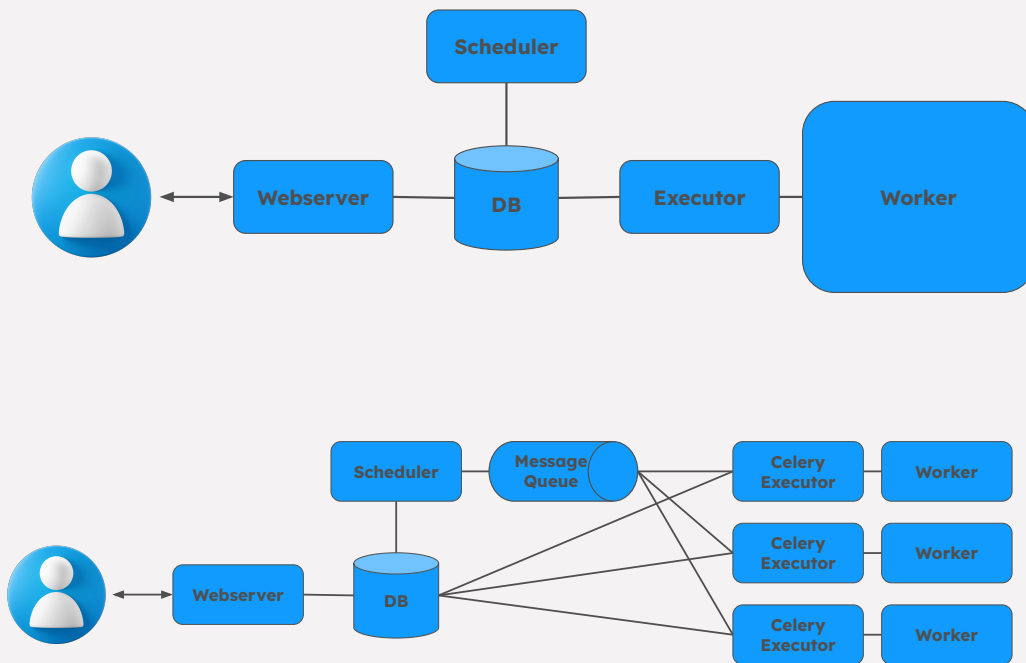




# Simplicidade...



# Escala Horizontal vs Vertical



# Mão na massa...

```
@task.branch
def branch_task():
    return 'odd_task' if random.randint(1, 2) == 1 else 'even_task'

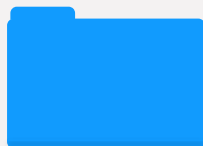
@task
def odd_task():
    return

@task
def even_task():
    return
```

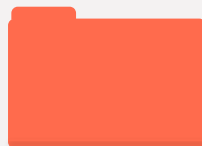
# Dúvidas?



Porquê?  
Quando?  
Quando não?



Arquitetura



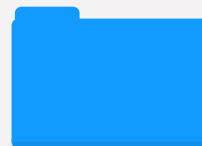
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores

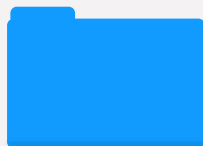


Padrões de  
Pipelines





Porquê?  
Quando?  
Quando não?



Arquitetura



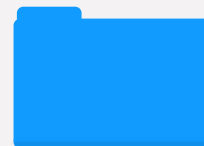
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores



Padrões de  
Pipelines



# Confiabilidade

O que é um sistema confiável?

Um sistema cujo resultado é confiável

1. Resultado sempre disponível
2. Resultado sempre certo

# Disponibilidade

Deploy automático

Healthcheck

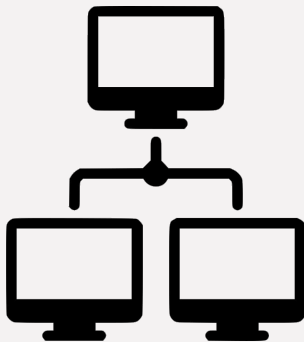
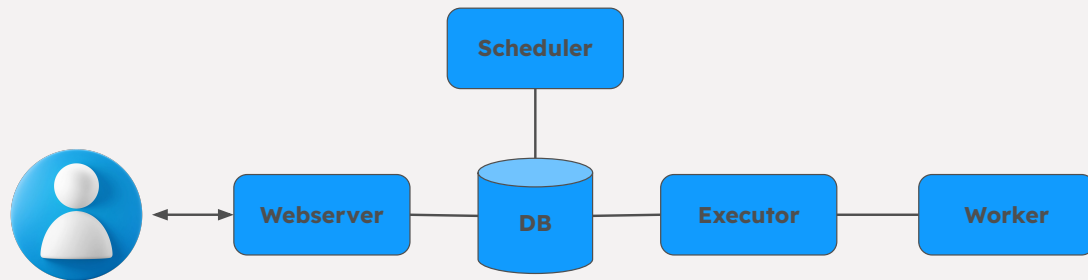
Escala em tempo de execução

Atenção aos pontos únicos de falhas

Monitoramento



# Disponibilidade



# Resultado correto

Testes do código

unitários, integração, e2e...

Testes dos dados

formato, tipos, valores, agregados...

# Resultado correto

Testes automatizados com Python, Pytest e Playwright

- Andressa

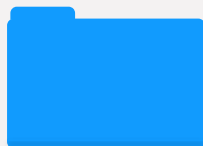
Testes automatizados e qualidade de dados em data pipelines

- Talissa
- Gabrielle

# Dúvidas?



Porquê?  
Quando?  
Quando não?



Arquitetura



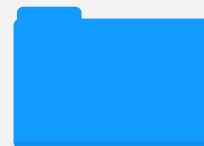
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores

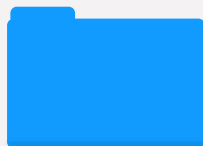


Padrões de  
Pipelines





Porquê?  
Quando?  
Quando não?



Arquitetura



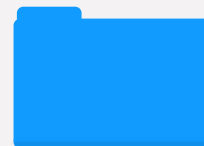
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores



Padrões de  
Pipelines



# Operators

Um operador é um modelo conceitual de uma Task

Nós já vimos dois (ou um?!): o [PythonOperator](#) e o [BranchPythonOperator](#)

Existem, por exemplo, o [BashOperator](#) que executa comandos bash

Outro útil é o [SQLExecuteQueryOperator](#)

E o [HttpOperator](#) e o [EmailOperator](#)

# Operators

```
resultado = SQLExecuteQueryOperator(  
    task_id="execute_query",  
    sql=f"SELECT 1; SELECT * FROM {TABLE} LIMIT 1;",  
    split_statements=True,  
    return_last=False,  
)
```



# Operators

Alguns exemplos de operadores úteis diversos:

TriggerDagRunOperator  
BranchDateTimeOperator  
SQLColumnCheckOperator  
SQLInsertRowsOperator  
EmptyOperator

PythonSensor  
TimeDeltaSensor  
ExternalTaskSensor  
FileSensor

DatabricksCreateJobsOperator  
DatabricksRunNowOperator  
S3FileTransformOperator  
EC2StartInstanceOperator  
ConsumeFromTopicOperator  
ProduceToTopicOperator  
SlackAPIOperator  
SlackAPIFileOperator  
SparkJDBCOperator  
SparkSubmitOperator  
SalesforceApexRestOperator  
TelegramOperator  
TelegramFileOperator

# Providers

Um Provider é um módulo de utilidades (operators, sensors, hooks)

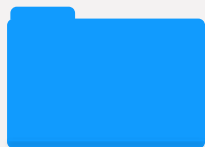
É o equivalente a um módulo do python

• Airbyte	• dbt Cloud	• Open Search
• Alibaba	• Dingding	• Opsgenie
• Amazon	• Discord	• Oracle
• Apache Beam	• Docker	• Pagerduty
• Apache Cassandra	• Edge3	• Papermill
• Apache Drill	• Elasticsearch	• PgVector
• Apache Druid	• Exasol	• Pinecone
• Apache Flink	• FAB (Flask-AppBuilder)	• PostgreSQL
• Apache HDFS	• Facebook	• Presto
• Apache Hive	• File Transfer Protocol (FTP)	• Qdrant
• Apache Iceberg	• Git	• Redis
• Apache Impala	• GitHub	• Salesforce
• Apache Kafka	• Google	• Samba
• Apache Kylin	• gRPC	• Segment
• Apache Livy	• Hashicorp	• Sendgrid
• Apache Pig	• Hypertext Transfer Protocol (HTTP)	• SFTP
• Apache Pinot	• IBM Cloudant	• Singularity
• Apache Spark	• Influx DB	• Slack
• Apache Tinkerpop	• Internet Message Access Protocol (IMAP)	• SMTP
• Apprise	• Java Database Connectivity (JDBC)	• Snowflake
• ArangoDB	• Jenkins	• SQLite
• Asana	• Keycloak	• SSH
• Atlassian Jira	• Microsoft Azure	• Standard
• Celery	• Microsoft SQL Server (MSSQL)	• Tableau
• Cloudant	• Microsoft PowerShell Remoting Protocol (PSRP)	• Telegram
• CNCF Kubernetes	• Microsoft Windows Remote Management (WinRM)	• Teradata
• Cohere	• MongoDB	• Trino
• Common Compat	• MySQL	• Vertica
• Common IO	• Neo4j	• Weaviate
• Common Messaging	• ODBC	• Yandex
• Common SQL	• OpenAI	• YDB
• Databricks	• OpenFaaS	• Zendesk
• Datadog	• OpenLineage	

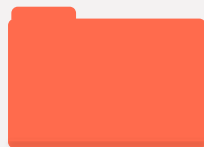
# Dúvidas?



Porquê?  
Quando?  
Quando não?



Arquitetura



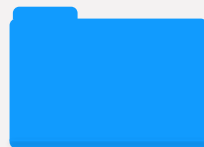
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores

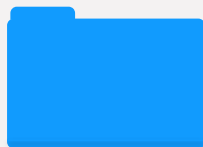


Padrões de  
Pipelines





Porquê?  
Quando?  
Quando não?



Arquitetura



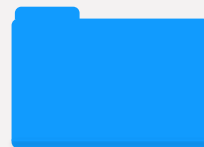
Criando um  
ambiente



Criando DAGs



Aumentando a  
Confiabilidade



Zoologico de  
Operadores



Padrões de  
Pipelines



# Múltiplos Ambientes

Mantenha um ambiente local muito parecido com o de produção

Use containers (docker, podman). Mantenha os serviços separados!

Se em produção for utilizar mais de um banco, faça o mesmo localmente

O mesmo Dockerfile que for utilizado em produção deve ser o local

Airflow permite (e as vezes exige) que sejam utilizadas muitas variáveis de ambiente. Use com sabedoria:

- tente eliminar variáveis que têm o mesmo valor em todos os ambientes
- foque nas variáveis cujo valor realmente é diferente de ambiente para ambiente

# Logs

O Airflow gera dois “níveis” de logs: Task Logs e System Logs

Considere que são logs de aplicações separadas:

- controle de acesso distintos
- alertas distintos
- políticas de retenção distintas
- (opcional) plataformas de armazenamento/visualização distintas

# Instalação

Restrinjas as versões de dependências externas com o constraints

Prefira construir seu próprio Dockerfile à utilizar o oficial

Evite instalar o apache-airflow[all]



# Banco de dados

Oficialmente o Airflow o metadb deve ser um Postgres ou MySQL

O Airflow é conhecido por utilizar muitas conexões com o banco. Considere utilizar um serviço de Pool de conexões (ex.: PgBouncer). Especialmente se estiver usando Postgres

Lembre-se de colocar um limite de timeout nas configurações de conexão com o banco.

Vale a pena fazer uma “limpeza” no metadb de tempos em tempos.

# Recursos externos

Utilize Pools para reduzir a chance de DoS em recursos externos

É possível utilizar dags para gerenciar o tamanho das Pools

# Deploy

É possível fazer deploy somente das dags: basta atualizar os arquivos das dags nos vários serviços

Cuidado ao atualizar tasks durante a execução de dags.

Caso faça mudanças frequentes, considere versionar as dags (ou, pelo menos, logar o commit ou hash do código da dag)

Considere o downtime quando precisar fazer deploy/reiniciar o Airflow.

# Testes Automatizados

Evite Class Tasks com muitos parametros

Se possível, faça testes da estrutura das dags

É possível utilizar o próprio Airflow para executar testes nele mesmo

Se possível, crie mocks dos serviços de terceiros para testes

Crie os testes de dados com a mesma importancia dos testes de código

Escreva testes de “integração”. É comum ter lógica importante em SQL

# Criando DAGs

Busque idempotencia

Evite código “caro” no nível da DAG ou do módulo

Use os retries automáticos das Tasks (com sabedoria!)

# Anti-padrões

Utilização de CRON jobs ou outros serviços orquestrados

Milhares de DAGs (??)

Utilização de tabelas para acompanhar “status” de serviços

Ambiente local diferente do ambiente de produção

Não ter testes automatizados do código

Não ter testes automatizados dos dados