# Training in the Cloud

## Plan for the lecture

1️⃣Reminders
2️⃣Objective
3️⃣Cloud platform
4️⃣Application parameters
5️⃣Model in the cloud
6️⃣Data in the cloud
7️⃣Training in the cloud

# 1️⃣Reminders

### Minimal package structure

```
.                                # 🌱 project root
└── scikit-learn                 # 📦 package root
    ├── .python-version          # 🐍 virtual environment
    ├── Makefile                 # 📝 command line reminders
    ├── requirements.txt         # 🐼 dependencies
    ├── setup.py                 # ⚙️ package declaration
    └── sklearn                  # 🥚 importable package
        ├── __init__.py          # 🚧 forget me not
        └── preprocess.py        # 🐣 importable class
```

### Package installation

```
pip install .             # 📦 release

pip install -e .          # 🚧 work in progress
```

## 2️⃣ Objective

**Where are we in our journey?**

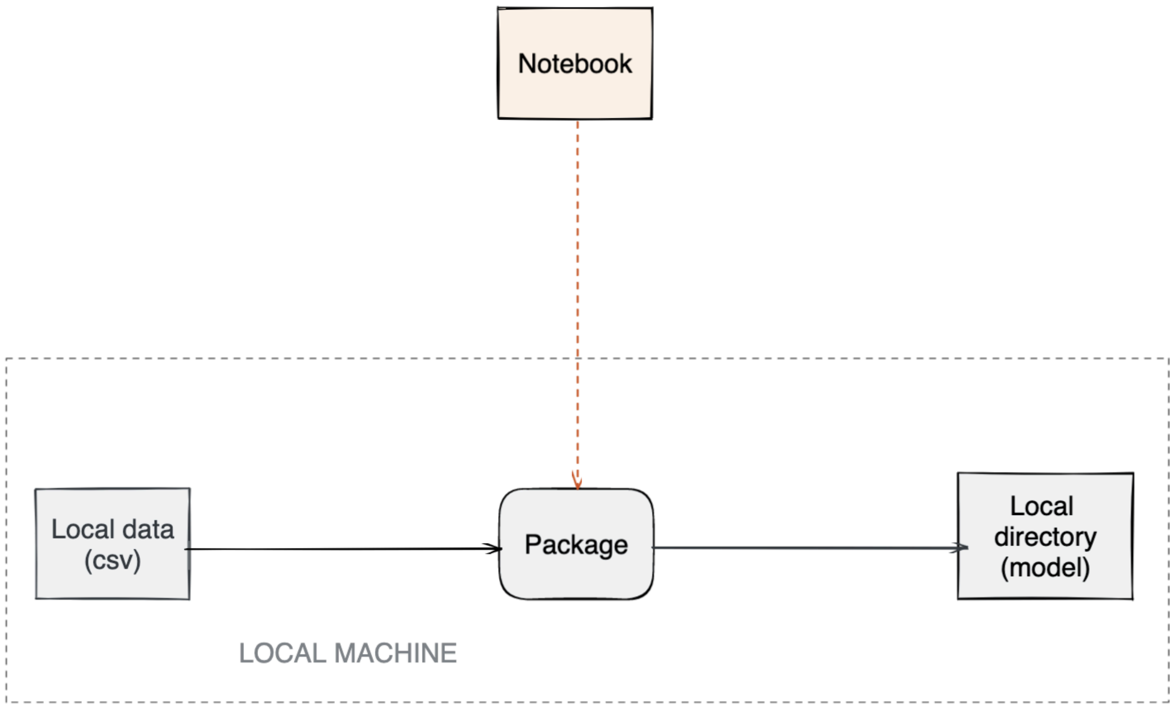
🔭 Build the **WagonCab** app 🚗 from the **notebook** provided by the Data Science team 🧑‍🔬

✅ Analyze 🔬 the **notebook**

✅ Convert the notebook into a Python **package** 📦 to make its code *operable* ♻️

✅ *Scale* 🗻 the code to train the package on the **full dataset**

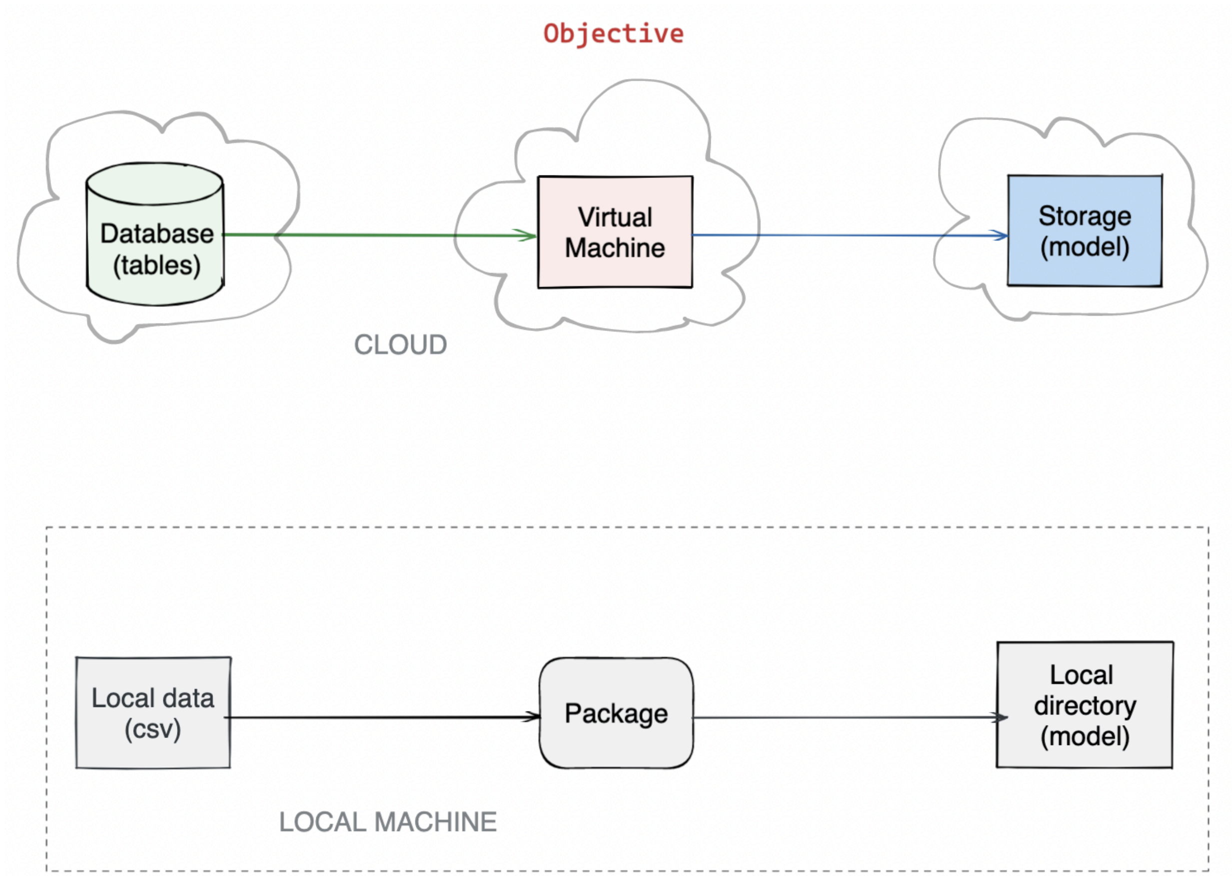
## `Train at scale`




**What's the next step?**

🎯 We want our **package** 📦 to work in the cloud:

- Allow team members to **collaborate**
- Stop **monopolizing** our machine during training
- *Plug* ♻️ constantly evolving, **real world data** and **production processes**
- Train on a *virtual machine* 🧮 with a **GPU**

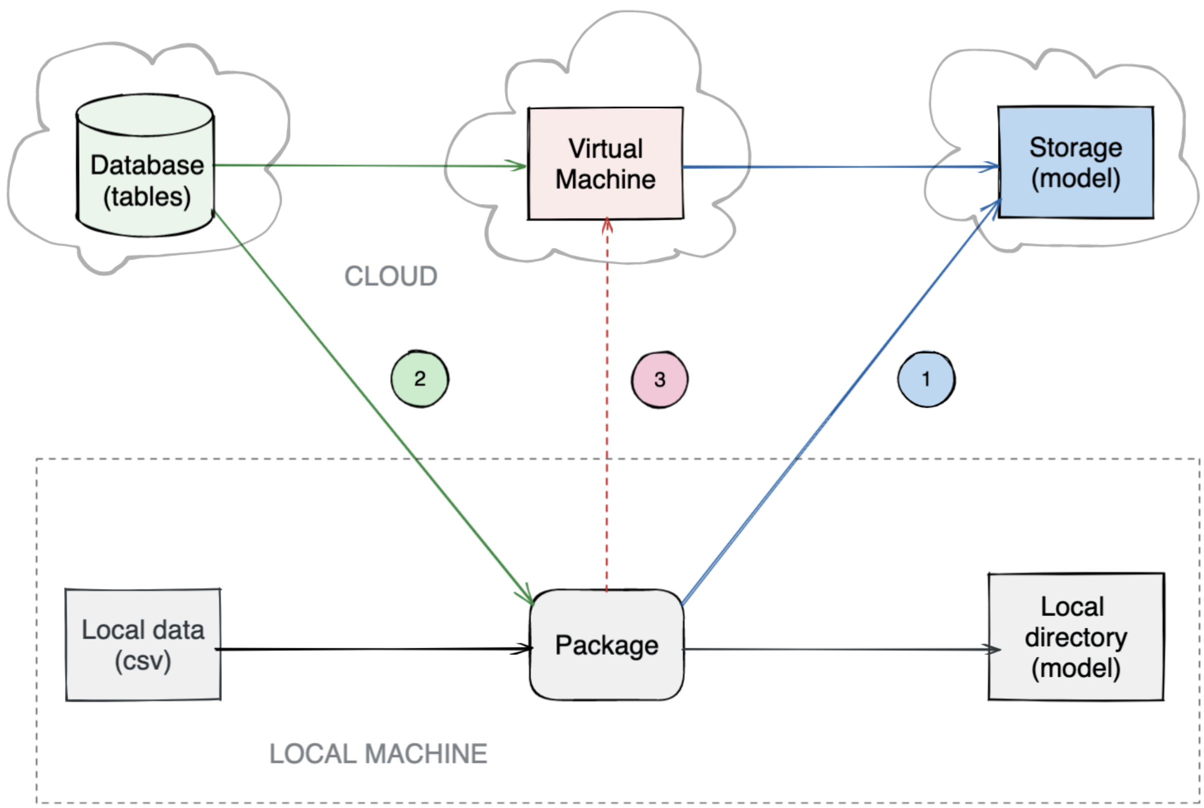


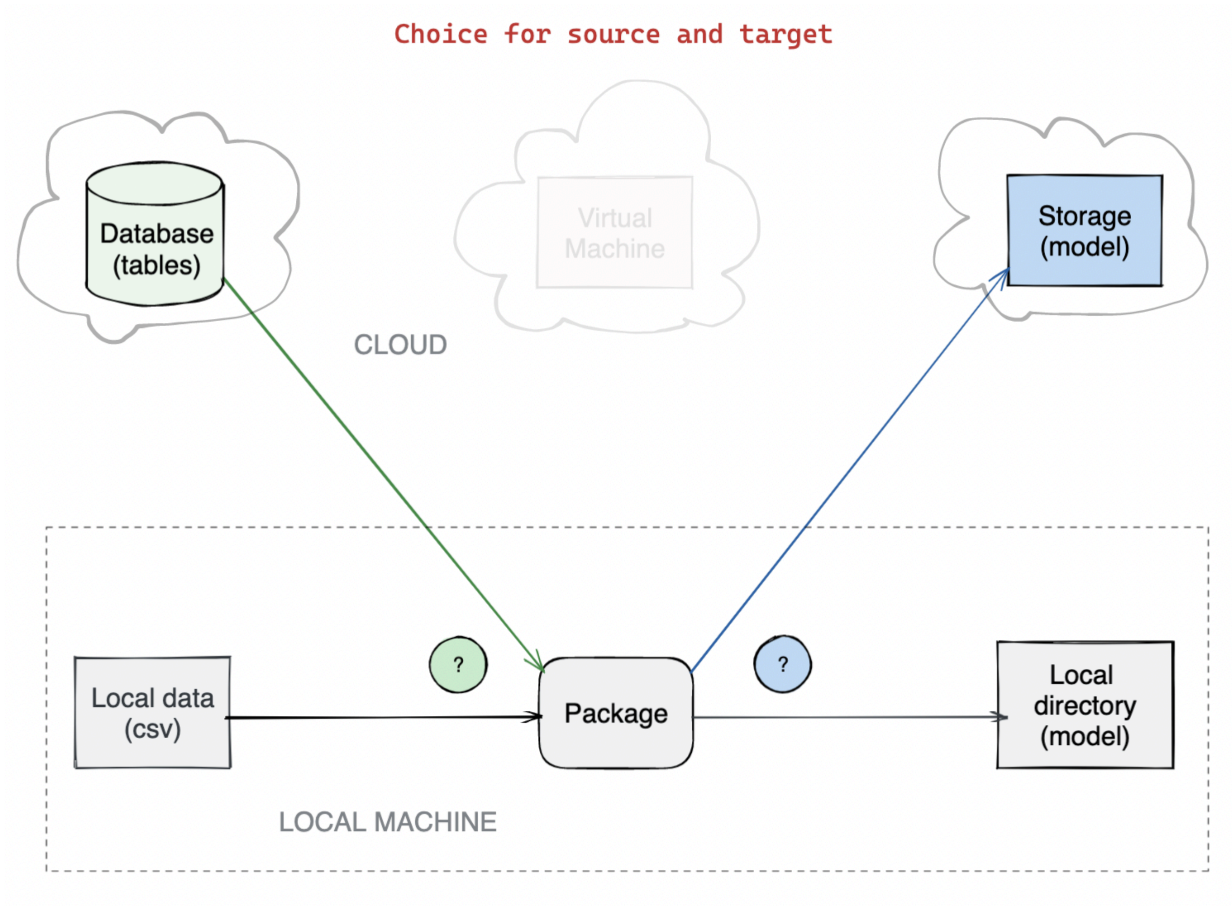

**Transition**

❓ How to **decouple** the package from our machine 💻

1. Change the output: save the **trained model** to the cloud
2. Change the input: train from data **in the cloud**
3. Change the execution: run the training on a **virtual machine** in the cloud

Transitions






# Updated Package structure

```
.
├── .env                    # ⚙️ Single source of config variables
├── .envrc                  # 🎬 .env loader (used by direnv)
├── Makefile                # New commands "run_train", "run_process", etc..
├── README.md
├── requirements.txt
├── setup.py
├── taxifare
│   ├── __init__.py
│   ├── interface
│   │   └── main_local.py       # 🚪 (OLD) entry point
│   │   └── main.py             # 🚪 (NEW) entry point: No more chunk!
│   ├── ml_logic
│   │   ├── data.py             # (UPDATED) Loading and storing data from/to Big Query !
│   │   ├── registry.py         # (UPDATED) Loading and storing model weights from/to Cloud Storage!
│   │   ├── ...
│   ├── params.py               # Simply load all .env variables into python objects
│   └── utils.py
```

```
└── tests
```

# 3️⃣ Cloud Platform

## Google Cloud Platform

👉 On-demand **computing** resources

👉 Resources are **elastic** (scale up and down)

## Service layers

#### On-premise

- Computer with a *virtualization* layer
- Manage the *location*, the *hardware*, the *operating system*, and the code *environment*

**IaaS**: **infrastructure** as a service (*Google Compute Engine*)

- *Virtual machine* in the cloud
- Choose the *location* and the *hardware*, pay for what you **allocate**
- Manage the *operating system* 🌍 and the *environment* 🌴

**PaaS**: **platform** as a service (*Cloud Run*)

- Choose the *environment* for your code, pay for what you **use**
- Manage the *package* 📦

**SaaS**: **software** as a service (*Google Big Query*)

- Choose the software 🏏

# Platform

### Google Cloud Platform

- A product for everything
- User-friendly
- Fast learning curve
- 20% cheaper than Microsoft Azure

| Cloud provider | Storage | Database | Compute | Products |
|---|---|---|---|---|
| Amazon | S3 (Simple Storage Service) | Athena/Redshift/Redshift Spectrum | EC2 (Elastic Compute Cloud) | AWS Cloud products |
| Microsoft | Azure Blob Storage | Azure Synapse Analytics | Azure Virtual Machines | Azure products |
| Google | Cloud Storage | Big Query | Compute Engine | Google Cloud products |

AWS vs Azure vs GCP product comparison

# Interface Rule of Thumb

🌍 **web console**

- Great for **exploration**, but quite slow
- For non-repetitive and precise operations

💻 **CLI** (`gcloud`, `gsutil`, `bq`)

- Steep learning curve but **faster**
- Great for precise operations

🧬 **code**

- For the behaviors that need to be **automated**

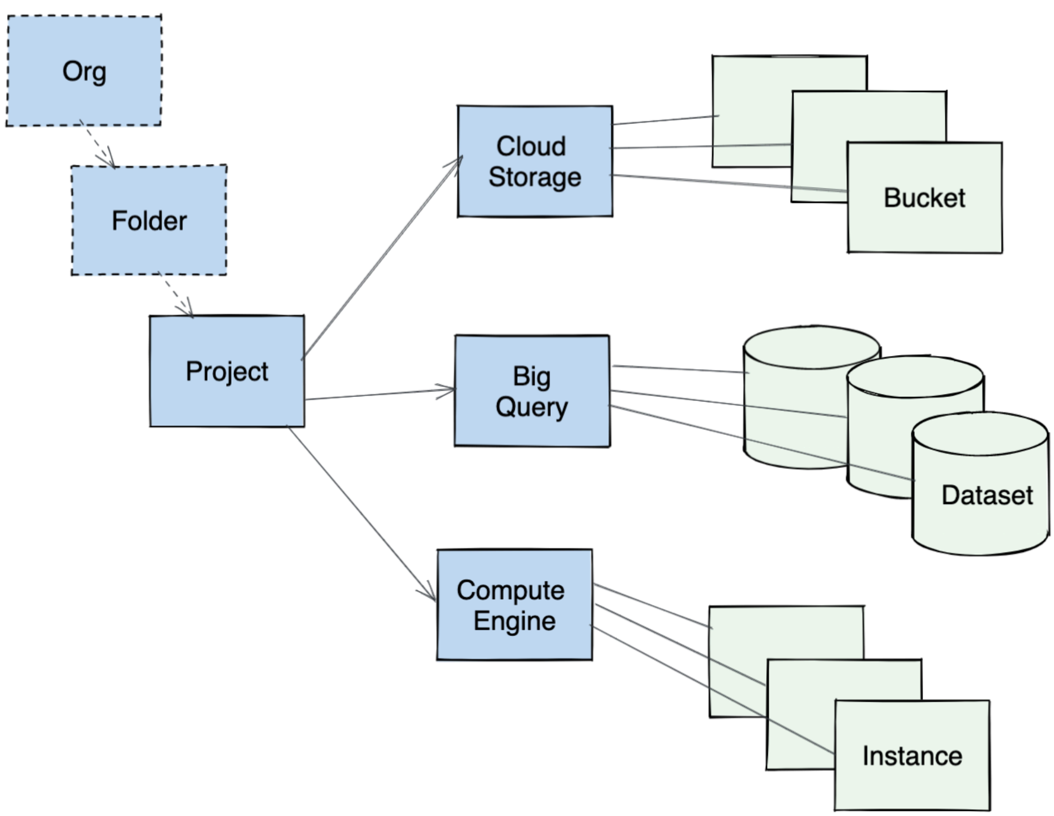
# Livecode 🚧

🎯 Checkout the webconsole and intoduce gcp and the services for today

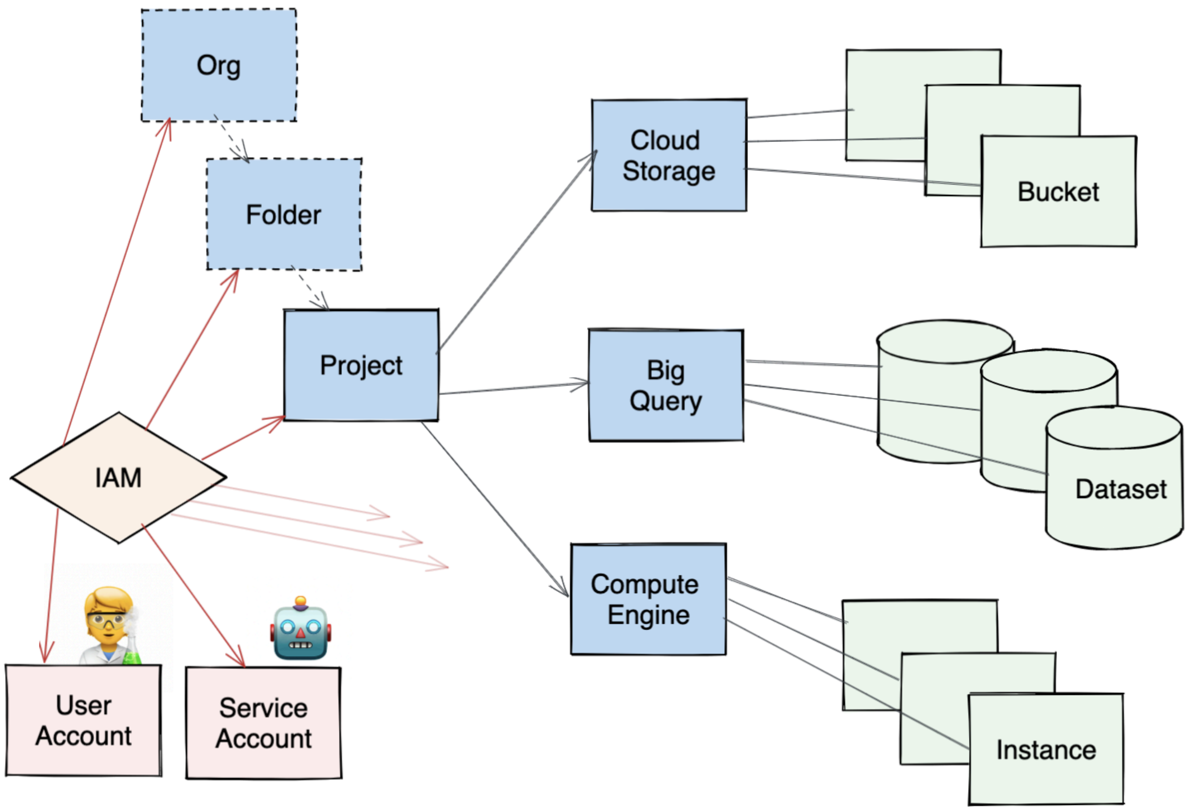
💻 Show `GOOGLE_APPLICATION_CREDENTIALS` that we setup on the very first day!

💻 Show how to authenticate **code and the cli**

`Resource structure`



Identity & access management




# Projects

**Projects**: organize **GCP resources**

**Organizational nodes & folders**: organize **projects**

**Regions** and **zones**: **deployment** areas for resources

**CLI**

```
gcloud projects list                # list projects

gcloud compute regions list         # list compute regions
gcloud compute zones list           # list compute zones
```

# Accounts

**Accounts**: **user** identifier

**Service account**: **application** identifier

**IAM**: identity and access management for account organization, groups, **roles** & audits

**CLI**

gcloud init *# setup CLI authentication*

gcloud config configurations list *# list CLI configurations*

gcloud auth application-default login *# setup code authentication*

gcloud services list --enabled *# list enabled service APIs*

More **commands** in the [Google Cloud Platform]() and [Service Account]() cheatsheets

## Budget Alert 🚧

🎯 Setup a [budget alert]() to follow your resource spendings on **Google Cloud Platform**
**Shortcuts**

Shortcuts

- ❌ `Ctrl + C`: cancel / stop a running command
- ✏️ `Ctrl + U`: erase current line
- ⬅️ `Ctrl + A`: move the cursor to the beginning of the line
- ➡️ `Ctrl + E`: move the cursor to the end of the line
- ⏩ `Opt/Alt (or`Ctrl`on WSL) + ⬅️/➡️`: move the cursor a word to the left / right

# 4️⃣ Application Parameters

## Livecode 🚧

🎯 Change the behavior of the **package** 📦 depending on the **execution context**:

- between **collaborators**
- **local** vs **cloud**

**Setup a .env file**

💻 Create a `.env` file

- Add a `MODEL_TARGET` variable with value `local`

💻 Load the `.env` variables

- Create a `.envrc` file with the content `dotenv`
- Load the variables into the **environment** with `direnv allow .`

**Environment variable demonstration**

💻 Use the value in `taxifare`

- Change `MODEL_TARGET` in `params.py` to load the environment variable
- At the end of `registry.py` add a `__main__` block
- Show how we can alter the flow of python using our `.env` and `params.py`
- Shift `MODEL_TARGET` between `local` and `gcs`

🕵️ Checkout how we utilize `MODEL_TARGET` in `registry.py`
**Well done!** 🎉

The **package** 📦 now adapts its behavior to its **execution context**, thanks to the `.env` file

# Theory 🔭

📖 What we saw:

- `.env` file
- `direnv` loader
- Code **refactoring** with **flow control**

👉 [Separate configuration from code]()

# `.env` Project Configuration File

Defines application properties that vary depending on the **execution context**

🧭 **Define behaviors**

- Change the model target or data source **type**

⚙️ **Store resources**

- Local file **paths**, database **URIs** (e.g. `postgresql://user:pwd@hostname:port/db`)

🔑 **Store credentials**

- Account **credentials**, API **tokens**, application **secrets**

🚨 🚨 🚨 Do **NOT** track this file with **Git** (check your `.gitignore`) 🚨 🚨 🚨

# `direnv` Configuration Loader

Exposes the `.env` variables in the **code**, in the **command line** or in a **Makefile** as environment variables

Install the `direnv` [command line script]

Create a `.envrc` loader file next to the `.env` file:

```
dotenv
```

Allow `direnv` to load the `.envrc` file:

```
direnv allow .              # path to the `.env` file
```

👉 `direnv reload` `direnv status`

## Environment Variables

In the **code** (we do this all in `params.py` to simplify code elsewhere!):

```python
import os

data_source = os.environ.get("DATA_SOURCE")  # if a default value is ok
data_source = os.environ["DATA_SOURCE"]      # to fail when the conf is missing
```

In the **command line**:

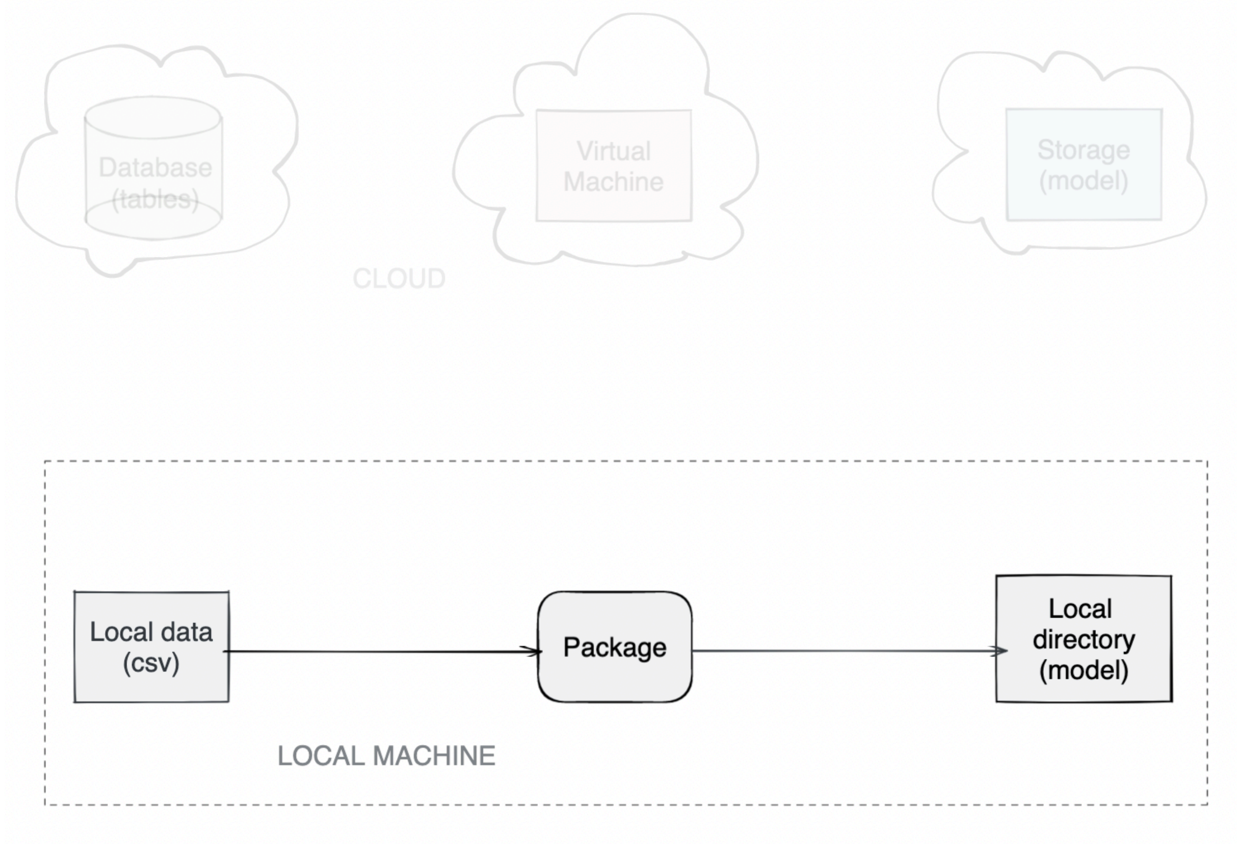
```bash
echo $DATA_SOURCE
```

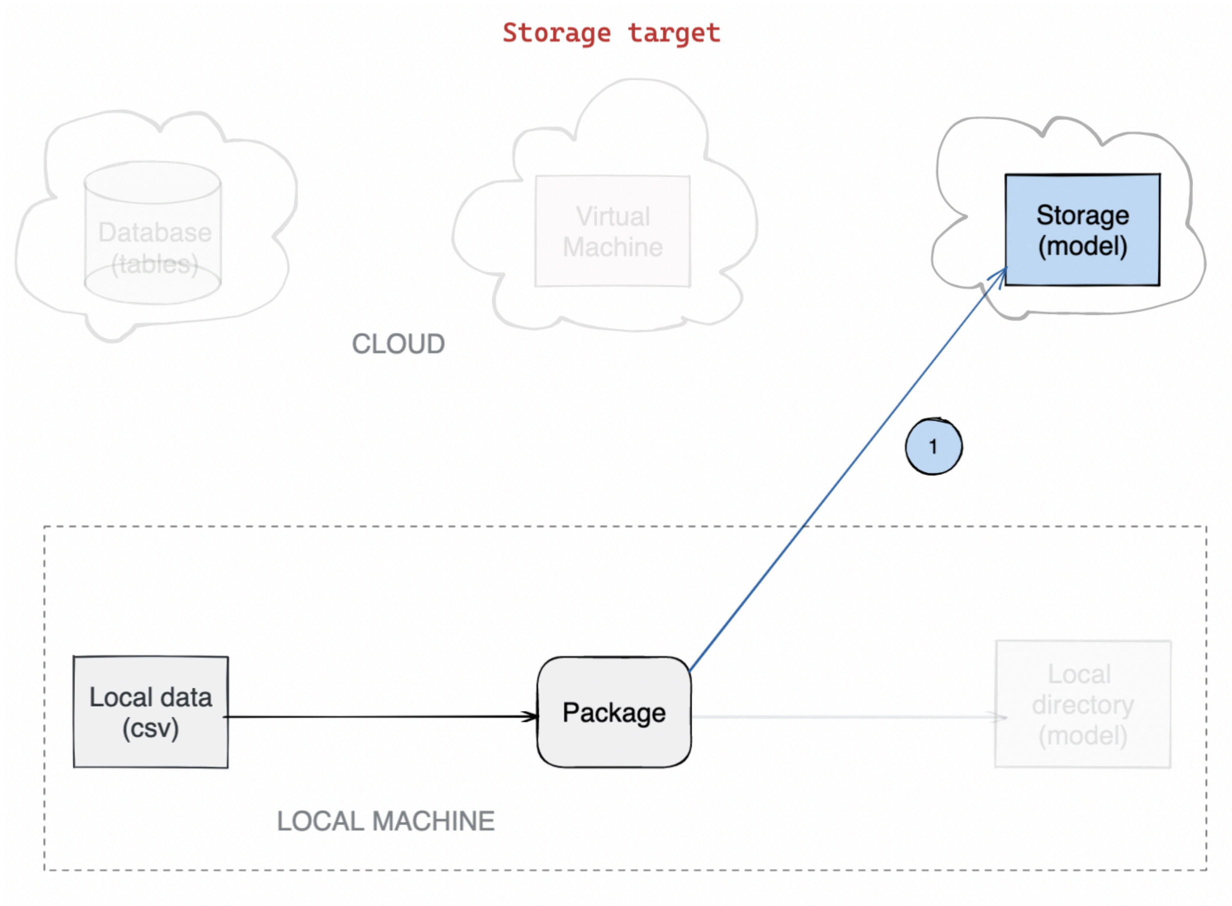
In a **Makefile**:

```makefile
print_data_source:
	echo $DATA_SOURCE              # ❌
	echo $DATA_SOURCE/data.csv     # ❌
	echo ${DATA_SOURCE}            # ✅
	echo ${DATA_SOURCE}/data.csv   # ✅
```

# 5️⃣ Model in the Cloud

Legacy training






## Google Cloud Storage

👉 Static storage for **unstructured** data

## Livecode 🚧

🎯 Push the trained model to the cloud on *every training*

**Save the Model to the Cloud**

💻 Create a bucket to save the model to!

- Create a bucket using the ui
- Upload one model manually!

💻 Configure the `.env`

- Change `MODEL_TARGET` to `gcs`
- Add `BUCKET_NAME` to params and .env

🕵️ Checkout the code

- In `taxifare.registry.py` checkout the `save_model` *function*
- Run a training (`make run_train`)!

**Checkout your bucket**

💻 Verify that the model was uploaded

- Using the *web console* 🌍
- Once we are confident with the *web console* 🌍, let's speed up things with the **CLI**

💻 Download the model

- Using the *web console* 🌍
- With the *CLI*

# Buckets

Containers for **blobs**, ❌ no tree structure

Worldwide **unique** kebab-case naming

**CLI**

```
REGION=europe-west1
PROJECT=project-id
BUCKET=bucket-name

gsutil ls                    # list buckets

gsutil mb \
  -l $REGION \
  -p $PROJECT \
  gs://$BUCKET               # create bucket
```

# Blobs

**Immutable** data storage

Best for *unstructured* data (text, images, videos, music)

Blob **URI**: `gs://bucket-name/blob/name` (URI vs URL)

**CLI**

```
gsutil ls gs://$BUCKET            # list blobs at the root of the bucket
gsutil ls -r gs://$BUCKET         # recursively list all blobs

gsutil cp *.csv gs://$BUCKET/     # copy all CSVs in the cwd to the bucket's root
gsutil cp gs://$BUCKET/*.csv .    # copy all CSVs at the bucket's root to the cwd
```

More **commands** in the Cloud Storage cheatsheet

**CODE**

**Download blob**

```
from google.cloud import storage

BUCKET_NAME = "my-bucket"

storage_filename = "models/xgboost_model.joblib"
local_filename = "model.joblib"

client = storage.Client()
bucket = client.bucket(BUCKET_NAME)
blob = bucket.blob(storage_filename)
blob.download_to_filename(local_filename)
```

**Upload blob**

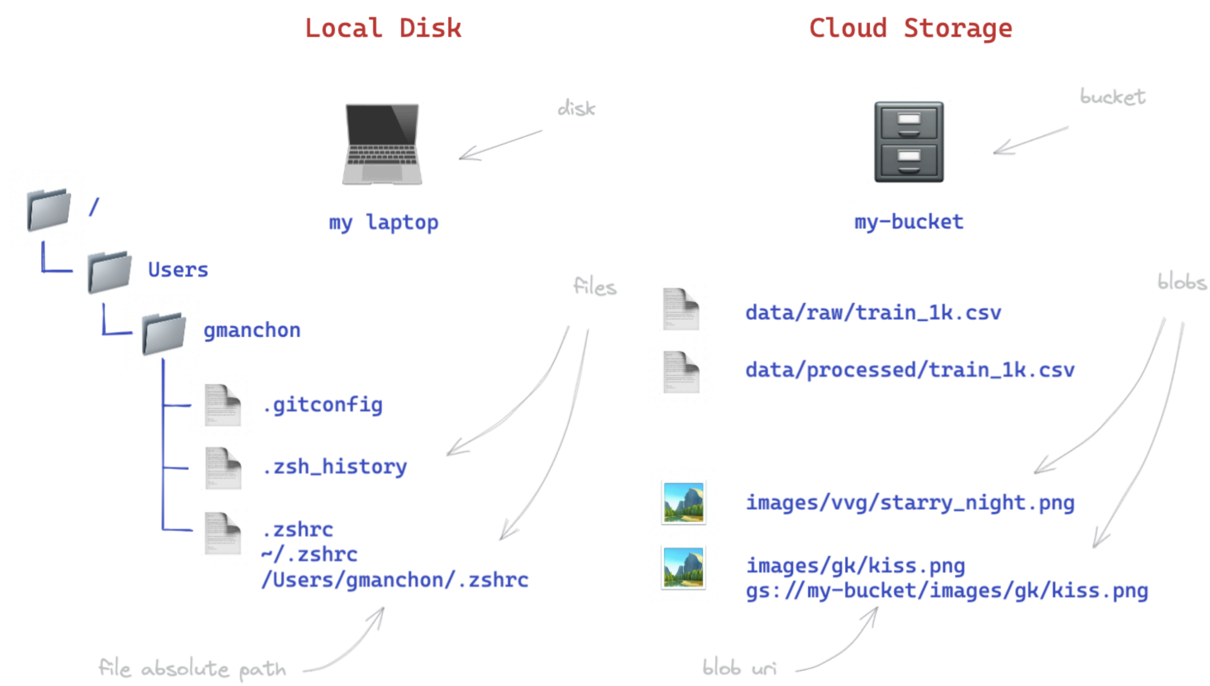
```
storage_filename = "models/random_forest_model.joblib"
local_filename = "model.joblib"

client = storage.Client()
bucket = client.bucket(BUCKET_NAME)
blob = bucket.blob(storage_filename)
blob.upload_from_filename(local_filename)
```

More **features** in the [Cloud Storage API]

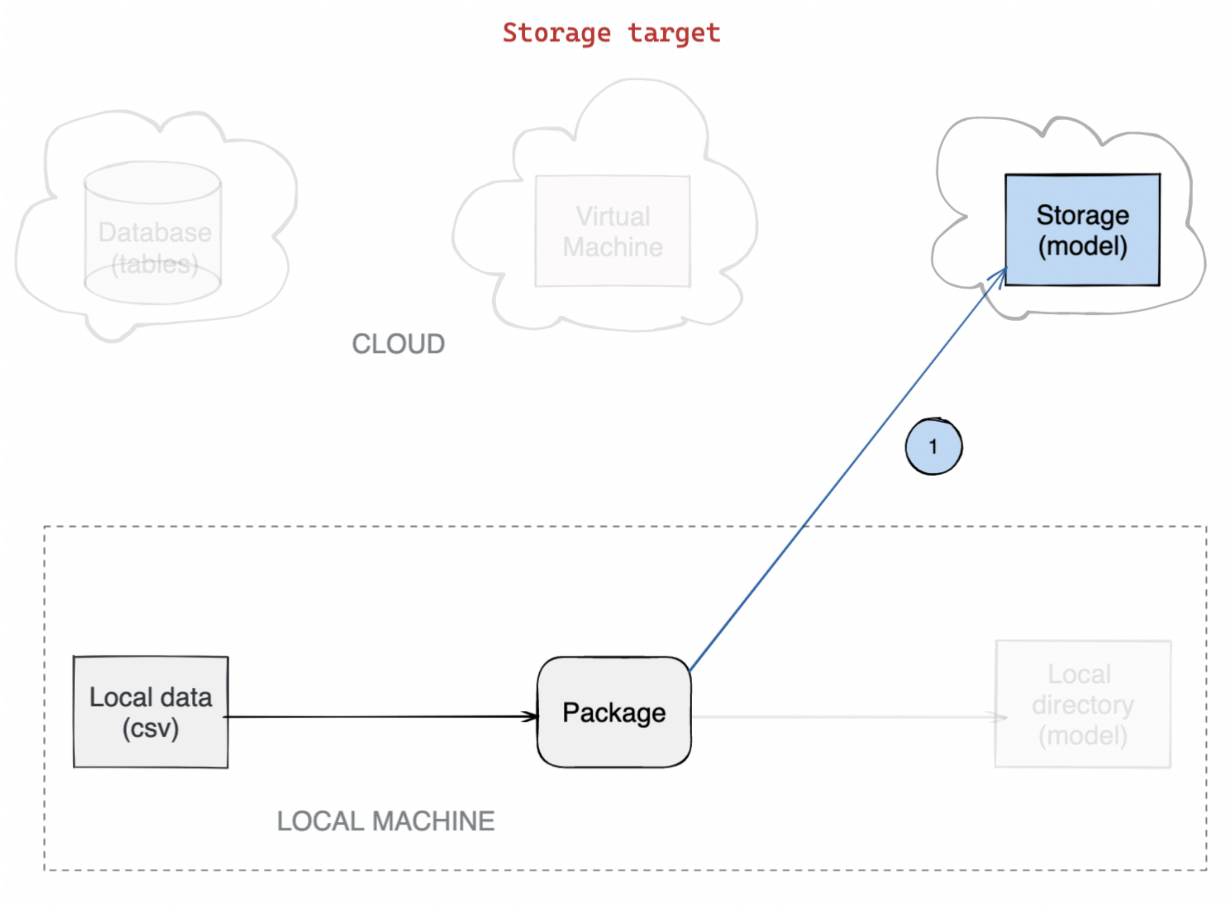
## Theory 🔭

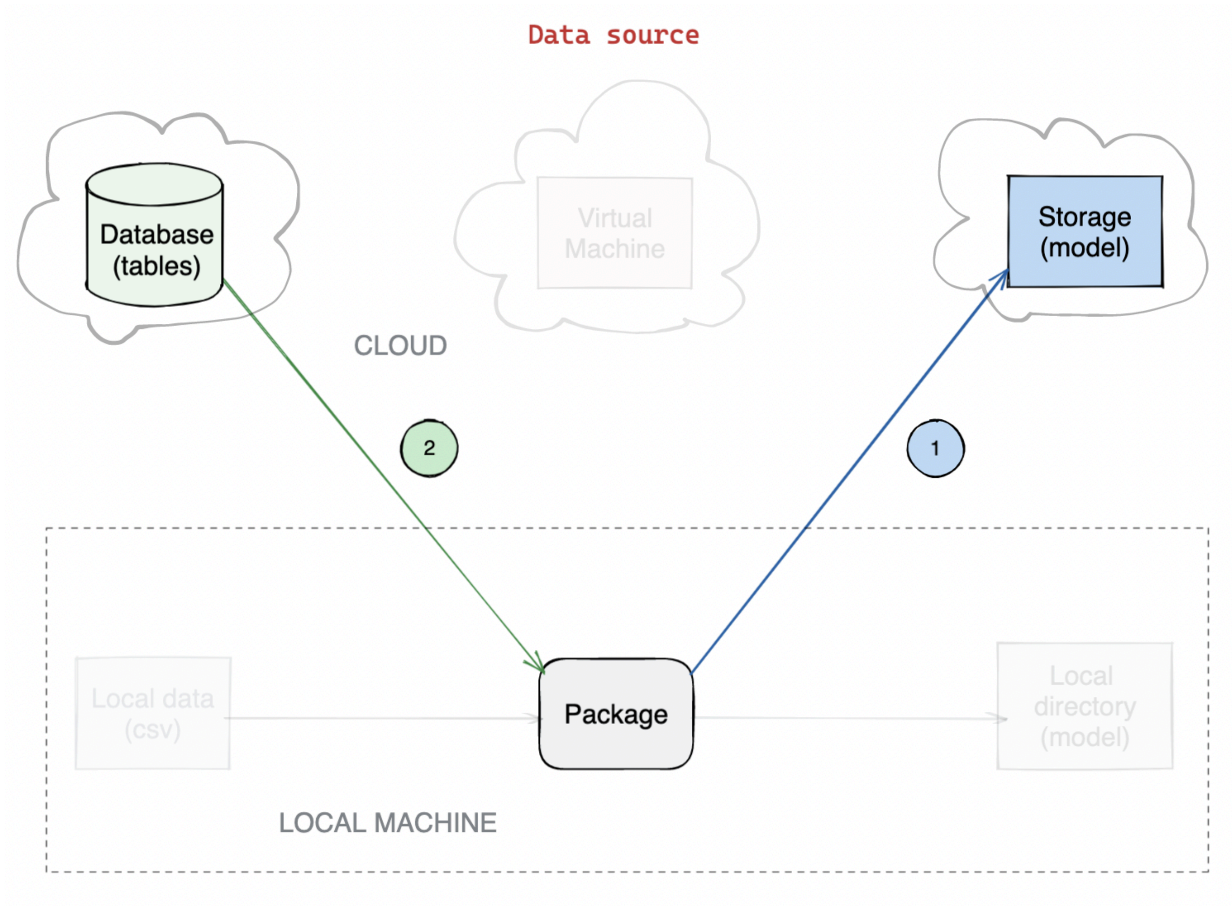
📖 What we saw:

- `Buckets` act as containers for our data
- Each `blob` stores one file




# 6️⃣ Data in the Cloud


Storage target
Database (tables)
Virtual Machine
Storage (model)
CLOUD
1
Local data (csv)
Package
Local directory (model)
LOCAL MACHINE




## Google Big Query

👉 Data warehouse in the cloud

👉 Handles Petabytes worth of data

👉 Built-in ML capabilities (out of scope)

**CODE**

**Load table**

```
from google.cloud import bigquery

PROJECT = "my-project"
DATASET = "taxifare_dataset"
TABLE = "processed_1k"

query = f"""
    SELECT *
    FROM {PROJECT}.{DATASET}.{TABLE}
    """
```

```python
client = bigquery.Client(project=gcp_project)
query_job = client.query(query)
result = query_job.result()
df = result.to_dataframe()
```

More **features** in the [Big Query API]

**Upload dataframe**

```python
from google.cloud import bigquery
import pandas as pd

PROJECT = "my-project"
DATASET = "taxifare_lecture"
TABLE = "lecture_data"

table = f"{PROJECT}.{DATASET}.{TABLE}"

df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})

client = bigquery.Client()

write_mode = "WRITE_TRUNCATE" # or "WRITE_APPEND"
job_config = bigquery.LoadJobConfig(write_disposition=write_mode)

job = client.load_table_from_dataframe(df, table, job_config=job_config)
result = job.result()
```

# Livecode 🚧

🎯 Load a dataframe into to big query table

**Training from the cloud**

💻 Upload a dataframe to BQ

- Create a dataset
- Create a new python script
- Use the upload dataframe code

💻 Verify that the data is uploaded

- Let's use the the *web console* 🌍 and then the *cli*

- Then try changing the `write_mode`

## Theory 🔭

📖 What we saw:

- **Data warehouse** in the cloud
- `datasets` as regular databases
- `tables` store data

## Datasets & Tables

**Database** in the cloud

Best for *structured/relational* data (tabular)

**CLI**

```
DATASET=taxifare_lecture
TABLE=lecture_data

bq ls                    # list datasets
bq ls $DATASET           # list dataset tables

bq show $DATASET.$TABLE  # show table format
```

## Imports & Queries

**CLI**

```
SOURCE=processed_1k.csv

# load data to dataset table
bq load --autodetect $DATASET.$TABLE $SOURCE
```

Legacy SQL vs custom SQL:

```
# show table's first rows
bq query "SELECT * FROM $DATASET.$TABLE LIMIT 5"
```

```
# show table usage
bq query \
  --nouse_legacy_sql \
  "SELECT * FROM $DATASET.INFORMATION_SCHEMA.PARTITIONS"
```

More **commands** in the Big Query cheatsheet

## Query Evaluation 💵

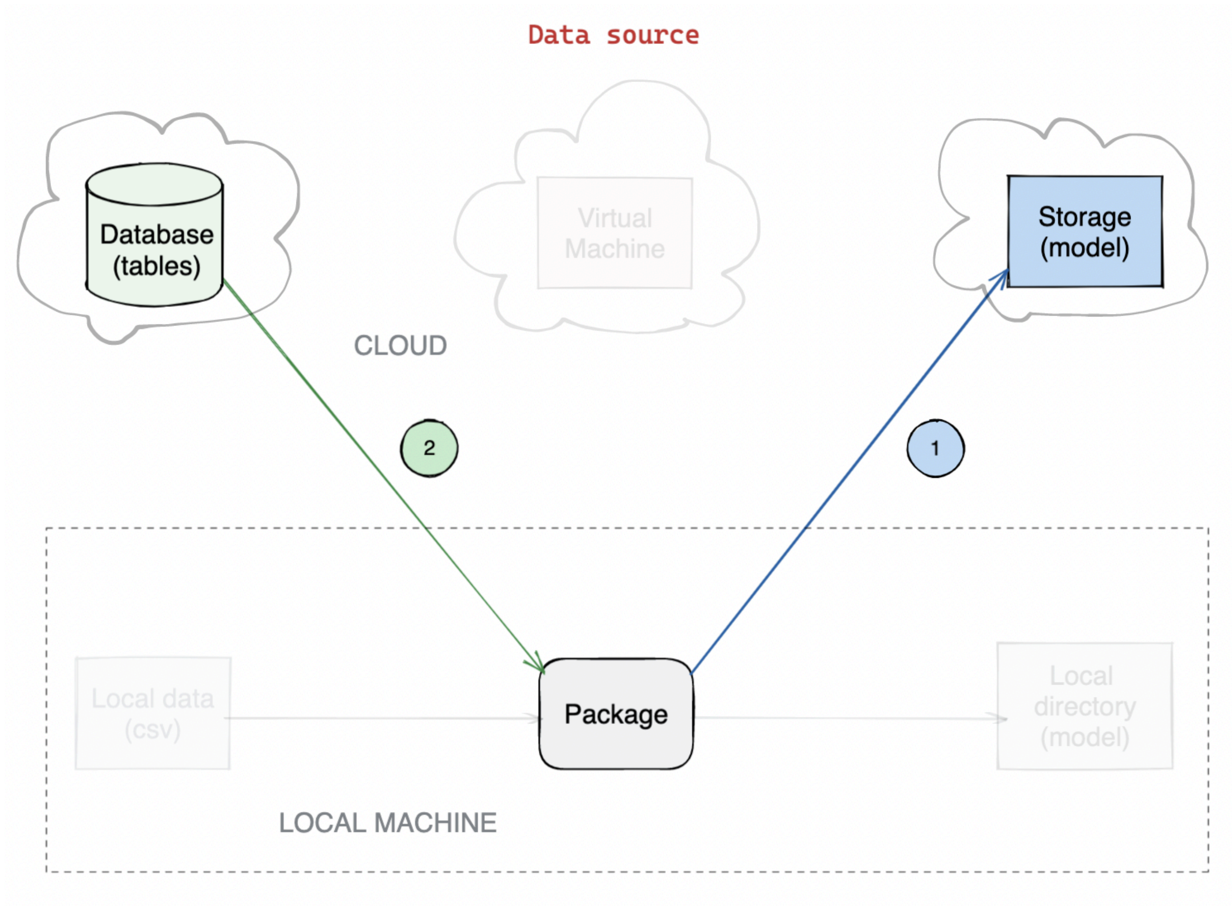
Cost per **scanned data**, ❌ not per returned row

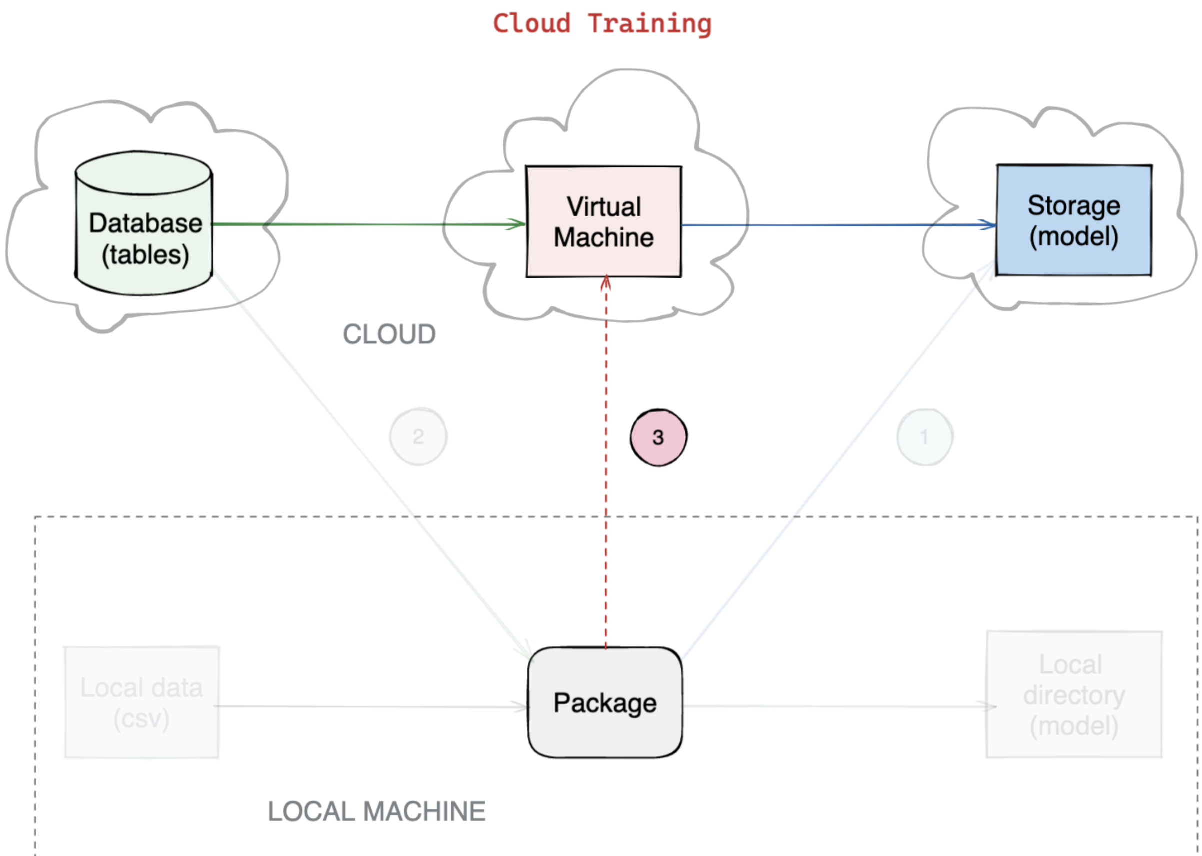
In contrast, managed cloud database services can bill dedicated instances per number of allocated CPUs and amount of memory used, or bill shared instances per uptime, amount of stored data and number of concurrent connections to the database

- Use the **CLI**'s `--dry_run` flag to evaluate the cost of a query
- Use the *web console* 🌍 preview to estimate the amount of data scanned
- Only select required **columns** (columnar storage)
- Use partitions
- Optimize the queries on partitions
- ❌ `LIMIT` does not impact the pricing

More on cost best practices

# 7️⃣ Training in the Cloud


Data source
Database
(tables)
Virtual
Machine
Storage
(model)
CLOUD
2
1
Local data
(csv)
Package
Local
directory
(model)
LOCAL MACHINE




# Livecode 🚧

🎯 Run the package training in the cloud

**Create a Virtual Machine**

💻 Create a *virtual machine* from the *web console* 🌍

- Overview of *region*, *zone*, *hardware*, *price*, *operating system*
- Select the **Ubuntu** *operating system* (*boot disk* section)

**Connect to the Virtual Machine**

💻 Connect to the *virtual machine*

- Start the *VM*
- Connect to the *VM* using `ssh`

- Explore the empty *VM* (no/bad `python` version, no `git` auth)

**Run the Package in the Cloud**

💻 Connect to an already configured *VM*

- Connect to an already setup vm
- `scp` across the lecture livecode directory

💻 Train the model

💻 Stop the virtual machine

# Theory 🔭

📖 What we saw:

- `virtual machines` as computers in the cloud ❌ without a screen
- `VM`s are switched on and off remotely
- `ssh` allows remote connections to a **shell** running in a *VM*, similarly to when a new tab is created in the **Terminal**

# Google Compute Engine

👉 **Virtual machine** in the cloud

👉 Custom **hardware** (GPU) and **operating system** (Ubuntu)

👉 Basic building block of many **Google Cloud Platform** products

# Virtual Machine

**CLI**

```
INSTANCE=my-instance

gcloud compute instances list                    # list virtual machine's status

gcloud compute instances start $INSTANCE        # start instance
gcloud compute instances stop $INSTANCE          # stop instance
```

🚨 🚨 🚨 Save the planet 🌱, schedule an auto shutdown for your VM 🚨 🚨 🚨

# Remote Operation

Interact with a remote machine:

- `ssh` to run commands in a shell on the remote machine through the SSH protocol
- `scp` to copy files from and to the machine

**CLI**

```
# interactive ssh to remote instance
gcloud compute ssh $INSTANCE

# run remote commands on remote instance
gcloud compute ssh $INSTANCE --command "ls -la"

# recursively copy home directory on instance to local directory
gcloud compute scp --recurse $INSTANCE:~/ .
```

More **commands** in the Compute Engine cheatsheet

# Your turn 🚀