Hybrid Cloud Preprocessing Script

📘 File

`preprocess.py`

🧠 Overview

This script preprocesses **Google Cluster Trace–style datasets** into a structured format suitable for **machine learning–based workload classification** in the **Hybrid Cloud Orchestrator** project.

Every time the script runs:

- It randomly samples **100 rows** (for performance and variability).
- Extracts, normalizes, and derives workload characteristics.
- Automatically detects and scales **time units**.
- Estimates missing values, such as **data size**.
- Saves a **clean, feature-rich CSV** ready for ML training and inference.

🧩 Input

A CSV file from **Google Cluster Data**, containing columns such as:

```
start_time, end_time, resource_request, assigned_memory,
priority, scheduling_class, collection_type, collection_name,
failed, ...
```

These fields typically represent event logs of resource allocations, usage patterns, and job metadata.

🧮 Output

A clean dataset with **6 engineered features**:

Column	Description	Type	Example
`cpu cores`	Number of CPU cores required	Float	`0.21`, `1.55`, `4.22`
`memory mb`	Memory footprint in MB	Float	`341.5`, `879.0`, `2048.0`
`latency sensitive`	Whether the workload is latency-critical	Integer (0/1)	`1`
`execution time`	Duration of the task in seconds	Float	`3.0`, `300.0`, `1200.0`
`data size mb`	Estimated data processed in MB	Float	`45.2`, `912.8`
`cost traditional`	Estimated cost for VM/Kubernetes deployment	Float	`0.0005, 0.034`
`cost serverless`	Estimated cost for serverless function deployment	Float	`0.00002, 0.067`
`cost ratio`	Ratio (serverless/traditional) — cost efficiency indicator	Float	`21.5, 0.9`
`target platform`	ML label: `serverless` or `traditional`	String	`serverless`

⚙️ How It Works — Step-by-Step

1️⃣ Sampling

Randomly selects **100 rows** from the dataset for preprocessing:

```
df_sample = df_raw.sample(n=100)
```

- Ensures quick iteration and balanced workload examples per run.

2️⃣ Parsing Resource Requests

Google cluster data encodes CPU and memory as fractional values of the machine’s total capacity, e.g.:

```
{'cpus': 0.020660400390625, 'memory': 0.014434814453125}
```

Computation:

CPU cores = cpu_fraction × machine_cores

Memory (MB) = memory_fraction × machine_memory_MB

Default:

- `machine_cores = 16`
- `machine_mem_mb = 65536` (≈ 64 GB)

Example:

```
{'cpus': 0.02066, 'memory': 0.01443}
```

becomes:

CPU :
0.02066 × 16 = 0.33 cores

Memory :
0.01443 × 65536 = 945 MB

If numeric values are larger than 1, they’re treated as **absolute units**.

3️⃣ Memory Normalization

If `assigned_memory` exists, it’s used to fill missing memory values.
Heuristics:

- Values ≤ 1 → fractions (× total machine memory)

- Values < 10000 → MB
- Very large numbers → bytes (converted via ÷ 1024^2)

4️⃣ Execution Time Calculation

Google trace timestamps (`start_time`, `end_time`) are large because they're often in **nanoseconds**.
The script automatically detects the unit and normalizes it to **seconds**.

Detection Logic:

```
if median_diff > 1e9:  # nanoseconds
    diff = diff / 1e9

elif median_diff > 1e6:  # microseconds
    diff = diff / 1e6

elif median_diff > 1e3:  # milliseconds
    diff = diff / 1e3
```

Mathematical Formula:
Execution Time (s) = (end_time − start_time) / k

Where $k \in \{1, 10^3, 10^6, 10^9\}$ depending on detected units.

Example:

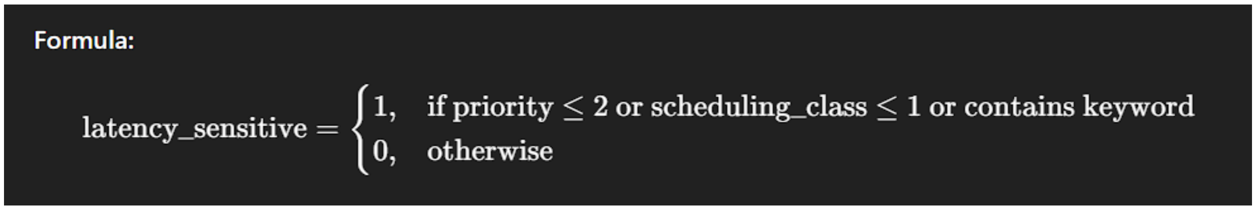
```
start_time = 2.748E+11
end_time   = 2.751E+11
diff = 3E+08 → divided by 1E+9 = 3.0 seconds
```

5️⃣ Latency Sensitivity Detection

Heuristic rules based on job metadata:

Field	Rule	Meaning
`priority`	≤ 2 → 1	High priority → latency-sensitive
`scheduling_class`	≤ 1 → 1	Real-time or interactive
`collection_name` / `collection_type`	Contains “api”, “realtime”, “interactive”, “latency”, “foreground” → 1	Textual cues for latency-bound tasks
Otherwise	0	Background or batch jobs

Formula:

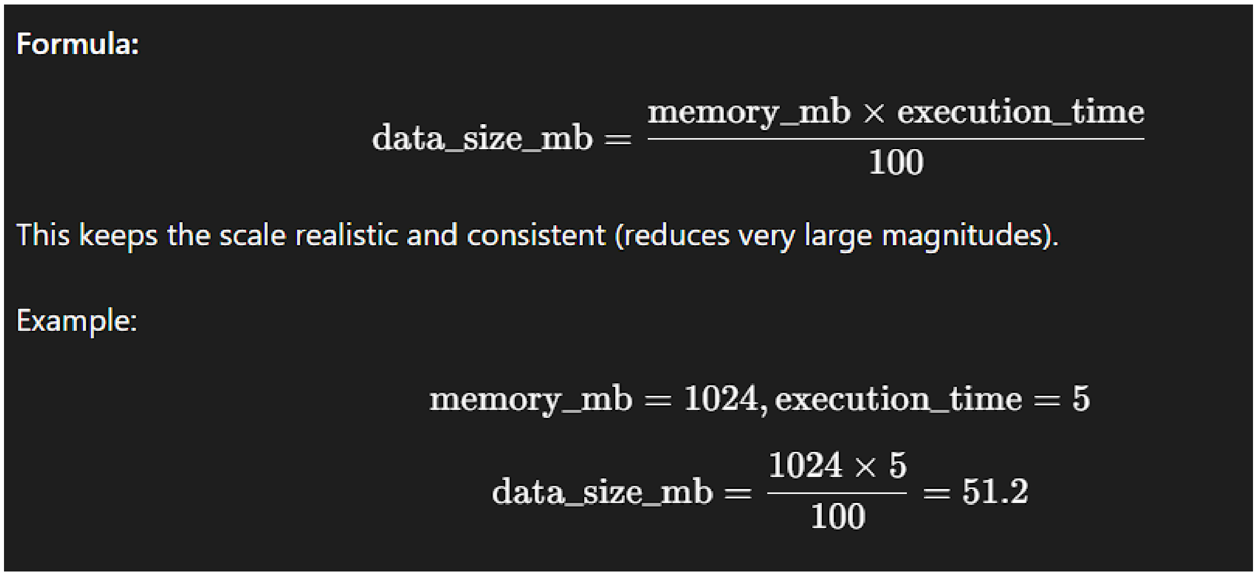


Formula:

$$\text{latency\_sensitive} = \begin{cases} 1, & \text{if priority} \leq 2 \text{ or scheduling\_class} \leq 1 \text{ or contains keyword} \\ 0, & \text{otherwise} \end{cases}$$

6️⃣Data Size Estimation

Google dataset doesn’t contain explicit data size..



Formula:

$$\text{data\_size\_mb} = \frac{\text{memory\_mb} \times \text{execution\_time}}{100}$$

This keeps the scale realistic and consistent (reduces very large magnitudes).

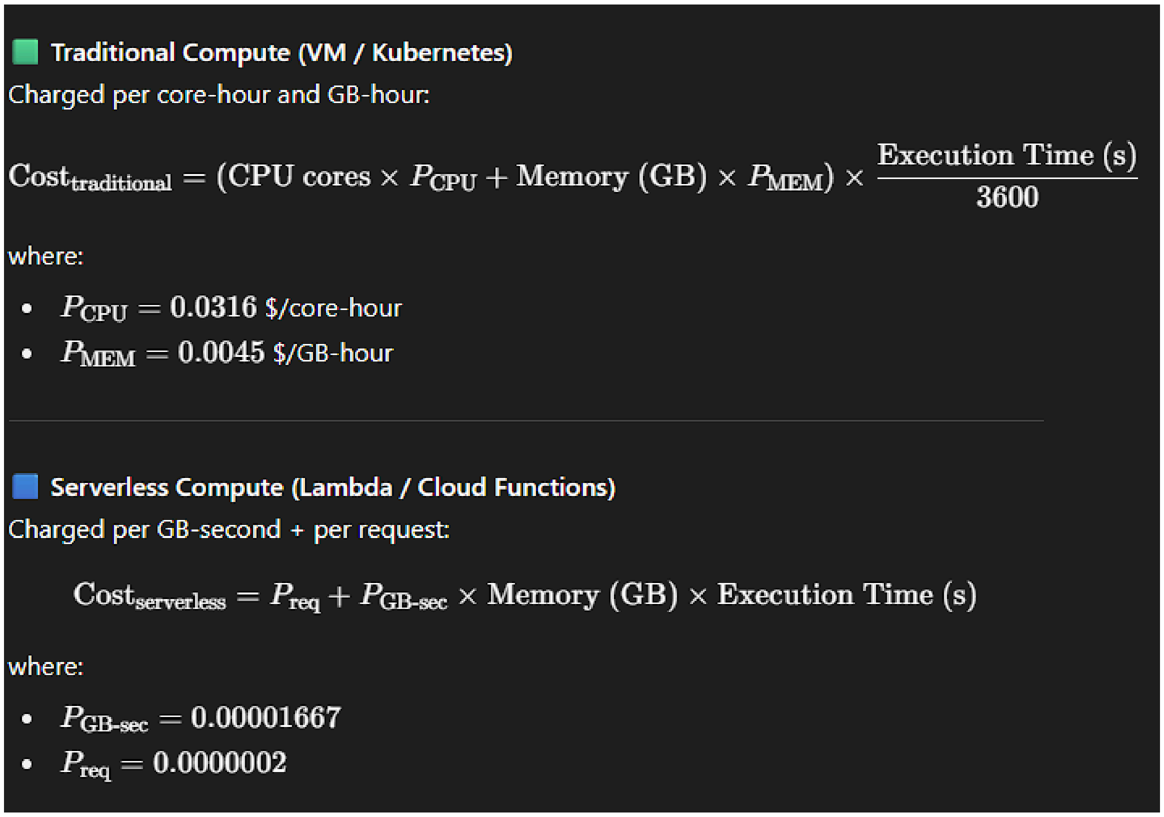
Example:

$$\text{memory\_mb} = 1024, \text{execution\_time} = 5$$

$$\text{data\_size\_mb} = \frac{1024 \times 5}{100} = 51.2$$

7️⃣💰 Cost Estimation

This section estimates both **Traditional** and **Serverless** costs for each workload.



🟩 **Traditional Compute (VM / Kubernetes)**

Charged per core-hour and GB-hour:

$$\text{Cost}_{\text{traditional}} = (\text{CPU cores} \times P_{\text{CPU}} + \text{Memory (GB)} \times P_{\text{MEM}}) \times \frac{\text{Execution Time (s)}}{3600}$$

where:

- $P_{\text{CPU}} = 0.0316$ $/core-hour
- $P_{\text{MEM}} = 0.0045$ $/GB-hour

🟦 **Serverless Compute (Lambda / Cloud Functions)**

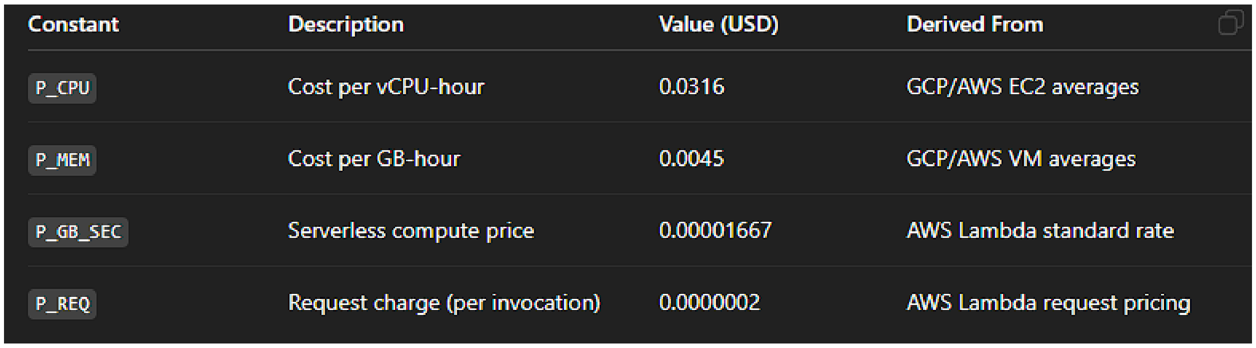
Charged per GB-second + per request:

$$\text{Cost}_{\text{serverless}} = P_{\text{req}} + P_{\text{GB-sec}} \times \text{Memory (GB)} \times \text{Execution Time (s)}$$

where:

- $P_{\text{GB-sec}} = 0.00001667$
- $P_{\text{req}} = 0.0000002$

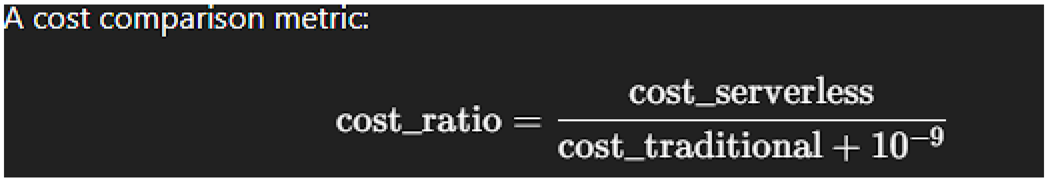
Pricing is not random.



Constant	Description	Value (USD)	Derived From
`P_CPU`	Cost per vCPU-hour	0.0316	GCP/AWS EC2 averages
`P_MEM`	Cost per GB-hour	0.0045	GCP/AWS VM averages
`P_GB_SEC`	Serverless compute price	0.00001667	AWS Lambda standard rate
`P_REQ`	Request charge (per invocation)	0.0000002	AWS Lambda request pricing

⚖️ Cost Ratio

A cost comparison metric:



A cost comparison metric:

$$\text{cost\_ratio} = \frac{\text{cost\_serverless}}{\text{cost\_traditional} + 10^{-9}}$$

Ratio Value	**Interpretation**	**Meaning in Context**
< 1	**Serverless cheaper**	The same job costs **less** on serverless than on a VM. → Ideal for bursty, short workloads.
= 1	**Equal cost**	Both environments cost roughly the same. Decision may depend on latency or management preference.
> 1	**Traditional cheaper**	Serverless costs **more** than running on a VM — typically long or compute-heavy jobs.

8️⃣ Failure Normalization

Converts textual failure states into boolean flags.

Input	Output
`1`, `true`, `y`, `fail`, `failed`, `failure`	`True`
`0`, empty, others	`False`

This is used in target classification.

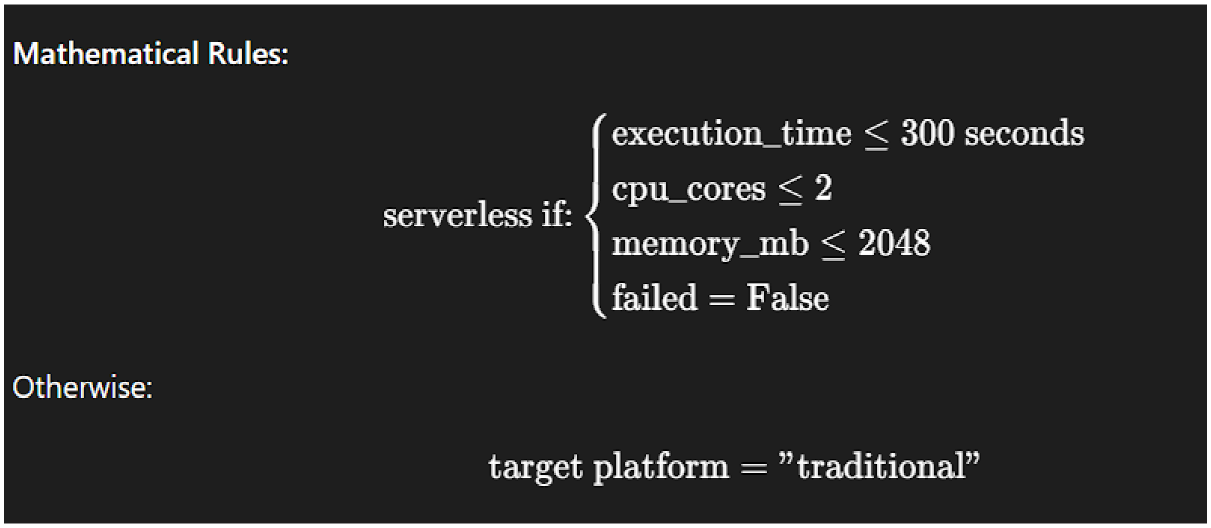
9️⃣ Target Platform Classification

The **core ML label** (`serverless` or `traditional`) is determined heuristically:

Logic:

A workload is considered **serverless** if:

- Short-lived
- Lightweight
- Memory-efficient



Mathematical Rules:

$$\text{serverless if: } \begin{cases} \text{execution\_time} \leq 300 \text{ seconds} \\ \text{cpu\_cores} \leq 2 \\ \text{memory\_mb} \leq 2048 \\ \text{failed} = \text{False} \end{cases}$$

Otherwise:

$$\text{target platform} = \text{"traditional"}$$

Sampling and Saving

- Randomly samples 100 rows each run (new subset each execution).

Writes processed dataset to:

`workload_dataset_sample.csv`

- File automatically **overwrites** if it already exists (for versioning simplicity).

📊 Example Output

cpu cores	memory mb	latency sensitive	execution time	data size mb	cost traditional	cost serverless	cost ratio	target platform
0.5	512	1	5	25.6	0.000002	0.000043	21.5	serverless
2	2048	0	300	6144	0.0016	0.102	63.75	traditional
8	8192	0	3600	294912	0.236	491.52	2081.0	traditional

⚙️ Default Parameters

Parameter	Default	Description
`--cores`	16	Number of CPU cores per machine
`--mem_mb`	65536	Machine total memory in MB
`--input`	*required*	Input dataset CSV
`--output`	`workload_dataset_sample100.csv`	Output file
`--seed`	None	Random seed for reproducible sampling

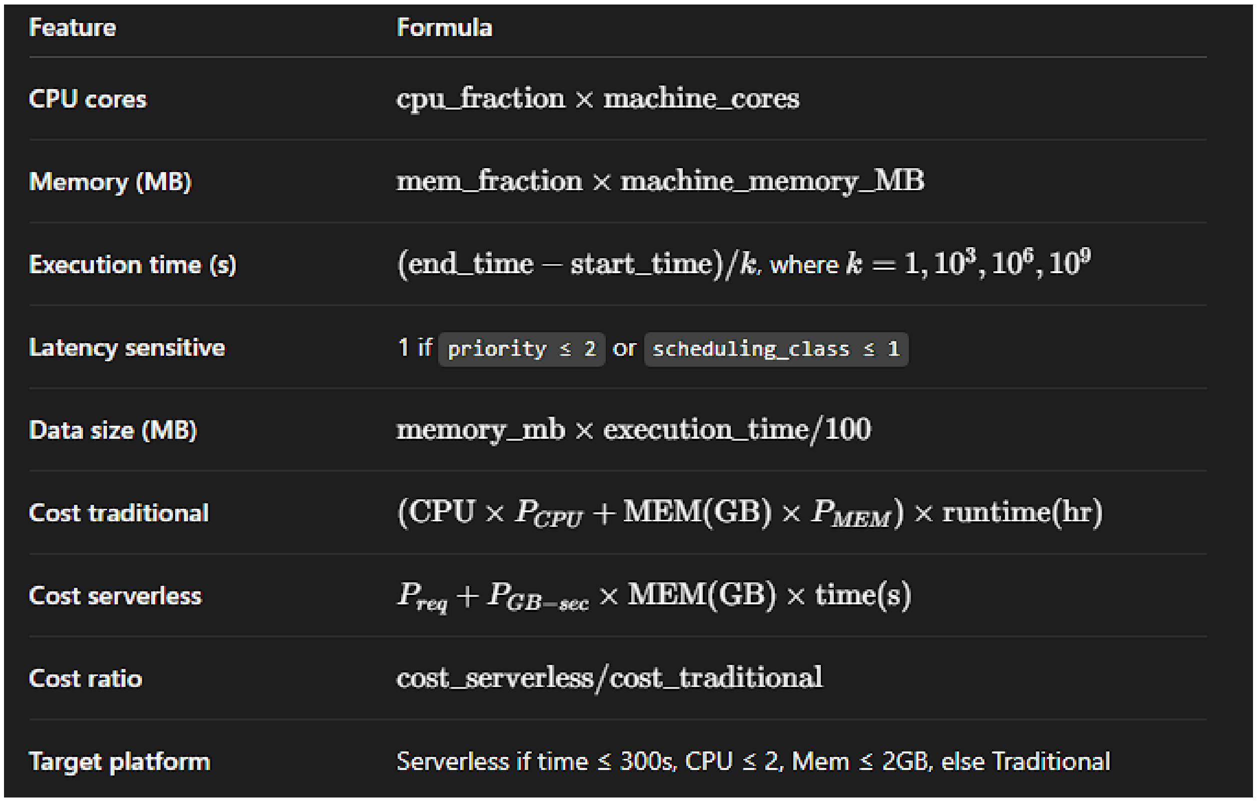
🧩 Example Command

```
python preprocess.py --input Sample_Dataset.csv --output
workload_dataset_sample.csv
```

or, with explicit configuration:

```
python sample_and_preprocess.py --input google_dataset.csv --output
workload_dataset_sample100.csv --cores 32 --mem_mb 131072
```

🧠 Summary of All Math & Logic



Feature	Formula
CPU cores	$\text{cpu\_fraction} \times \text{machine\_cores}$
Memory (MB)	$\text{mem\_fraction} \times \text{machine\_memory\_MB}$
Execution time (s)	$(\text{end\_time} - \text{start\_time})/k$, where $k = 1, 10^3, 10^6, 10^9$
Latency sensitive	1 if `priority ≤ 2` or `scheduling_class ≤ 1`
Data size (MB)	$\text{memory\_mb} \times \text{execution\_time}/100$
Cost traditional	$(\text{CPU} \times P_{CPU} + \text{MEM(GB)} \times P_{MEM}) \times \text{runtime(hr)}$
Cost serverless	$P_{req} + P_{GB-sec} \times \text{MEM(GB)} \times \text{time(s)}$
Cost ratio	$\text{cost\_serverless}/\text{cost\_traditional}$
Target platform	Serverless if time ≤ 300s, CPU ≤ 2, Mem ≤ 2GB, else Traditional

---

🧩 Purpose in the Project

This script is the **first stage of the intelligent hybrid cloud orchestration pipeline** — it converts raw Google cluster traces into **structured, ML-ready workload features**, enabling the **ML classifier** to learn how to route workloads between:

- 🟢 **Serverless platforms** (e.g., AWS Lambda, GCP Functions)
- 🔵 **Traditional infrastructure** (e.g., EC2, Kubernetes)