1. What is Daft?

Daft is an open-source distributed data engine built to handle **multimodal** (i.e., many types) data at scale like structured tables (like CSV), unstructured data (images, text, audio, embeddings) all under one framework.
It offers a DataFrame-style API (for Python) and/or SQL interface, implemented with high-performance underpinnings (Rust + Apache Arrow format) to get performance and scalability.
Rust → The programming language used to build the engine that powers Daft.
Apache Arrow → A **columnar memory format** for data.
It targets both local/small scale experimentation **and** large distributed clusters (cloud scale)
The company behind it, Eventual, has raised significant funding (US$20–30 M) to build this "AI-native data infrastructure" for the multimodal era.

2. Why DAFT ?
"Why use Daft instead of existing tools (Pandas, Spark)?

- **Multimodal support**: Many traditional tools focus on tabular data (rows & columns). But modern AI/ML workflows often include images, audio, nested JSON, embeddings, video, etc. Daft explicitly supports those: images, embeddings, tensors, URLs, etc.
- **Performance & scale**: According to benchmarks published, Daft performed much faster (e.g., ~3–7× faster than Spark, Dask in some tests) for some large-scale queries, and was
- able to handle out-of-core (data larger than memory) reliably.
- **Unified workflow**: If you have a pipeline that mixes "load images, decode, embedding, join with tabular data, filter, feature extraction, feed to ML", Daft allows you to stay in the same ecosystem rather than stitching many specialized tools.
- **Modern architecture**: Python-native API for data scientists (so less friction), but the backend is Rust + Arrow (for memory/IO efficiency). Also works locally and scales transparently.
- **Cloud / enterprise readiness**: Integrations with data lakes, storage (S3, GCS), table formats (Apache Iceberg, Delta Lake), catalogs (AWS Glue, Unity Catalog) show that it's meant for serious enterprise data workloads

3. Key Features & Capabilities

- **Native multimodal data types**: Columns can contain images, tensors, URLs, embeddings, nested data. So operations like "decode images", "resize", "extract embeddings", "apply UDFs" become more natural.
- **Python-first API**: While you can use SQL, the primary interface is Python DataFrame-style.
- **Distributed & scalable execution**: You can start on one machine, and later run on a cluster of machines without rewriting your code. Daft manages partitioning, I/O, memory management.
- **High performance I/O and memory management**: Built to avoid "Out-Of-Memory" issues, optimize I/O, memory usage, and support data larger than memory ("out-of-core").
- **An Out of Memory (OOM) issue happens when your computer or server runs out of RAM (Random Access Memory) while processing data.**
- **Integration with modern data lakehouse architectures**: Works with table formats (Iceberg, Delta Lake), works with catalogs (Unity Catalog) for governance and enterprise-scale data.

A **data catalog** stores **metadata** — data about your data.
It doesn't store the actual data, but rather:

- The **table names**
- **Schema** (column names, data types)
- **Location** (e.g., S3 bucket path, database URI)
- **Partitions**
- **Ownership / access control**
- **Versioning information**

Open-source engine + enterprise ambitions: While the engine is open source, the company is building "Eventual Cloud", an enterprise offering. So there's both a community and commercial trajectory.

A Parquet file is an open-source, column-oriented data file format optimized for efficient data storage and fast retrieval, especially for big data analytics

Why "Anonymous Data Access Mode" Exists

When you work with **cloud storage** (S3, GCS, Azure Blob), there are two kinds of access:

1. **Authenticated access** → You use credentials (API keys, IAM roles, etc.)
   ✅ Used for private data.
   ❌ Requires setup, permissions, and authentication tokens.
2. **Anonymous access** → You read **publicly available data** without credentials.
   ✅ Used for public datasets or demos.
   ❌ You can't modify or access private files.

So, when Daft says

> "Set I/O configurations to use anonymous data access mode"

…it means:

> "We're reading data from a **public bucket** (like a sample dataset on S3), so don't expect credentials — just connect anonymously."

The Message:

> **"(No data to display: DataFrame not materialized)"**

This means:

> The DataFrame exists **logically** (i.e., Daft knows *what to do* with it),
> but it hasn't actually **executed the computation** yet.

Daft — like Spark — uses **lazy evaluation** (also called *lazy execution*).

Daft doesn't actually read or process the data *immediately*.
Instead, it builds a **plan** — a graph of operations that says:

> "Okay, I'll read a CSV, filter it, and select these columns… but I'll wait until the user actually *needs* the result."

So at this point, the DataFrame is not yet *materialized* — i.e., data hasn't been loaded into memory or computed.

When It *Does* Materialize

The data gets **materialized** (actually read, computed, and shown) only when you run an *action* such as:

```
df.show()
df.collect()
df.to_pandas()
df.write_parquet("output.parquet")
```

These are "trigger" operations that tell Daft:

> "Now go execute all the transformations and give me the actual results."

If you run just `print(df)` or try to visualize it *before* calling those, you'll get:

> "Expressions are an API for defining computation that needs to happen over columns,"

it means:

> In **Daft (or similar DataFrame systems like Spark or Pandas)**, an **Expression** is how you *describe what computation* should be performed **on data columns** — before actually running it.

When we say:

> "Expressions are an API for defining computation over columns,"

we mean:

> Daft gives you a **programming interface (a set of functions, classes, and operators)** that lets you **describe computations** — like filters, aggregations, or mathematical operations — on your data columns.

You're **not** manually looping over rows or writing SQL — you're using Daft's API to *express* what should happen.

When Daft says:

> "Transform columns with custom logic,"

it means:

> You can apply your **own Python functions or ML models** to DataFrame columns — not just use the built-in operations (like sum, mean, etc.).

Why This Is Useful

Most real-world data tasks involve domain-specific transformations like:

- Text **tokenization** (for NLP)
- **Inference** using ML models (like BERT, GPT, etc.)

- Cleaning or normalizing custom data

Daft lets you do this using its **apply** or **UDF (User Defined Function)** mechanism.

Example - **Tokenizing Sentences Using Hugging Face Transformers**

Here's a simple example showing how to use the Hugging Face AutoTokenizer to tokenize a column of text inside Daft

Step-by-Step Explanation

- **Daft** → A **high-performance DataFrame library** (like Pandas, but distributed & faster).
- **AutoTokenizer** → Comes from Hugging Face's Transformers library; it automatically loads the correct tokenizer for any pre-trained model (like BERT, GPT, etc.).

🧱2️⃣Creating a Daft DataFrame

- df = daft.from_pydict({
-     "text": [
-         "Daft is a high-performance DataFrame library.",
-         "It supports distributed computation and ML integration.",
-         "You can use custom logic like tokenization!"
-     ]
- })
- This creates a Daft DataFrame (df) from a **Python dictionary**.
- The column is named "text", and it holds 3 sentences.
- Think of it like a Pandas DataFrame — each row is a text string.

🤖3️⃣Loading the BERT Tokenizer

- tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
- Loads the **pretrained tokenizer** for bert-base-uncased from Hugging Face.
- This tokenizer converts text into **tokens (subword units)** and then **token IDs (numbers)** that the model understands.

Example:

“Daft is great” → [101, 6123, 2003, 2307, 102]
(Here, 101 and 102 are special start/end tokens.)

🧮4️⃣Defining the Tokenization Function

- def tokenize_text(t):
-     print(f"Text: {t}")
-     tokens = tokenizer.tokenize(t)
-     encoded = tokenizer.encode(t, truncation=True, padding="max_length", max_length=16)
-     print(f"Tokens: {tokens}")
-     print(f"Embeddings (IDs): {encoded}\n")
-     return encoded

This function does 3 main things:

1. **Prints the original text**
   → just for clarity when running the code.

- **Tokenizes the text**

  tokens = tokenizer.tokenize(t)

2.
   - Splits text into smaller units (like words or word pieces).
     Example: "Daft is cool" → ['daft', 'is', 'cool']

- **Encodes the text**

  encoded = tokenizer.encode(t, truncation=True, padding="max_length", max_length=16)

3.
   - Converts tokens into **numeric IDs** (embeddings that models understand).
   - truncation=True → Cuts off long text beyond 16 tokens.
   - padding="max_length" → Pads short text with zeros up to 16 tokens.

    ○ max_length=16 → Fixes the embedding length at 16.

4. **Prints the tokens and embeddings**, then returns the embeddings (a list of integers).

🧬5️⃣Applying Tokenization to Each Row

- df = df.with_column(
-    "embeddings",
-    df["text"].apply(tokenize_text, return_dtype=daft.DataType.list(daft.DataType.int64()))
-)

Here's what happens:

- df["text"].apply(...) → runs tokenize_text() **on each row** in the text column.
- Each text string goes through the tokenizer function individually.
- The **result** (a list of integers) is added as a **new column** named "embeddings".
- return_dtype tells Daft the data type of the result: daft.DataType.list(daft.DataType.int64())
- meaning "a list of 64-bit integers."

So after this line:

text	**embeddings**
Daft is a high-performance DataFrame library.	[101, 6123, 2003, …, 0, 0]
It supports distributed computation and ML integration.	[101, 2009, 4672, …, 0, 0]

You can use custom logic like tokenization!	[101, 2017, 2064, …, 0, 0]

🖥️6️⃣Displaying the DataFrame

- df.show()
- Displays the Daft DataFrame in a readable table format.
- You’ll see your original text plus the new "embeddings" column containing token IDs.

⚙️ Overall Flow Summary

1. You load text data into a Daft DataFrame.
2. You load a pretrained tokenizer (BERT).
3. You define a function that:
   - Prints the text
   - Tokenizes it
   - Encodes it to numerical embeddings

4. You apply that function to each row and store the results in a new column.
5. You display the final DataFrame.

Sentiment analysis

High-Level Summary: How Daft Improves Sentiment Analysis

1️⃣It’s faster and scalable
Daft can process huge amounts of text in parallel across multiple CPU cores or even multiple machines.
So instead of analyzing one review at a time, it handles thousands simultaneously.

2️⃣It integrates directly with ML models
You can plug in models from libraries like Hugging Face, and Daft will handle the batching, tokenization, and inference efficiently under the hood.

3️⃣ **It keeps everything in one clean pipeline**
You can chain steps like cleaning text, tokenizing, and predicting sentiment together — Daft figures out the most efficient way to run them.

4️⃣ **It's optimized for modern data formats**
Daft uses Apache Arrow (a super-fast columnar data format) and Rust, meaning it's much faster and more memory-efficient than Pandas or vanilla Python loops.

5️⃣ **It's flexible and explainable**
You can easily add extra columns for confidence scores, SHAP explanations, or model metrics — all within the same Daft DataFrame.

🧠 In short:

> "Daft makes sentiment analysis not just possible — but **scalable, efficient, and production-ready**. It combines the simplicity of Python with the power of distributed data processing."

1️⃣ What Daft Is

Daft is a **distributed data processing framework**, like **Pandas + Spark + Polars combined**, but built for **Python and AI workloads**.

It lets you:

- Create **dataframes** (like Pandas)
- Run **parallel transformations** across multiple CPU cores (and even clusters)
- Natively integrate with **Python functions**, **machine learning models**, and **deep learning pipelines**

So think of Daft as your **smart, parallel spreadsheet for AI pipelines**.

🧠2️⃣ What Hugging Face Is Doing Here

This line:

```
from transformers import pipeline
```

loads **Hugging Face Transformers**, which gives pre-trained NLP models for:

- Sentiment analysis
- Translation
  Summarization
- Text generation, etc.

Here, you load a **3-class sentiment model**:

```
sentiment_pipeline = pipeline(
    "sentiment-analysis",
    model="cardiffnlp/twitter-roberta-base-sentiment"
)
```

That model outputs one of:

- LABEL_0 → NEGATIVE
- LABEL_1 → NEUTRAL
- LABEL_2 → POSITIVE

So it can detect neutral tones — unlike the default SST-2 model.

⚙️3️⃣The Function That Wraps Hugging Face

You define a small helper function:

```
def hf_sentiment(text: str) -> str:
    result = sentiment_pipeline(text)[0]
    return label_map[result["label"]]
```

What happens here:

`sentiment_pipeline(text)` runs your text through RoBERTa,
returning a dictionary like:

```
[{'label': 'LABEL_2', 'score': 0.9812}]
```

Then you map that numeric label to something readable:

```
label_map = {"LABEL_0": "NEGATIVE", "LABEL_1": "NEUTRAL", "LABEL_2":
"POSITIVE"}
```

- Finally, you return "POSITIVE", "NEGATIVE", or "NEUTRAL" as a string.

⚡4️⃣ The Daft UDF (User-Defined Function)

This line is the key:

```
sentiment_udf = daft.func(hf_sentiment, return_dtype=str)
```

Daft wraps your Python function hf_sentiment() into a **UDF** —
a *vectorized operation* that Daft can distribute across data chunks.

That means:

- Daft can split your dataframe into partitions.
- Run hf_sentiment() in **parallel** on each partition (multi-core or multi-node).
- Recombine the results into one dataframe column.

This is similar to how Spark or Dask would distribute work — but Daft is lighter, simpler, and more Pythonic.

🧾5️⃣ Creating the DataFrame

You create a Daft dataframe from a Python dictionary:

```
df = daft.from_pydict({
    "review": [
        "I absolutely loved this product!",
        "The service was terrible and I'm disappointed.",
        "It works as expected, nothing special.",
        "Just okay, not too bad, not great either."
    ]
})
```

🪄6️⃣Applying the Sentiment Function

Now the magic line:

```
df = df.with_column("sentiment", sentiment_udf(df["review"])).collect()
```

Step-by-step:

1. .with_column("sentiment", ...)
   adds a **new column** called sentiment by applying your UDF to the "review" column.

2. sentiment_udf(df["review"])
   tells Daft: "Apply the function to every row's review text."

3. .collect()
   triggers execution — Daft actually runs the pipeline, applies the UDF in parallel, and brings the results back into memory.

🚀 TL;DR: How Daft + HF Work Together

Role	Tool	What It Does
Text understanding	🤗 Hugging Face	Runs a pre-trained model (RoBERTa) for sentiment classification

Data orchestration	⚡ Daft	Distributes your function across all data entries (parallel processing)
Integration bridge	🧠 daft.func()	Converts your Python/HF function into a Daft-compatible, scalable UDF

💬 **Analogy**

Imagine you have 10,000 reviews.

- Hugging Face is the *brain* — the model that understands each review.
- Daft is the *body* — it splits the work across multiple processors.
- Together, they give you **fast, scalable AI-powered dataframes**.

What Is "Batch Inference"?

Batch inference = Running your model (e.g., sentiment analysis, image classifier, tabular predictor) **on many rows at once** — not one by one.

Instead of:

```
for row in df:
    prediction = model.predict(row)
```

Daft does:

```
# Parallel, batched, vectorized inference
df.with_column("prediction", model_udf(df["text"]))
```

Under the hood, Daft:

1. Splits the data into partitions (chunks)
2. Sends batches of data to each worker
3. Runs your model function **on an entire**

When to use Daft for batch inference#

You need to run models over your data: Express inference on a column (e.g., llm_generate, embed_text, embed_image) and let Daft handle batching, concurrency, and backpressure.

You have data that are large objects in cloud storage: Daft has record-setting performance when reading from and writing to S3, and provides flexible APIs for working with URLs and Files.

You're working with multimodal data: Daft supports datatypes like images and videos, and supports the ability to define custom data sources and sinks and custom functions over this data.

You want end-to-end pipelines where data sizes expand and shrink: For example, downloading images from URLs, decoding them, then embedding them; Daft streams across stages to keep memory well-behaved

Core idea#

Daft provides first-class APIs for model inference. Under the hood, Daft pipelines data operations so that reading, inference, and writing overlap automatically, and is optimized for throughput.

🧩 1. What “Scaling Out on Ray” Means

When your dataset or model inference workload grows beyond what a single machine can handle, you need **distributed execution** — running pieces of your job across multiple CPUs, GPUs, or even machines (a cluster).

Daft supports this *natively* using **Ray** (a distributed compute framework developed by UC Berkeley / Anyscale).

So instead of:

- manually writing parallel code,

- setting up job queues, or
- dealing with Dask / Spark configs,

you can just write your normal Daft code, and add **one line**:

```
import daft

daft.context.set_runner_ray()
```

That’s it — Daft automatically distributes your computation.

⚙️ 2. What Happens Internally

When you run that line, Daft switches its execution “engine” (runner):

Runner Type	What It Does
Default Runner (Local)	Runs everything on your local CPU in a single process
Ray Runner (set_runner_ray())	Uses Ray’s distributed execution engine to schedule your Daft computations across multiple cores, GPUs, or nodes

In Ray mode:

1. Daft splits your dataset into **partitions**
2. Each partition is sent as a **Ray task** to a remote worker
3. Ray handles **scheduling**, **fault tolerance**, and **load balancing**
4. Daft collects the results back automatically

You don't need to change your pipeline logic at all — Daft abstracts away the distributed complexity.

---

3. Example: Scaling Sentiment Analysis Across a Cluster

✅ **Step 1 — Write your normal Daft pipeline**

✅ **Step 2 — Enable distributed mode**

➡️ Daft will:

- Partition the df into chunks (e.g., 1000 rows each)
- Send each batch to a different Ray worker
- Run infer_batch() in parallel
- Aggregate the results

You're now using **multiple machines or GPUs**, without rewriting anything.

🚀 **4. "Read → Preprocess → Infer → Write" Pattern**

Daft is designed to **pipeline and parallelize these stages automatically**.

Here's a typical scalable pattern:

```
(
    daft.read_csv("s3://my-dataset/data.csv")
        .with_column("clean_text", clean_text_udf(df["raw_text"]))
        .with_column("sentiment", infer_udf(df["clean_text"]))
        .write_parquet("s3://my-output/predictions.parquet")
)
```

Daft executes this as:

1. **Read** in parallel (each partition reads a chunk of the file)
2. **Preprocess** concurrently on all workers
3. **Infer** using your model (batched & distributed)
4. **Write** results in parallel

All four steps run **asynchronously and pipelined** — meaning Daft can start inference on early batches before reading the entire dataset, maximizing throughput.

🌎 5. Provider-Agnostic Pipelines

Daft is designed to be **provider-agnostic** — meaning:

- You can run local models (like Hugging Face, scikit-learn, XGBoost)
- Or switch to **cloud APIs** (like OpenAI, Anthropic, Gemini, etc.)

...simply by changing one parameter.

For example:

```
def infer_batch(texts):
    if USE_OPENAI:
        return openai_infer(texts)
    else:
        return local_model_infer(texts)
```

Then just set:

```
USE_OPENAI = True  # or False
```

Your Daft pipeline doesn't change — only the model provider does.

💡 **Summary**

Concept	Explanation
daft.context.set_runner_ray()	Turns on distributed execution via Ray
Automatic partitioning	Daft splits your data and runs in parallel
No pipeline rewrites	Same Python code, just faster and scalable
End-to-end pipelining	Read → Preprocess → Infer → Write, all parallel
Provider-agnostic	Easily switch between local or API-based models

What is a Runner in Daft?

A **runner** in **Daft** defines **how and where** your computation actually executes.

Think of Daft like a modern version of Pandas + Spark + Ray combined — it builds a *logical plan* for your data operations, but the **runner** decides how that plan is **physically executed**.

Types of Runners in Daft

Runner	Description	When to Use
local	Runs everything on a single machine, single process	Small data or debugging

ray	Runs computations distributed across many CPU/GPU nodes via **Ray**	Large data, batch inference, or distributed model workloads
(Future) cloud runners	Will allow running on managed clusters like AWS, GCP, etc.	Production pipelines

Working with Different Modalities in Daft

Concept Overview

- **Modality** → A *type or form of data* with specific meaning and use.
- In AI, data isn't just numbers — it can be:
  - Text (e.g., reviews, articles)
  - Images (e.g., photos, logos)
  - Audio (e.g., speech, music)
  - PDFs (structured documents)
  - Embeddings (vectorized representations)
  - URLs (pointers to external data sources)

Why Modality Matters

- Different modalities need different handling — you can't process an image the same way as text.

- Each modality carries unique *semantic meaning* and *intended use*:
  - Text → analyze sentiment, extract keywords
  - Image → detect objects, extract embeddings
  - PDF → extract both text and images
  - URL → fetch and process remote content

Daft's Capability

Daft is designed as a **universal multimodal data engine**, meaning:

- It can **load, transform, and process** all these modalities in a single pipeline.
- You can handle diverse data sources **seamlessly** (no separate tools or workflows).
- It works efficiently both **locally** and **at scale** (with Ray or distributed backends).

Supported Modalities

Modality	Description / Usage
URLs & Files	Handle local or remote file paths; read directly from URLs.
Text	Natural language data; used for NLP, sentiment analysis, etc.
Images	Visual data for classification, detection, embedding extraction.
Videos	Frame-level analysis, scene detection, embeddings (coming soon).
JSON & Nested Data	Process semi-structured data with nested fields.

PDFs	Extract text, tables, and embedded images from documents.
Embeddings	Vectorized data used in similarity search or ML models.
Tensors / Sparse Tensors	Multi-dimensional numeric data for deep learning workloads.
Custom Modalities	Define your own (e.g., 3D meshes, sensor data) using Daft’s flexible API.

🔧 Key Benefits

- **Unified pipeline:** No need to switch tools between modalities.
- **Scalability:** Same code runs locally or across clusters (via Ray).
- **Extensibility:** You can define your *own modalities* and connectors.
- **Efficiency:** Daft handles I/O, parallelism, and batching automatically.

Example Use Cases

1. **Multimodal ML Pipelines**
   - Combine text + image data for product review analysis.
2. **Document Understanding**
   - Parse PDFs → extract text + tables → summarize using LLMs.
3. **Embedding Workflows**
   - Generate and store embeddings for search or retrieval.
4. **Data Lake Ingestion**
   - Read from URLs or cloud storage → preprocess → write to parquet.

Explanation (Step-by-Step)

1. daft.from_pydict

Creates a **Daft DataFrame** from a Python dictionary.
Each URL in the list becomes a row in the column urls.

2. @daft.func

This decorator turns a normal Python function into a **UDF (User Defined Function)** that Daft can distribute and run efficiently.

Here, detect_file_type():

- Receives a daft.File object (which can be a remote file, local file, or web resource).
- Opens it and reads the **first 12 bytes**.
- Detects file type based on **magic bytes** (signature patterns in file headers):
  - b"\xff\xd8\xff" → JPEG
  - b"\x89PNG\r\n\x1a\n" → PNG
  - b"GIF87a" / b"GIF89a" → GIF
  - b"<!html" → HTML page

3. file(df["urls"])

This is Daft's **file modality function**.
It converts each URL (or path) into a daft.File object that you can open, read, or inspect in distributed pipelines.

4. df.with_column("file_type", detect_file_type(...))

Adds a new column "file_type" to the DataFrame, which holds the detected file type for each URL.

5. df.collect() and df.show()

Materialize and display the computation results.

The first code: Default Hugging Face embeddings

```
import daft
from daft.functions.ai import embed_text

(
    daft.read_huggingface("togethercomputer/RedPajama-Data-1T")
    .with_column("embedding", embed_text(daft.col("text")))
    .show()
)
```

What's happening:

- You're calling embed_text() **without specifying a provider**.
- So Daft uses its **default embedding backend**, which is a **Hugging Face Transformers pipeline** under the hood.
- It automatically picks a **default text embedding model**, typically from the 🤗 Transformers ecosystem (like sentence-transformers/all-MiniLM-L6-v2 or equivalent lightweight model).
- It's simple, zero-config, and works out of the box.

✅ **Pros**

- Easiest to use — no setup.
- Great for quick tests or small demos.

⚠️ **Cons**

- Less control — you can't choose the embedding model.
- Might not be optimized for your specific use case (e.g., semantic search, multilingual data).
- Performance and embedding dimensionality depend on Daft's default choice.

🧠2️⃣The second code: Using Sentence Transformers explicitly

```
import daft
from daft.functions.ai import embed_text

provider = "sentence_transformers"
model = "BAAI/bge-base-en-v1.5"

(
    daft.read_huggingface("togethercomputer/RedPajama-Data-1T")
    .with_column("embedding", embed_text(daft.col("text"), provider=provider,
model=model))
    .show()
)
```

What's happening:

- Here, you explicitly specify:
  - provider="sentence_transformers" → tells Daft to use the **Sentence Transformers** library for embeddings.
  - model="BAAI/bge-base-en-v1.5" → specifies *which* model from Hugging Face to load.
- Sentence Transformers models are **specialized for semantic similarity**, **retrieval**, and **embedding-rich tasks**.

✅ Pros

- You have **full control** over which embedding model to use.
- Can use **high-quality open models** (like bge-base-en, all-MiniLM, mpnet, etc.).
- Embeddings are usually **better for semantic search, clustering, or recommendation** tasks.

⚠️ Cons

You must install the optional dependency:

```
pip install -U "daft[sentence-transformers]"
```

-
- Slightly heavier models → slower inference compared to default.

⚖️ TL;DR Comparison Table

Feature	Default (`embed_text()`)	With `provider="sentence_transformers"`
Backend	Hugging Face Transformers (auto)	Sentence Transformers
Control	Minimal	Full control
Model	Auto-selected default	You choose (`BAAI/bge-base-en-v1.5`, etc.)
Install Required	None	`pip install -U "daft[sentence-transformers]"`
Output Quality	Basic embeddings	High-quality semantic embeddings
Use Case	Quick testing	Production-grade NLP / semantic tasks

⚙️ Step-by-Step Explanation

Let’s unpack what’s happening line by line 👇

🧱1️⃣Daft + Hugging Face Integration

```
daft.read_huggingface("roneneldan/TinyStories")
```

- This line loads a dataset **directly from the Hugging Face Hub** into a **Daft DataFrame**.
- Example dataset: "roneneldan/TinyStories" — small English text stories (great for demos).
- Each row in the DataFrame has a "text" column containing a story.

This is equivalent to doing:

```
from datasets import load_dataset
dataset = load_dataset("roneneldan/TinyStories")
```

…but Daft reads it natively and can parallelize operations across large datasets.

🧠2️⃣spaCy Setup

```
nlp_model_name = "en_core_web_sm"
```

- Specifies which **spaCy language model** to use.
- en_core_web_sm → small English model trained for:
    - Sentence segmentation
    - Part-of-speech tagging
    - Named entity recognition, etc.

If you had el_core_news_sm, that would be the **Greek** model — so switching to en_core_web_sm ensures English NLP.

🔧3️⃣The Daft User Function (@daft.func)

```
@daft.func
def chunk_by_sentences(text: str) -> typing.Iterator[str]:
```

```
import spacy
nlp = spacy.load(nlp_model_name)
doc = nlp(text)
for sentence in doc.sents:
    yield sentence.text
```

Here's what's going on:

- **@daft.func**
  This decorator tells Daft:

  "I'm defining a function that can be distributed and run in parallel on each row or partition of the DataFrame."

- Inside it, we:
  - Load **spaCy** (the NLP engine).
  - Process the text with `nlp(text)` to get a **Doc object**.
  - Iterate through `doc.sents` → these are spaCy's automatically detected **sentences**.
  - `yield sentence.text` → returns one sentence at a time as an **iterator** (Daft collects them into a list-like column).

This means each row's text is **split into sentences**, and those are stored as a new column.

🧩4️⃣Applying the Function in Daft

```
.with_column("chunks", chunk_by_sentences(daft.col("text")))
```

- Adds a new column `"chunks"` to the Daft DataFrame.

- For each row's "text", Daft calls chunk_by_sentences(text).
- The output (iterator of sentences) becomes a **list of strings** in "chunks".

For example:

text	chunks
"Once upon a time. There was a cat."	["Once upon a time.", "There was a cat."]

🧮5️⃣ Limiting and Displaying

.limit(8).show()

- limit(8) → only shows the first 8 rows.
- show() → displays the DataFrame in a readable console table.

🚀 How Daft + spaCy Work Together

Layer	Role
spaCy	Handles **linguistic processing** — breaking text into sentences using its language model.
Daft	Manages **data distribution and parallelization** — runs your function efficiently on many rows at once.
Hugging Face dataset	Provides the **raw text data**.
@daft.func	Bridges both — turns your spaCy-based function into a distributed transformation callable by Daft.

Essentially:

> Daft acts like a distributed “controller”, sending your chunk_by_sentences() function to multiple workers — each worker loads spaCy, processes text, and returns sentence chunks — all merged back into one distributed DataFrame.

🧠 **Optional Optimization Tip**

If your dataset is large, **loading spaCy inside the function** can be slow.
You can improve performance by loading spacy.load() **outside** the function once:

Working with images
Step-by-Step Explanation

1️⃣Importing Libraries

```
import daft
from daft.functions.ai import embed_image
```

- daft: the main Daft library for distributed data processing and ML pipelines.
- embed_image: a **Daft AI function** that converts images into **vector embeddings** using pre-trained deep learning models (like CLIP, ViT, etc.).

2️⃣Loading a Public Dataset

```
daft.read_huggingface("xai-org/RealworldQA")
```

- This loads the **RealworldQA** dataset directly from **Hugging Face Datasets Hub**.
- Each record includes **images** and **associated metadata/questions**.
- Daft integrates with Hugging Face to stream and process data efficiently.

🧠 Think of it as:

> “Load this dataset from the cloud, I’ll process it directly without downloading manually.”

3️⃣Extract and Decode Image Bytes

.with_column("image", daft.col("image")["bytes"].image.decode())

- The dataset's images are stored as **byte blobs** (raw binary data).
- daft.col("image")["bytes"] selects that field.
- .image.decode() tells Daft:

  "Convert these raw bytes into actual image tensors that can be used for inference."

Result:
A new column "image" that holds decoded image data in memory, ready for AI models.

4️⃣Generate Image Embeddings
.with_column("embedding", embed_image(daft.col("image")))

- embed_image() computes a **numerical representation (embedding)** for each image.
- These embeddings capture visual similarity and semantic information.
- Under the hood, Daft uses a default model like **CLIP** (OpenAI's Contrastive Language-Image Pretraining) to generate these vectors.

Example:
If you have a cat and a tiger image — their embeddings will be **close together** in vector space.

5️⃣Display the Results
.show()

Prints a small preview of the processed DataFrame:

🚀 What You Achieved

✅ Loaded a **public image dataset** (no AWS/S3 needed).
✅ Decoded raw image bytes.
✅ Computed **deep embeddings** (image → vector).
✅ Displayed the processed results in a DataFrame.

🧠 Next Steps (if you want classification)

To **classify** images instead of just embedding them:

➡️ classify_image() will directly output a label (e.g., "dog", "car", "person") using a pre-trained vision model.

🧠 **Overall Idea**

You're building a **mini image classification pipeline** that:

1. Downloads images from the internet,

2. Decodes them into actual image objects,

3. Classifies them using a pretrained **Vision Transformer (ViT)** model from Hugging Face,

4. Displays the image next to its predicted class.

🧩 Step-by-step Breakdown

1. Import Dependencies

- **daft** → The distributed data processing engine (like a smart DataFrame, but optimized for multimodal AI data).
- **transformers.pipeline** → A high-level interface from Hugging Face to load pre-trained models easily.
- **PIL.Image** → For image decoding and manipulation.
- **io** → Lets you handle byte streams (needed to convert downloaded bytes into images).

---

2. Create a Daft DataFrame

```
df = daft.from_pydict({
   "image_url": [
     "https://upload.wikimedia.org/wikipedia/commons/9/9a/Pug_600.jpg",

"https://upload.wikimedia.org/wikipedia/commons/4/47/PNG_transparency_demonstr
ation_1.png",
     "https://upload.wikimedia.org/wikipedia/commons/a/a3/81_INF_DIV_SSI.jpg",
      "https://upload.wikimedia.org/wikipedia/commons/3/3a/Cat03.jpg",
   ]
})
```

- You're creating a Daft DataFrame (like pandas, but distributed and multimodal).

- Each row contains an image URL that you'll later process.

3. Download the Images

```
df = df.with_column("image_bytes", df["image_url"].url.download())
```

- Daft automatically fetches each image using its URL.
- It stores the **raw bytes** of each image in a new column called "image_bytes".
- This step is **vectorized and parallelized** — Daft downloads multiple images simultaneously if distributed execution is enabled (e.g., via Ray).

4. Decode Images for Display

```
df = df.with_column("image", df["image_bytes"].image.decode())
```

- Converts the binary bytes (raw data) into an **actual image object** that can be rendered visually (like in notebooks or Daft's .show() output).
- This allows you to *see* the image in the output table later.

5. Load the Image Classification Model

```
image_classifier = pipeline("image-classification",
model="google/vit-base-patch16-224")
```

- Loads a pretrained **Vision Transformer (ViT)** model from Google trained on ImageNet.
- This model predicts what's inside an image (dog, cat, car, etc.).
- The pipeline automatically handles preprocessing, resizing, normalization, etc.

6. Define a Daft UDF (User Defined Function)

- The @daft.func decorator turns this into a **parallelizable Daft function**.
- Input: raw image bytes.
- Inside the function:

    1. Bytes → BytesIO → Open as an image using PIL.
    2. Pass the image to the Hugging Face pipeline.
    3. Get the **top-1 prediction** (result[0]).
    4. Return a formatted string like "Pug (0.98)".

7. Apply the Classifier to Each Row

```
df = df.with_column("classification", classify_image(df["image_bytes"]))
```

- Daft applies your function to every image row in the DataFrame.
- It runs in parallel if distributed mode (like Ray) is enabled.
- Adds a new column "classification" with each image's label and confidence score.

---

8. Collect Results & Display

- **collect()**: Executes the entire Daft pipeline (it's lazy by default).
- **select(...).show()**: Displays only the image and its classification in a neat table.

⚙️ Behind the Scenes

- Daft handles **parallel execution** and **lazy computation**.
- Each operation (with_column, url.download, etc.) adds a new stage to the pipeline.

- Only when you call .collect() does it execute the graph efficiently (like Spark or Dask).
- You can scale the same script to run on **Ray clusters** with one line:

Daft Parralization
Daft achieves parallelism by combining **a distributed query engine**, **lazy evaluation**, and **multi-backend execution** — similar in spirit to Spark or Polars, but with modern design and tight integration with ML + GenAI tooling.

When you write Daft code like:

```
df = daft.read_huggingface("roneneldan/TinyStories")
df = df.with_column("chunks", chunk_by_sentences(df["text"]))
df.show()
```

Daft doesn't execute the operations immediately.

Instead, it **builds a query plan** — a DAG (Directed Acyclic Graph) that represents:

- Read operations
- Transformations
- Function applications (@daft.func)
- Joins, filters, aggregations, etc.

When you finally call an **action** (like .show(), .collect(), .to_pandas()),
Daft **optimizes and executes the DAG** across workers.

This deferred approach allows Daft to:

- Reorder operations for efficiency
- Parallelize independent tasks automatically
- Combine computation stages (fusion)

Partitioned Data Model

Internally, Daft splits every DataFrame into **partitions** (also called *batches* or *fragments*).

Example:

```
df = daft.read_parquet("huge_dataset.parquet")
```

Daft automatically partitions this dataset — e.g.:

- Partition 1 → rows 0–9999

- Partition 2 → rows 10,000–19,999
- … and so on

Each partition can be processed **independently** by a worker thread or process.

The @daft.func → Parallel Execution Layer

When you mark something with @daft.func, Daft wraps it in a *serializable function object* that:

- Knows how to handle its inputs/outputs
- Can be sent to remote executors
- Can run on multiple partitions simultaneously

So for 1M rows of text:

- Daft splits them into, say, 100 partitions
- Each partition is sent to a worker (thread/process/node)
- Your function (chunk_by_sentences, etc.) runs independently on each partition
- Daft merges all results back into a single DataFrame

Parallelization happens *per partition*, not per row — so overhead stays low.

What @daft.func Actually Does

When you write:

```
@daft.func

def chunk_by_sentences(text: str):

    ...
```

you are **not** just defining a normal Python function.
You are defining a **Daft user-defined function (UDF)** — meaning Daft can take that function and **run it in parallel** across many rows, batches, or even distributed nodes

Normally, a Python function runs **sequentially** — one input at a time, on one machine.

But Daft is a **distributed DataFrame engine**, designed for **parallel computation**.
So when you add @daft.func, you're telling Daft:

What Happens Internally

Let’s break it down behind the scenes:

1. **Decorator Registration**
   When Daft sees @daft.func, it wraps your function and records metadata about:
   - Input types
   - Output types
   - Serialization (how to send code and data to workers)
   - Safe execution in parallel environments (like Ray or Polars backend)

Distributed Execution
When you later do something like:

```
df = df.with_column("chunks", chunk_by_sentences(daft.col("text")))
```

2. Daft takes each partition (or chunk) of your DataFrame and **runs your function on multiple workers** — each processing a different subset of rows.
3. **Automatic Merging**
   The results from all workers (each returning lists, numbers, etc.) are **merged back** into the final DataFrame automatically.

What is a Struct Column in Daft?

A **struct column** (or **struct type**) is a **column that contains structured data** — basically a small “record” (like a JSON object or Python dict) stored **inside a single cell** of a table. **In Daft terms**

Daft supports **complex data types**, and StructType is one of them.

PILOT PROJECT - CODE ANALYZER

Project Structure and Workflow

1. Initial Embedding Generation (main.py)

The process starts with main.py which:

1. **Data Loading with Daft**

   - Creates a Daft DataFrame to efficiently manage Python files
   - Uses efficient memory management through Daft's DataFrame structure

```
df = daft.from_pydict({

    "filename": [f["filename"] for f in code_files],

    "code": [f["code"] for f in code_files]

})
```

2. **Embedding Generation using LM Studio**
   - Connects to LM Studio's API using OpenAI-compatible interface
   - Uses the text-embedding-nomic-embed-text-v1.5 model for embeddings
   - Processes each code file to generate numerical vector representations
   - Saves embeddings to embeddings.json

2. Enhanced Analysis (understand_enhanced.py)

After embeddings are generated, understand_enhanced.py performs advanced analysis:

1. **Loading and Setup**
   2. **client = OpenAI(**
   3. **base_url=os.getenv("LM_STUDIO_BASE_URL", "http://localhost:1234/v1"),**
   4. **api_key=os.getenv("LM_STUDIO_API_KEY", "lm-studio")**
   5. **)**
   6. **provider = load_provider("lm_studio")**
7.
8. **Code Similarity Analysis**
   - Uses scipy's cosine distance to compare embeddings
   - Identifies code blocks that might be redundant or related

```
def find_similar_code(embeddings, names, threshold=0.8):

    # Uses cosine similarity to find similar code blocks

    # Returns pairs of files that are similar above the threshold
```

9. **In-depth Code Analysis**

  - Leverages LM Studio's CodeLlama model for deep code understanding
  - Analyzes:
    - Code functionality
    - Key functions and purposes
    - Coding patterns
    - Potential improvements

```
def analyze_code(client, code):

    # Uses LM Studio's codellama-7b-instruct model

    # Provides detailed analysis of each code file
```

10. **Efficient Data Processing**
    - Uses Daft for efficient handling of high-dimensional embedding vectors

```
df = daft.from_pydict({

    "filename": filenames,

    "code": codes,

    "embedding": embeddings

})
```

How Daft and LM Studio Work Together

Daft's Role:

1. **Data Management**
   - Efficient DataFrame operations
   - Type-safe data handling
   - Memory-efficient processing of large code files
   - Vectorized operations for better performance
2. **Integration Features**
   - Native LM Studio integration through providers
   - Efficient handling of high-dimensional embedding vectors
   - Built-in error handling and progress tracking

LM Studio's Role:

1. **Embedding Generation**
    - Provides text-embedding-nomic-embed-text-v1.5 model
    - Converts code into meaningful numerical vectors
    - Enables similarity comparisons
2. **Code Analysis**
    - Uses codellama-7b-instruct model
    - Provides detailed code understanding
    - Offers suggestions and improvements

Project Outputs

1. **embeddings.json**
    - Contains raw embeddings for each code file
    - Includes:
        - Filename
        - Original code
        - Embedding vectors
2. **enhanced_analysis.json**
    - Contains comprehensive analysis including:
        - File details
        - Similarity comparisons
        - Code analysis results

Key Benefits of This Approach

1. **Efficiency**
    - Daft's optimized DataFrame operations
    - Parallel processing capabilities
    - Memory-efficient handling of large datasets
2. **Accuracy**
    - High-quality embeddings from LM Studio
    - Detailed code analysis from CodeLlama
    - Reliable similarity detection
3. **Scalability**
    - Can handle large codebases
    - Efficient processing of multiple files
    - Memory-efficient vector operations
4. **Integration**
    - Seamless interaction between Daft and LM Studio
    - Easy-to-use API interfaces
    - Consistent error handling

This setup provides a powerful system for code analysis, combining Daft's efficient data processing with LM Studio's advanced language models, creating a comprehensive code understanding and analysis pipeline.

The entire process is automated and can be run with two simple commands:

```
# First generate embeddings

$env:LM_STUDIO_BASE_URL="http://localhost:1234/v1";
$env:LM_STUDIO_API_KEY="lm-studio"; python main.py

# Then run enhanced analysis

$env:LM_STUDIO_BASE_URL="http://localhost:1234/v1";
$env:LM_STUDIO_API_KEY="lm-studio"; python understand_enhanced.py
```