EL HARAKAOUI Chaymae --- 11-08-2024

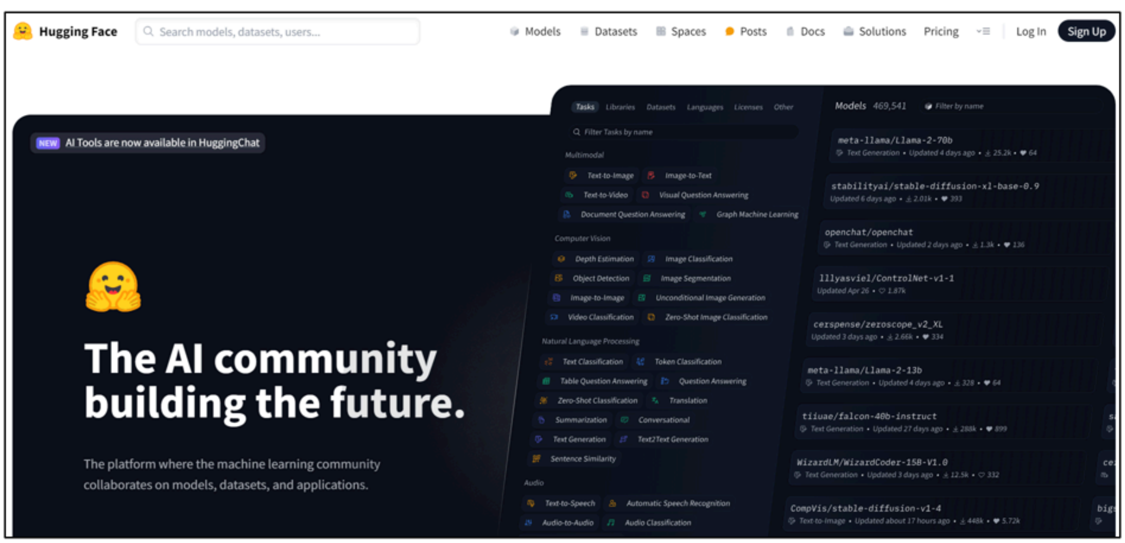
Text to Image Generation

Hugging Face

What is Hugging Face ?

Hugging Face is a **company** - **platform** and an **open-source community** that specializes in **machine learning** - particularly in **natural language processing** - NLP.

It is widely known for its development of **Transformers** - an open-source library that provides a large collection of pre-trained models for various NLP tasks - including text classification - translation - summarization - question answering ....



Key Aspects of Hugging Face

Transformers Library

This is the most popular library provided by HF. It supports models like **BERT** - **GPT** - **T5** and many others - allowing developers to easily implement state-of-the-art NLP models in their projects.

Model Hub

It hosts a model hub where users can **find** and **share** many **pre-trained models**. It includes thousands of models contributed by the community - making it easier for developers to fine-tune or use pre-trained models for specific tasks.

Community and Open Source

It has a strong focus on community and open-source development - encouraging collaboration and sharing of models - datasets - research.

Datasets Library

It also offers a Datasets library - which provides access to a wide range of datasets for NLP tasks. This helps users easily find and utilize datasets for training and evaluating their models.

Why are we using Hugging Face Here ?

Loading the pre-trained model **StableDiffusionPipeline** from Hugging Face's model hub.

What is authentication token ?

An **Authentication token** typically a **string** that serves as a **secure key** allowing you to **access** certain resources - services - data that require authentication.

Specifically - in this case - it is used to authenticate our access to models and datasets hosted on Hugging Face's model hub.

Hugging Face hosts a variety of pre-trained models - some of which are **restricted** or **require** a Hugging Face account to access. These models are **not always publicly available** due to licensing - usage restrictions or because they are under active development.

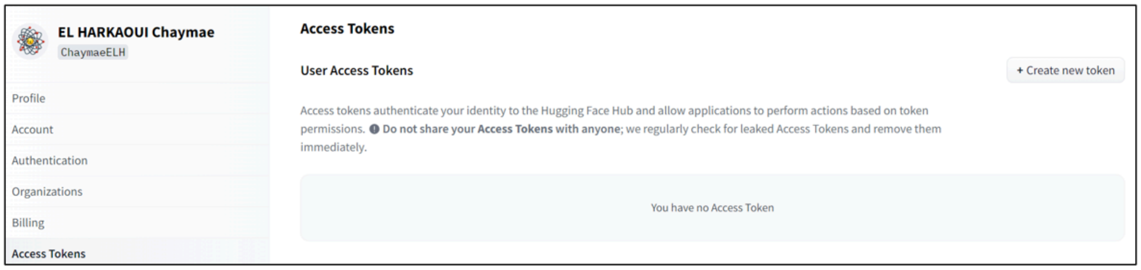
When we **create** an **account** on Hugging Face - we can **generate** an **API key** also known as an **authentication token**.
This key is unique to our account and can be **used in scripts and applications** to **authenticate our access**.

When we use the **authentication token** in our code - it tells the Hugging Face **API** that the **request** is coming from an **authenticated user**.
The **API** then **checks** if **the user associated with that token** has the **necessary permissions** to **access the requested resource**.

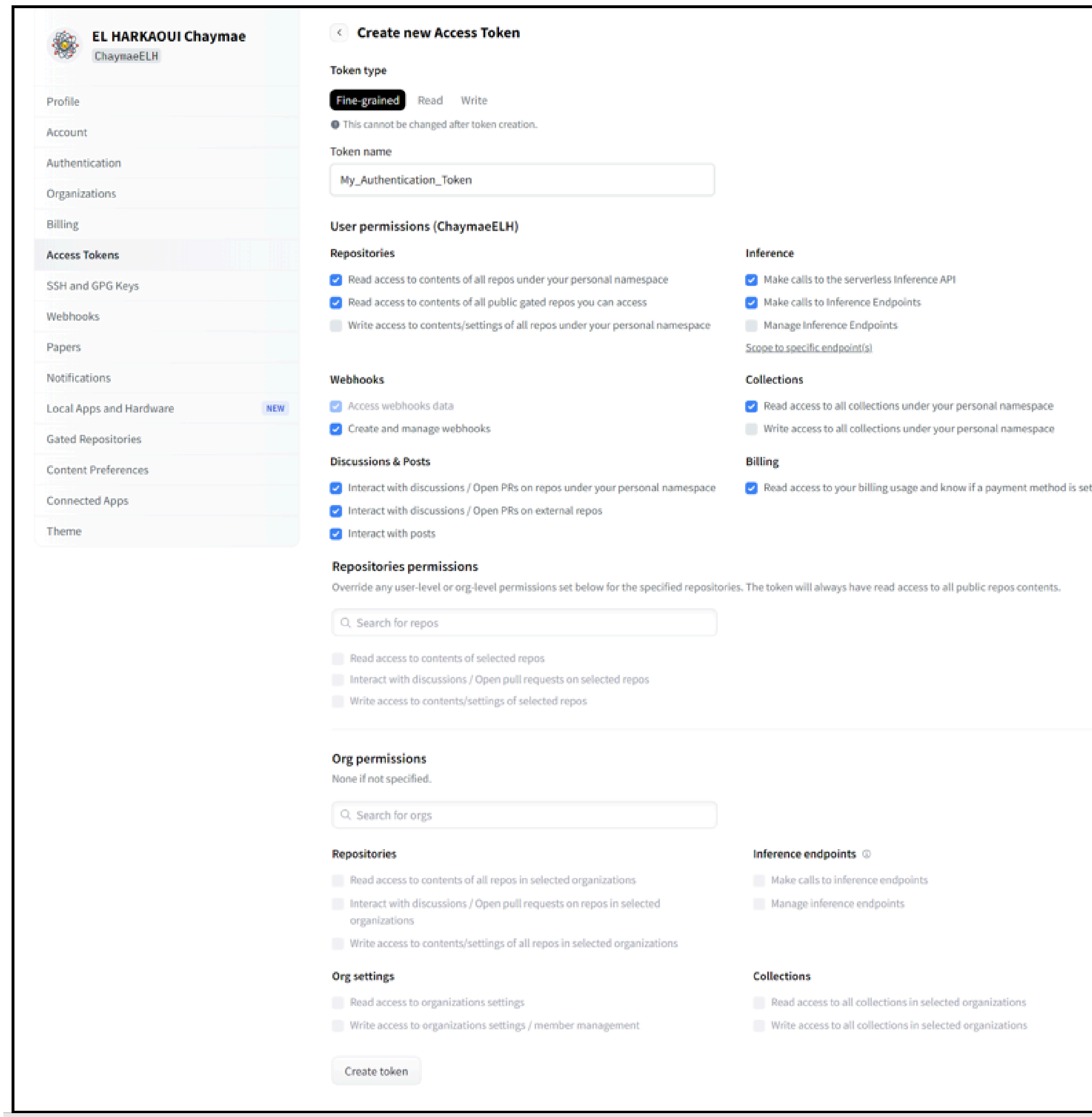
How to get an authentication token ?

1 - Create a Hugging Face Account.

2 - Go to your account settings and look for the section labeled **Access Tokens**.

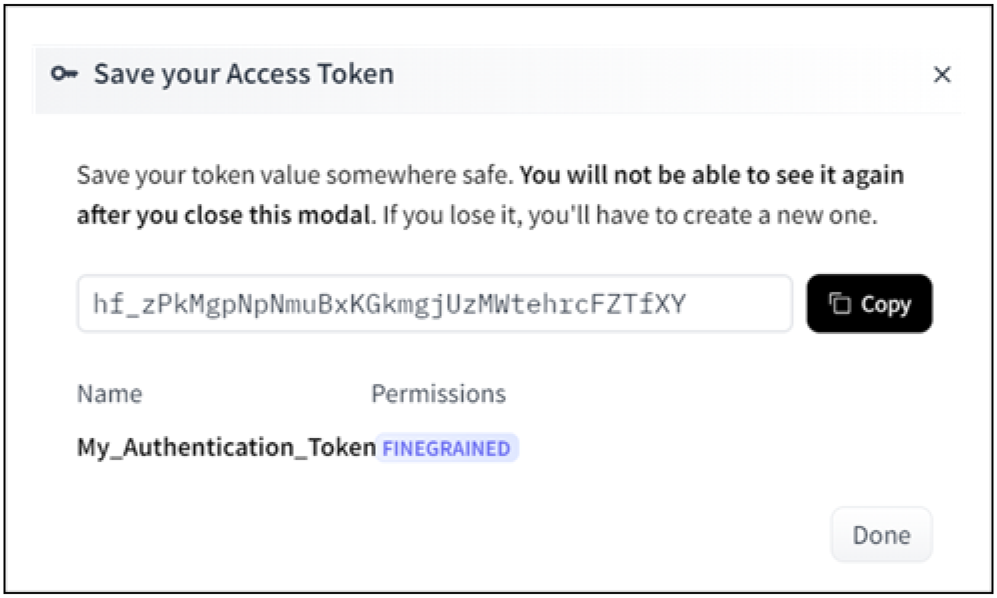


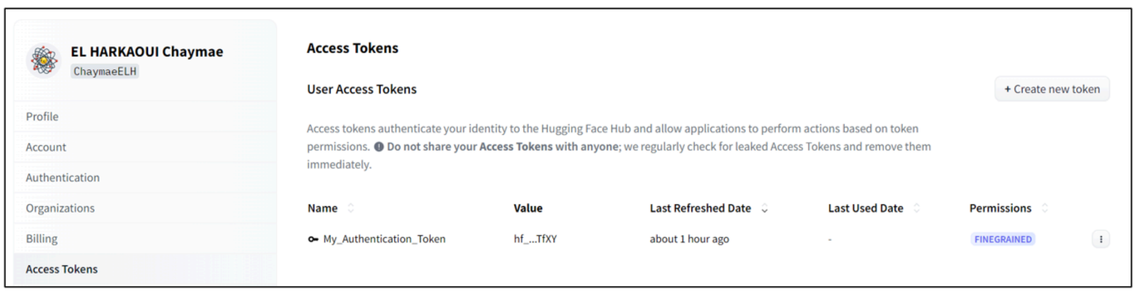
3 - Click on the **create new token** button to start the process of generating a new token.



4 - Once the token is generated -it will be displayed on the screen. Copy this token immediately - as it may not be displayed again for security reasons.

my generated token is : **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**





How to use the Generated authentication token in our code ?

In [ ]:
```
# my_auth_token = "my_huggingface_api_token_here"
# pipe = StableDiffusionPipeline.from_pretrained( modelid , revision="fp16", torch_
```

Necessary Librairies

In [ ]:
```
import tkinter as tk
import customtkinter as ctk

from PIL import ImageTk


import torch
from torch import autocast
```

```python
from diffusers import StableDiffusionPipeline
```

PyTorch

PyTorch is an open-source **deep learning framework** primarily developed by **Facebook's AI Research lab**. It provides a flexible **platform** for building and training **neural networks**.

Key Features

Dynamic Computational Graph

PyTorch uses a **dynamic computation graph** also known as **define-by-run** - which means that the **graph** is **built** on-the-fly as **operations** are executed.
This makes it easier to modify and experiment with models.

Automatic Differentiation

PyTorch's **autograd** module automatically **computes gradients** - which are essential for training neural networks using **optimization algorithms** like **gradient descent**.

Tensor Computation

PyTorch supports powerful tensor computation - similar to NumPy - but with **GPU** acceleration. This makes it highly efficient for large-scale computations.

Rich Ecosystem

PyTorch has a growing ecosystem - including libraries for vision - **TorchVision** - natural language processing - **TorchText** - reinforcement learning - **TorchRL** and more.

Interoperability

PyTorch **integrates** well with Python and **other** scientific computing **libraries **- making it easy to incorporate into existing projects.

Community and Support

PyTorch has a large and active **community** - with extensive **documentation** - tutorials - forums for support.

Encountered Error When Installing PyTorch

OSError: [WinError 126] The specified module could not be found. Error loading "c:\Users\MTechno\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\lib\fbgemm.dll" or one of its dependencies.

Solution : Install Visual Studio Installer - Visual Studio C ++ Compiler

What is Visual Studio Installer ?

Visual Studio Installer is a **tool** provided by **Microsoft** that allows users to **install** - **update** and **manage** various **components** of **Visual Studio** - IDE. It provides a **streamlined way** to **select** the **necessary workloads** - **tools** - **libraries** based on the type of development we plan to do.

The Visual Studio Installer is a **background** utility tool.

How Visual Studio Installer Manages Dependencies ?

Selection of Workloads and Components

When we **run** the **Visual Studio Installer** - we are presented with various **workloads** - ex: Desktop Development with C++ - Web Development .... and **individual components**. **Selecting** a **workload** or **component** that includes **tools** or **libraries** like **compilers** - **linkers** - **runtime libraries** will **prompt** the **installer** to **download** and **install** those **necessary components** if **they are not already present** on our system.

Automatic Installation of Dependencies

The **installer** ensures that the **necessary dependencies** for the **selected** workloads and components are **installed**. This includes :
Compilers - **Linkers** - **Build Tools** - **Libraries** - **SDKs** ......
For example - if we choose to install the **C++ development workload** - the **installer** will also **install** related **tools** and **libraries** needed to **build** and **run** C++ **applications**.

Updating and Patching

The Visual Studio **Installer** can also **update** existing **installations** of **Visual Studio** and its **components**.

It **checks** for **updates** and **patches** to ensure that your **development environment** is **up-to-date**.

Configuration

During the installation - the installer **configures** environment variables - paths - other settings **required** for the **tools** to **function correctly**.
This includes updating the system PATH to include directories where the necessary binaries and libraries are located.

Examples of What the Installer Manages

Compilers and Linkers

Ensures that the required **compiler** - ex : MSVC - and **linker** tools are installed and available.

Runtime Libraries

Installs necessary runtime libraries like **the Visual C++ Redistributable packages**.

SDKs and Libraries

Installs **Software Development Kits** - SDKs - and other **libraries** needed for specific development scenarios - ex: .NET SDKs for .NET development - DirectX SDKs for game development ...

Build Tools

Includes **tools** required for building and debugging code such as **build systems** and **debugging tools**.

Why Did the Error Disappear After Installing the VS Code Installer - C++ Compiler ?

The error we encountered - **OSError: [WinError 126] The specified module could not be found** - was due to **missing dependencies** that are **required** by the **fbgemm.dll** file in PyTorch.

VS Code itself is an Integrated Development Environment - **IDE** - for **editing** and **running** code - but it **doesn't come** with the **underlying system libraries** and **runtime dependencies** required by certain **external packages** such as PyTorch - which **rely on native C++ libraries**.

The **fbgemm.dll** file in PyTorch has **dependencie**s on certain DLLs - **Dynamic Link Libraries** - provided by the **Visual C++ Redistributable packages**. Without these dependencies - Windows was **unable to load fbgemm.dll** resulting in the WinError 126 we encountered.

In the case of the error encountered with PyTorch related to **missing DLLs like fbgemm.dll** - the **Visual Studio Installer** likely **installed** the Microsoft Visual **C++ compiler** and associated components that PyTorch depends on.

When the **VS Code C++ compiler** is installed - it also installed the necessary **Visual C++ Redistributable packages** as part of the setup.
These **redistributables** provided the **missing DLLs** that **fbgemm.dll** and potentially other parts of PyTorch were **depending on**.

With the required **DLLs** now **present** on our system - Windows could successfully **load fbgemm.dll** when we imported PyTorch and the error was resolved.

What Are Runtime Librairies ?

Runtime libraries are **collections** of **pre-compiled routines** as functions - classes - data structures - that a **program** can call **during execution** - even though they are **not part of the program's original code**. These libraries provide **standard functionality** that is used by many programs and ensure that programs can be executed correctly on a system.

Example Scenario

Suppose we write a **C++ program** that uses the **printf() function** to print to the console. Instead of writing the printf() function ourself - we rely on the **implementation** of printf() **provided** in the **C runtime library** - CRT. When we compile the **program** - the **C runtime library** is **linked** with our **program**. If we dynamically link the CRT - our program will **call** printf() from the **MSVCP140.dll** library at **runtime**. If the system where our program is executed does not have this runtime library - our program might fail to run.

What Are Visual C++ Redistributable packages ?

Visual C++ Redistributable Packages are **runtime libraries** that are **required** to **run applications** built using **Microsoft Visual C++**. These packages contain the **necessary components** needed to execute programs that have been developed with **Microsoft’s Visual C++ tools**.

What Are Dependencies ?

Dependencies refer to **external components** - **libraries** - **modules** that a **software program** or **system** requires in order to **function correctly**. These dependencies provide necessary **functionalities** or **resources** that the main program itself does not include.

Dependencies are **external** in the sense that they are **not part** of our project's **core codebase** - we did not write them - but come from **third-party sources**. Where they are stored depends on how we manage them.

Where the Dependencies could be located ?

Stored in the Project Folder

In some cases - these external **dependencies** are installed **locally in our project directory**. Although they are external to our code - meaning we didn't write them - they are **downloaded** and **stored** within our **project's folder structure** - such as in a **venv/ - Python - folder**. This makes them **external** but **local**.

When we use a **package manager** like npm - pip - yarn - these tools will **download** the required **external libraries** and **store** them **within the project folder** or a designated location like a virtual environment. This allows the project to be self-contained and others who work on the project can simply **install** these **dependencies** by running the appropriate commands - e.g. : npm install or pip install -r requirements.txt.

Truly External - Not Stored in the Project Folder

In other cases - the external dependencies may not physically reside within your project folder. Instead they may be **installed globally on your system** or **exist in some centralized location** or **referenced from another location as if installed or hosted on the web**. In this scenario - the project folder contains no actual code for these external libraries - it only **references them** and the system **looks for them** in **predefined locations** - ex: system paths or global package directories or URLs .....

Types of Dependencies

+ Library Dependencies

Static Libraries

These are **compiled** directly into the application at **build time**. The **code** from the **library** becomes **part** of the **final executable**.

During the **compilation process** - the code from static libraries is **copied** directly into the **executable** - all the code needed from the static library is **included** in our **final executable file**.

SL are included in the executable at **compile time** - **no need** for the **library** file at **runtime**.

Dynamic Libraries

These are **separate files** - ex: **.dll in Windows** - **.so in Linux** - that are **loaded** into memory at **runtime**. The program will fail to run if the required dynamic library is missing or incompatible.

DL are **loaded** into **memory** when the application **runs**. The **executable** contains **references** to the dynamic libraries it needs - but the actual library **code** is **loaded** from the file at **runtime**.

At **runtime** - the **dynamic library** must be **available** in the system's library path or in a **location** where the application **expects to find it**. If the library is not found - the application might fail to start or may not work correctly.

DL are loaded at **runtime** - the **library file** must be **accessible** when the application is **running**.

+ Framework Dependencies

Some applications depend on larger frameworks like .NET - Java - Angular. These frameworks provide the foundation on which the application is built.

+ System Dependencies

Programs may depend on specific system **components** like **drivers** or core OS **services**.

+ Hardware Dependencies

Some software requires specific **hardware components** to function such as **GPUs** for AI/ML workloads or specialized **sensors** for IoT devices.

autocast

autocast is a **feature** from the **torch.cuda.amp** - **Automatic Mixed Precision module** in PyTorch - which is used to **automatically switch between different data types** - specifically between float32 and float16 - to optimize performance on modern **GPUs**.

Key Features

Mixed Precision Training

It enables **mixed precision training** - where parts of the model are run in lower precision - float16 - while others remain in higher precision - float32. This can greatly improve performance without significantly affecting model accuracy.

GPU Efficiency

Mixed precision helps leverage Tensor Cores available on NVIDIA GPUs - which are optimized for operations on half-precision floating-point numbers - making computations faster and more efficient.

Automatic Casting

autocast **automatically** selects the **appropriate precision** for different **operations**. For example - matrix multiplications might run in float16 -while reductions - which could be more sensitive to precision - may run in float32.

diffusers

diffusers is an **open-source library** developed by **Hugging Face** that provides a unified interface for working with **diffusion models**. **Diffusion models** are a **class of generative models** that **create data** like images - audio - text --- by progressively **refining random noise into meaningful content through a diffusion process**. The diffusers library aims to make it easy to **implement** - **train** - **use** these models - particularly in **text-to-image generation** tasks like **Stable Diffusion**.

Stable Diffusion

Stable Diffusion is a type of **generative model** primarily used for creating **images** based on **text prompts**. It's part of a broader category of models known as **diffusion models**.

Diffusion models are inspired by the process of **diffusion** in **physics** - where **particles spread out over time**. In the context of **generative modeling** - they use a similar concept but in **reverse**. The idea is to gradually **convert noise** into a **structured image** through a series of steps.

Training Phase

During **training** - the model learns to **reverse** a **diffusion process**. This involves two main stages :

+ Forward Diffusion Process

The model gradually adds **noise to images in the dataset** over many steps until the images are completely noisy and indistinguishable from **random noise**. This process is designed to teach the model how noise corrupts images.

+ Reverse Diffusion Process

The model then learns to reverse this process. It is trained to **denoise the noisy images** step-by-step - eventually **reconstructing the original images from noisy versions**. The model learns this by comparing its denoised outputs with the original images and adjusting its parameters to minimize the difference.

Generative Phase

Once trained - the Stable Diffusion model can generate new images from scratch based on text prompts :

+ Starting with Noise

To create a new image - the model **begins** with a **random noise pattern**.

+ Guided Denoising

It then iteratively applies a **series of transformations** to this **noise** - **guided by the text prompt**. These transformations involve denoising the image progressively while incorporating the details from the text description.

+ Final Image

After several iterations - the **noise** is **transformed** into a **coherent image** that **aligns** with the **text prompt**.

Text-to-Image Synthesis

The Stable Diffusion model is particularly notable for its ability to generate images based on text prompts. This is achieved through a **combination** of :

+ Text Encoder

A **neural networ**k that processes and **converts** the **text prompt** into a **feature vector**.

+ Conditioning Mechanism

This **feature vector** is used to **condition** the **denoising process** - guiding the model to generate an image that **matches** the description.

StableDiffusionPipeline

The **StableDiffusionPipeline** is a **pipeline class** provided by the **diffusers library** that organizes the **components** of the **Stable Diffusion model** for easier use.

This class wraps the entire workflow for **image generation** from **text prompts.** It abstracts away the complexity and provides an easy-to-use **interface** for interacting with the model.

It contains various components such as the pre-trained text **encoder** - CLIP - UNet used for **denoising** - **scheduler** for controlling the **diffusion process** - VAE **Variational Autoencoder** used for encoding and decoding **latent images**.

Inside the **pipeline** - the actual **Stable Diffusion model** is the key **generative model** responsible for creating the images. The pipeline integrates this model with other **supporting pieces**.

When we say that **StableDiffusionPipeline** is a **pipeline** - we mean that it is a **structured workflow** that **integrates several components** to achieve a **specific task** - in this case - **generating images from text prompts**.

what is a PipeLine ?

In machine learning and deep learning - a **pipeline** refers to a **sequence** of data processing **steps** or **stages** that are **applied** in a **specific order** to produce a desired **outcome**. Each stage in the pipeline typically performs a **specific task** - and the **output** of one stage becomes the **input** for the next stage.

In a machine learning pipeline - the typical steps might include :

1 - Data Preprocessing : Cleaning and transforming raw data - e.g. handling missing values - scaling ...
2 - Feature Engineering : Creating new features or selecting important features.
3 - Model Training : Fitting a machine learning model to the data.
4 - Evaluation : Assessing the performance of the model.
5 - Prediction : Making predictions based on new data.

In the Context of StableDiffusionPipeline :

The **StableDiffusionPipeline** is a **specific pipeline** designed for **text-to-image generation**. It orchestrates the following components in sequence :

Text Encoder : Converts the **input text** - prompt - into a **latent representation**.
UNet Model : Uses this **latent representation** to progressively refine an image by **reducing noise** in a **diffusion process**.
Scheduler : Controls how the **denoising process** happens over time.
VAE Decoder : Converts the final **latent image** back into **pixel space** to produce a visible image.

transformers

The Hugging Face transformers library is an open-source Python library that provides **access** to a wide range of pretrained **Transformer models** for various **NLP** tasks.

Key Features

Pretrained Models

Hugging Face offers thousands of **pretrained models** for various tasks such as **text classification** - **question answering** - **summarization** - **translation** - **text generation** and more. These models can be used **directly** or **fine-tuned** on custom datasets.

Easy-to-Use API

The library provides a simple **API** to **load** - **use** - **fine-tune** models with just a few lines of code. It abstracts many of the complexities of dealing with machine learning models.

Task-Specific Pipelines

The library includes high-level **pipelines** that allow users to perform common **NLP tasks** out of the box without needing to understand the inner workings of the models.

Fine-Tuning

Hugging Face makes it straightforward to **fine-tune pretrained models** on **custom datasets** - enabling users to tailor models to their **specific** needs.

tokenizer

A **tokenizer** is a crucial component in NLP models - particularly in models based on the **Transformer architecture**. The tokenizer **converts** raw text like sentences or paragraphs into a **structured format** that a model can understand and process.

Transformers and most **NLP models** work with **numerical representations** of **text**. Since models **cannot** directly **interpret** plain **text** - a **tokenizer** transforms words or characters into **tokens** - which are **numerical indices or vectors**. These tokens are then fed into the model.

Create the graphical interface

In [ ]:
```
# the root
```

```
root = tk.Tk()
root.geometry("600x700")
root.title("Text To Image Generation")

# text prompt

prompt_text = ctk.CTkEntry(master=root, height=40, width=580, font=("Arial", 20), t
prompt_text.place(x=10, y=10)


# a Label that contains the image

label_image = ctk.CTkLabel(master=root, height=570, width=580,fg_color="navy",text=
label_image.place(x=10, y=118)
```

Load the Stable Diffusion pipeline locally

In [ ]:

```
# Load the Stable Diffusion pipeline locally

model_id = "CompVis/stable-diffusion-v1-4"
pipe = StableDiffusionPipeline.from_pretrained(model_id, revision="fp16", torch_dty

# Save the model to the project directory
pipe.save_pretrained("./stable_diffusion_v1_4")
```

```
c:\Users\MTechno\AppData\Local\Programs\Python\Python311\Lib\site-packages\diffusers
\pipelines\pipeline_loading_utils.py:219: FutureWarning: You are loading the variant
fp16 from CompVis/stable-diffusion-v1-4 via `revision='fp16'` even though you can lo
ad it via `variant=`fp16`. Loading model variants via `revision='fp16'` is deprecate
d and will be removed in diffusers v1. Please use `variant='fp16'` instead.
  warnings.warn(
safety_checker\model.safetensors not found
Keyword arguments {'use_auth_token': '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'} are no
t expected by StableDiffusionPipeline and will be ignored.
Loading pipeline components...:   0%|          | 0/7 [00:00<?, ?it/s]
```

```
An error occurred while trying to fetch C:\Users\MTechno\.cache\huggingface\hub\mode
ls--CompVis--stable-diffusion-v1-4\snapshots\2880f2ca379f41b0226444936bb7a6766a22758
7\unet: Error no file named diffusion_pytorch_model.safetensors found in directory
C:\Users\MTechno\.cache\huggingface\hub\models--CompVis--stable-diffusion-v1-4\snaps
hots\2880f2ca379f41b0226444936bb7a6766a227587\unet.
Defaulting to unsafe serialization. Pass `allow_pickle=False` to raise an error inst
ead.
An error occurred while trying to fetch C:\Users\MTechno\.cache\huggingface\hub\mode
ls--CompVis--stable-diffusion-v1-4\snapshots\2880f2ca379f41b0226444936bb7a6766a22758
7\vae: Error no file named diffusion_pytorch_model.safetensors found in directory
C:\Users\MTechno\.cache\huggingface\hub\models--CompVis--stable-diffusion-v1-4\snaps
hots\2880f2ca379f41b0226444936bb7a6766a227587\vae.
Defaulting to unsafe serialization. Pass `allow_pickle=False` to raise an error inst
ead.
c:\Users\MTechno\AppData\Local\Programs\Python\Python311\Lib\site-packages\transform
ers\tokenization_utils_base.py:1601: FutureWarning: `clean_up_tokenization_spaces` w
as not set. It will be set to `True` by default. This behavior will be depracted in
transformers v4.45, and will be then set to `False` by default. For more details che
ck this issue: https://github.com/huggingface/transformers/issues/31884
  warnings.warn(
```

from pretrained method

the **from_pretrained** is a **method** of the class **StableDiffusionPipeline** from the librairy **diffusers** - it is used to **load** a **pre-trained diffusion model** like **Stable Diffusion** and its associated **components** from the **Hugging Face Model Hub** or a **local directory**.

What from_pretrained Does ?

Download Model Weights and Components

If the model is not already **stored locally** - from_pretrained **downloads** the pre-trained model **weights** - **configuration files** and any other necessary components such as **tokenizers** - **schedulers** or **safety filters** from the **Hugging Face Model Hub**.

Initialize the Pipeline

It **initializes** the **pipeline** with all the components needed to **run** the model. For StableDiffusionPipeline - this typically includes the **U-Net model** - the **variational autoencoder** - VAE - the **text encoder** and the **scheduler**. These components are essential for generating images from text prompts using the Stable Diffusion model.

Configure the Model

The method allows us to pass various options - such as :

- **revision** : To specify a particular version of the model.
- **torch_dtype** : To set the data type for tensors - ex: torch.float16 for half-precision - which can save memory and speed up computations.

- **use_auth_token** : To authenticate and download private models that require access permissions.
- **device** : To specify where the model should run - ex: CPU or GPU.

Cache the Model Locally

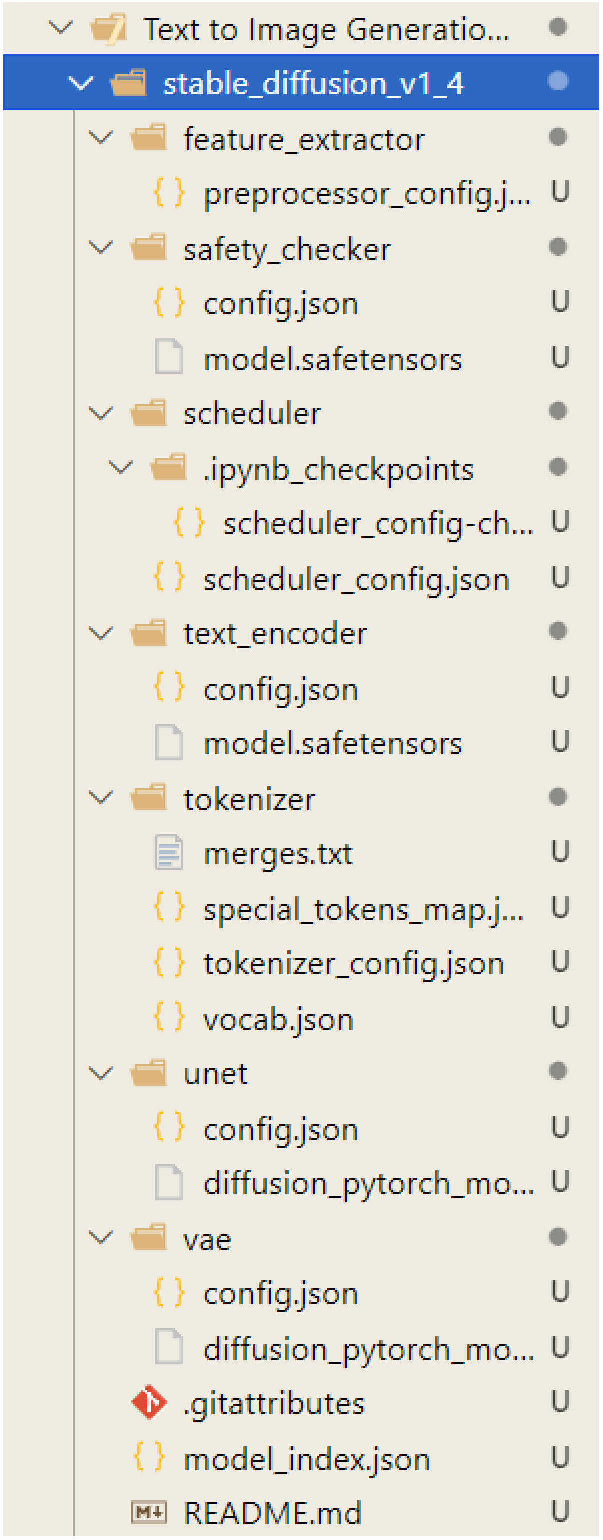
Once the model is downloaded - it is **cached** locally on our machine. This means that **subsequent call**s to from_pretrained with the same model identifier will load the model from the **local cache** avoiding redundant downloads.

Return a Ready-to-Use Pipeline

The method returns an **instance** of the **StableDiffusionPipeline** that is fully set up and ready to generate images from text prompts.
we can then use this pipeline to generate images - fine-tune the model or perform other tasks that the model is capable of.

The Downloaded model folder's structure

The model directory doesn’t contain **Python files** because it’s meant to store the **model's data** - not the code. The actual Python code is found in **external libraries** like **diffusers** or in **custom scripts** we write. This approach keeps the model files and code separate - allowing for flexibility - reusability - ease of distribution



Load the Stable Diffusion pipeline from the local directory

In [ ]:
```
# Load the Stable Diffusion pipeline from the local directory

device = "cuda" if torch.cuda.is_available() else "cpu"
```

```python
pipe = StableDiffusionPipeline.from_pretrained("./stable_diffusion_v1_4", torch_dty
pipe.to(device)
```

```
Loading pipeline components...:   0%|          | 0/7 [00:00<?, ?it/s]
```

Out[ ]:

```
StableDiffusionPipeline {
  "_class_name": "StableDiffusionPipeline",
  "_diffusers_version": "0.30.0",
  "_name_or_path": "./stable_diffusion_v1_4",
  "feature_extractor": [
    "transformers",
    "CLIPFeatureExtractor"
  ],
  "image_encoder": [
    null,
    null
  ],
  "requires_safety_checker": true,
  "safety_checker": [
    "stable_diffusion",
    "StableDiffusionSafetyChecker"
  ],
  "scheduler": [
    "diffusers",
    "PNDMScheduler"
  ],
  "text_encoder": [
    "transformers",
    "CLIPTextModel"
  ],
  "tokenizer": [
    "transformers",
    "CLIPTokenizer"
  ],
  "unet": [
    "diffusers",
    "UNet2DConditionModel"
  ],
  "vae": [
    "diffusers",
    "AutoencoderKL"
  ]
}
```

pipe.to(device)

This line **transfers** the **entire model** including its weights and components to the specified device either **GPU** or **CPU**.

Define the Generate Image function

```
In [ ]: # Define the Generate_Image function

def Generate_Image():
    try:
        with autocast(device):
            result = pipe(prompt_text.get(), guidance_scale=8.5)
            print(result)
            image = result["images"][0]

            for i in len(result) :
                print(result["images"][i])

        # Resize the image to fit the label dimensions
        image = image.resize((label_image.winfo_width(), label_image.winfo_height()

        # Save and display the image
        image.save('Generated_Image.png')
        img = ImageTk.PhotoImage(image)
        label_image.configure(image=img)
        label_image.image = img
    except Exception as e:
        print(f"An error occurred: {e}")
```

result = pipe(prompt_text.get() , guidance_scale = 8.5)

guidance_scale is a **parameter** from **1** to **10** that controls how **strongly** the model should **follow** the given **text prompt** during image generation.

Models that **generate images** - especially in the context of tasks like text-to-image synthesis - often **return multiple images**.

result is typically a **dictionary**. This dictionary contains **various** pieces of information - one of which is a **list of images**. The exact structure of result would depend on the implementation of pipe - but in this context :

- result is a **dictionary**.
- One of the keys in the dictionary is - **images**.
- The value associated with the - **images** - key is a **list** - and the **items** in this list are usually **image objects**.

Define the generate_image button

In [ ]:

```
# Generate Image button

generation_btn = ctk.CTkButton(master=root, height=40, width=120, font=("Arial", 20
generation_btn.configure(text="Generate Image")
generation_btn.place(x=210, y=60)
```

In [ ]:

```
# Start the GUI loop
root.mainloop()
```

The Whole Code

In [ ]:

```
import tkinter as tk
import customtkinter as ctk
from PIL import Image, ImageTk
import torch
from torch import autocast
from diffusers import StableDiffusionPipeline

# the root

root = tk.Tk()
root.geometry("600x700")
root.title("Text To Image Generation")

# text prompt

prompt_text = ctk.CTkEntry(master=root, height=40, width=580, font=("Arial", 20), t
prompt_text.place(x=10, y=10)


# a Label that contains the image

label_image = ctk.CTkLabel(master=root, height=570, width=580,fg_color="navy",text=
label_image.place(x=10, y=118)




# ---------------------------------------------------------------------------

# Load the Stable Diffusion pipeline from the local directory

device = "cuda" if torch.cuda.is_available() else "cpu"
pipe = StableDiffusionPipeline.from_pretrained("./stable_diffusion_v1_4", torch_dty
pipe.to(device)




# Define the Generate_Image function

def Generate_Image():
    try:
        with autocast(device):
```

```
            result = pipe(prompt_text.get(), guidance_scale=8.5)
            image = result["images"][0]

        # Resize the image to fit the label dimensions
        image = image.resize((label_image.winfo_width(), label_image.winfo_height()

        # Save and display the image
        image.save('Generated_Image.png')
        img = ImageTk.PhotoImage(image)
        label_image.configure(image=img)
        label_image.image = img
    except Exception as e:
        print(f"An error occurred: {e}")




# Generate Image button

generation_btn = ctk.CTkButton(master=root, height=40, width=120, font=("Arial", 20
generation_btn.configure(text="Generate Image")
generation_btn.place(x=210, y=60)

# Start the GUI loop
root.mainloop()
```

```
Loading pipeline components...:   0%|          | 0/7 [00:00<?, ?it/s]
  0%|          | 0/50 [00:00<?, ?it/s]
```

```
C:\Users\MTechno\AppData\Local\Temp\ipykernel_19432\405013505.py:46: DeprecationWarn
ing: ANTIALIAS is deprecated and will be removed in Pillow 10 (2023-07-01). Use Resa
mpling.LANCZOS instead.
  image = image.resize((label_image.winfo_width(), label_image.winfo_height()), Imag
e.ANTIALIAS)
```

Encountered Errors

Cannot initialize model with low cpu memory usage because accelerate was not found in the environment. Defaulting to low_cpu_mem_usage=False. It is strongly recommended to install accelerate for faster and less memory-intense model loading.

safety_checker\model.safetensors not found.