how to run the visual language model

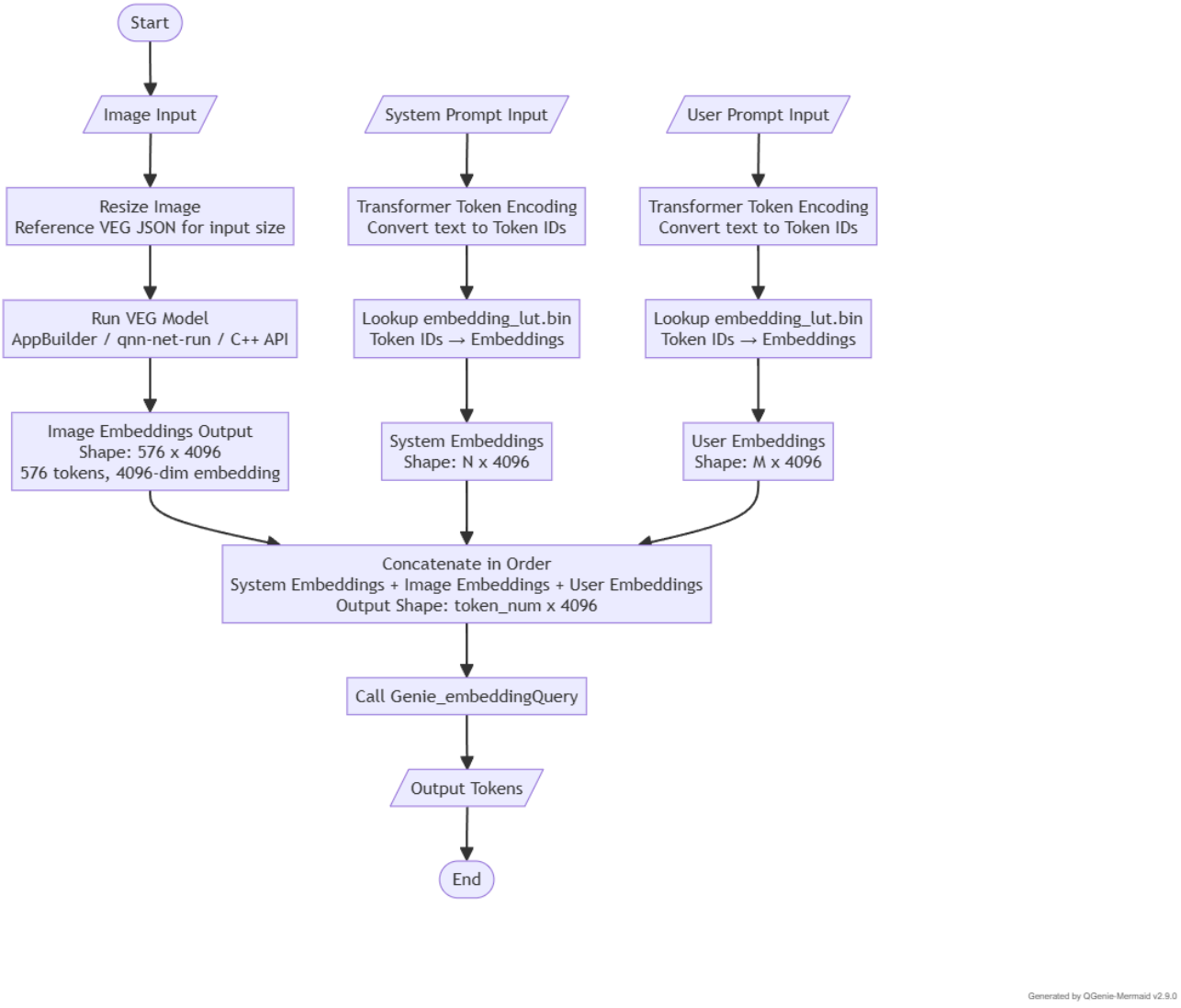
In our VLM deployment example, we use LLaVa 1.5-7B as the demonstration model.

The overall process consists of the following main steps:

1. Pre-compile the model
2. Input pre-processing
3. Execute inference using libGenie.so
4. How to integrate with OLLaMa server

The diagram below illustrates the workflow of the first three steps, providing an initial conceptual overview.

The high-level flow of this process is illustrated in the diagram below. The following sections will expand on each step in detail and provide more guidance on the implementation process.!



According to the workflow diagram, once we have prepared all the required files (including the VEG file, LUT file, and the module responsible for generating text), and completed the HuggingFace configuration—either by using *online get* or *cached* mode

connect to Ollama server

This section can be referenced to learn How OpenWebUI is deployed on the IQ-9.

At the API level, it is necessary to check whether the incoming request contains image data.

When OpenWebUI communicates with the Ollama server, image data is transmitted using Base64 encoding.

As a result, the Base64-encoded image must be decoded and converted into an OpenCV-compatible image format before being used in the downstream pipeline.

```
def base64_to_opencv(base64_string):

    """

    Converts a Base64 encoded image string to an OpenCV image (numpy array).

    """

    # 1. Decode the Base64 string into bytes

    img_bytes = base64.b64decode(base64_string)

    # 2. Convert the bytes into a 1D NumPy array

    im_arr = np.frombuffer(img_bytes, dtype=np.uint8)

    # 3. Use OpenCV to decode the NumPy array into an image format (e.g., BGR)

    # The flags argument cv2.IMREAD_COLOR specifies to load a color image

    img = cv2.imdecode(im_arr, flags=cv2.IMREAD_COLOR)

    return img
```

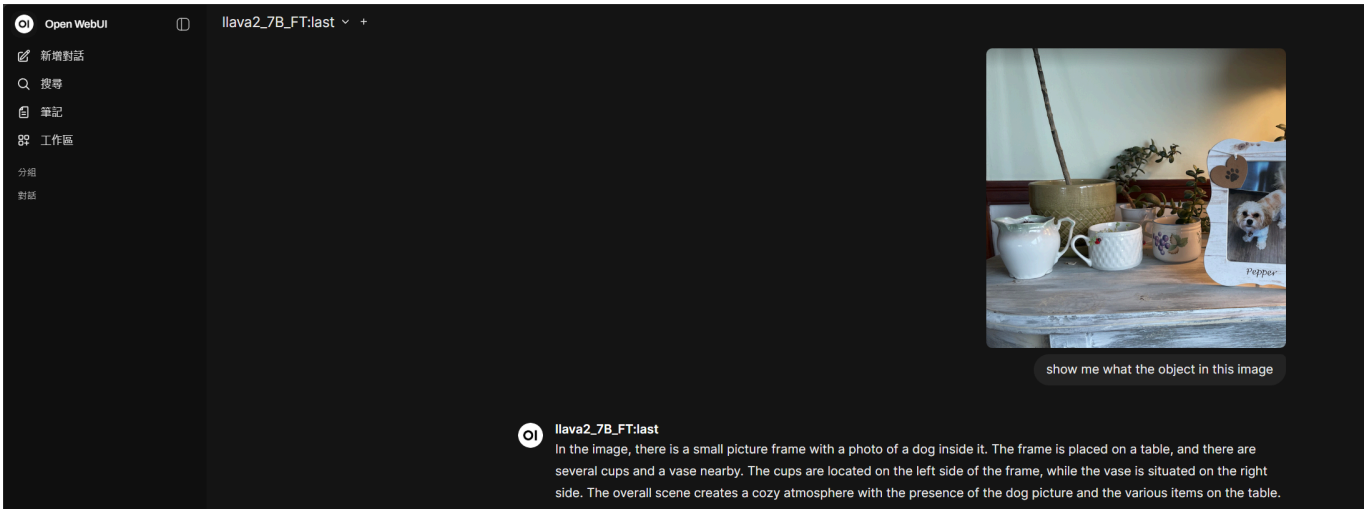
The Ollama server receives the payload in the following format.

After decoding the Base64-encoded image data using base64_to_image, the images may be concatenated or grouped into batches and forwarded to LLaVA for processing.

Afterward, the outputs can be merged to generate a final response or to support other user-defined post-processing logic.

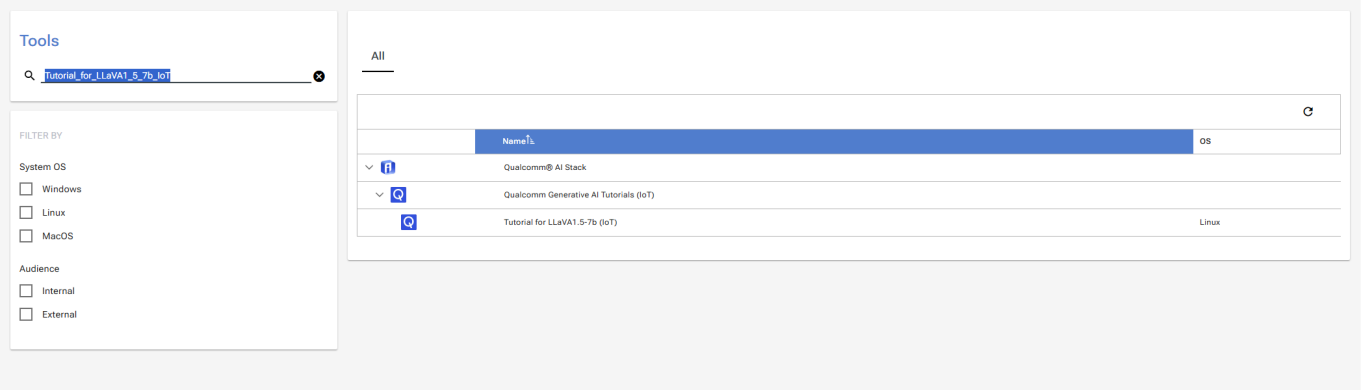
```
    data = await request.json()
     messages_dict = data.get("messages", {"role":"user", "content":"not data"})[-1]
     content = messages_dict.get("content", "")
     images = messages_dict.get("images", [])
```

The final result is as follows.



pre compile the model

The first step is to download the tutorial package from QPM.



Once you enter QPM, go to the Tools section and search for

“Tutorial_for_LLaVA1_5_7b_IoT” to find the tutorial package.
You may also perform the same steps using the QPM CLI, as shown below:

```
qpm-cli --login
qpm-cli --license-activate Tutorial_for_LLaVA1_5_7b_IoT
qpm-cli --extract Tutorial_for_LLaVA1_5_7b_IoT
```

If the package has already been downloaded, you can extract it directly by specifying the full path to the .qik file.

Within the tutorial, Example 1 and Example 2 guide you through the process of generating the necessary binary files required for later deployment.

The deployment requires three main components:

- The vision encoder (VEG) output,
- The language model, and
- A lookup table (LUT).

← 返回：how to run the visual language model

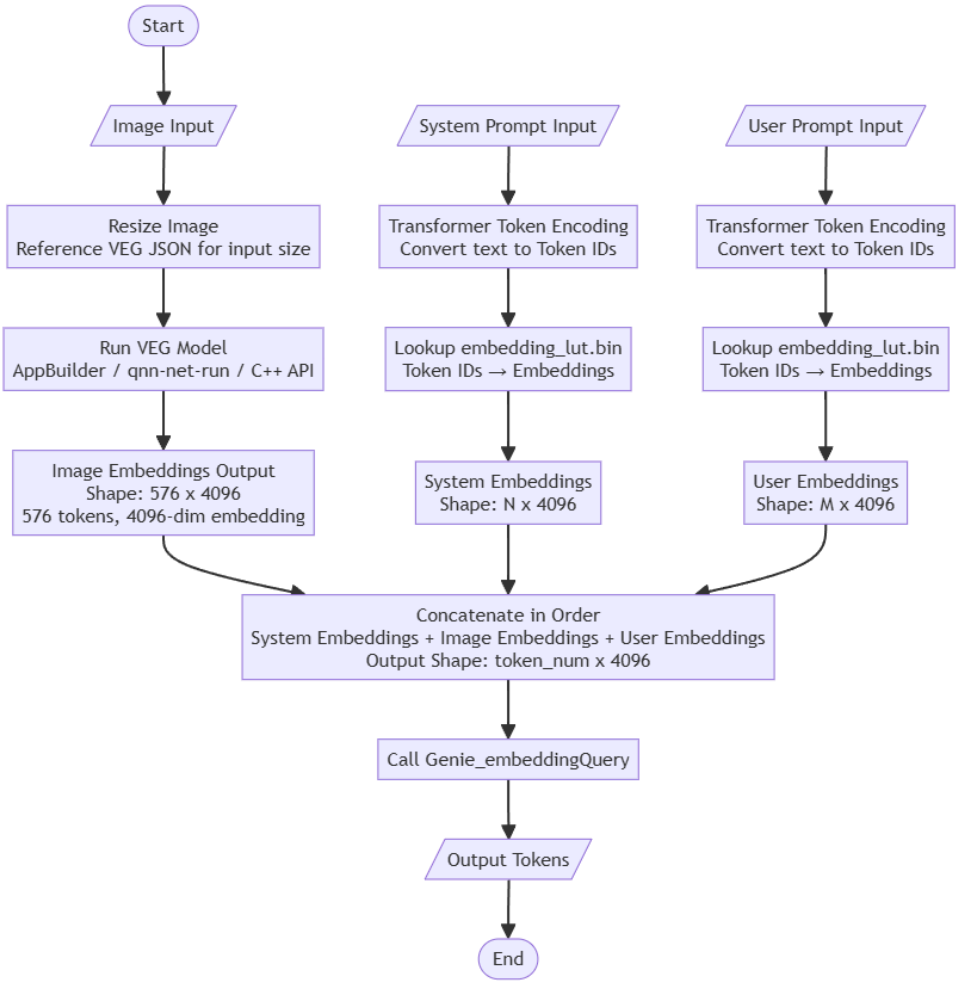
input pre processing

To ensure proper tokenization using an actively maintained tokenizer, the workflow requires performing an HF authentication login. This allows access to the tokenizer resources needed for correct token conversion or you can use the cache by pre-download.

```
from transformers import AutoProcessor
#this need use the hugging face to download some meta data
hf_processor = AutoProcessor.from_pretrained("llava-hf/llava-1.5-7b-hf", revision = "a272c74")
token_ids = hf_processor.tokenizer.encode(prompt)
```

For the VEG portion, we use QNN to obtain the embedding outputs. This can be done either with Onnxruntime by Onnx-warp-context binary file or qnn-net-run. However, we recommend using Onnxruntime, as relying on system calls to trigger external executables is generally not suitable for production deployments, you can see the next section to know How to Run the CV Base Model on device.

The high-level flow of this process is illustrated in the diagram below. The following sections will expand on each step in detail and provide more guidance on the implementation process.!



According to the workflow diagram, once we have prepared all the required files (including the VEG file, LUT file, and the module responsible for generating text), and completed the HuggingFace configuration—either by using *online get* or *cached* mode—we can begin introducing the left section of the process: converting images into embeddings.

converting images into embeddings

Since the example uses **LLaVA 1.5–7B**, image preprocessing is required. You may refer to the following code snippet to convert images into the required format. After preprocessing, you can follow the Onnx Runtime for CV Base model to run the VEG model and obtain the desired output for subsequent processing.

Preprocessing Code:

```
def load_and_preprocess_image(cv_image, processor):

    '''
    the process is come from AutoProcessor.from_pretrained("llava-hf/llava-1.5-7b-hf"

    Steps:

    1. expand image to be square, which llava_processor does not do

    2. Pass image through image processor from llava

    3. Transpose preprocessed image to NHWC to get an image tensor that will be passed to the VEG on device

    '''

    color_converted_image = cv2.cvtColor(cv_image, cv2.COLOR_BGR2RGB)



    # 3. Convert the NumPy array (OpenCV image format) to a PIL Image

    image = Image.fromarray(color_converted_image)



    # Background color in expand2square is image processor mean:

    image_expanded = expand2square(image, tuple(int(x * 255) for x in processor.image_processor.image_mean))

    preprocessed_image_tensor = processor.image_processor(image_expanded, return_tensors='pt')
["pixel_values"].detach().numpy()



    # Convert preprocessed_image_tensor from NCHW to NHWC

    preprocessed_image_tensor = np.transpose(preprocessed_image_tensor, (0, 2, 3, 1)).copy()

    return preprocessed_image_tensor
```

Combine the input

In a typical VLM usage scenario, a system prompt is required to specify the AI's current objective.
Images and questions are then provided as additional inputs, forming a workflow similar to a flowchart.

Based on this design, the system prompt pool can be merged with the other inputs into a single combined input.

> Note that the img_embedding represents the feature matrix produced earlier by running the CV-based model through QNN.

In real-world usage, not all queries include an image as input.
As a result, when no image is present, the system prompt can be omitted or replaced with an alternative prompt set that is more suitable for text-only interactions.

```
    def get_embeddings(self, token_ids):
        if not self.lookup_table_np is None:
            token_embeddings =  []
            # Get embedding for each token:
            for token_id in token_ids:
                token_embeddings.append(self.lookup_table_np[token_id, :])
            # Stack all token embeddings together:
            token_embeddings_np = np.stack(token_embeddings, axis=0)
            return token_embeddings_np
        else:
            return token_ids

    def merge_promt_with_img(self, prompt, img_embedding):
        if not img_embedding is None:
            system_prompt = {
                "prompt": {
                    "type": "default",
                    "inst-tags": ["USER: ", "ASSISTANT: "],
                    "sys-prompt": "A chat between a curious human and an artificial intelligence assistant. The
assistant gives helpful, detailed, and polite answers to the human's questions. \n IMAGE :"
                }
            }

            sys_prompt_for_processing = system_prompt["prompt"]["sys-prompt"]
            sys_prompt_tokens = utilize.run_tokenizer(sys_prompt_for_processing, self.hf_processor)
            sys_prompt_token_embeddings = self.get_embeddings(sys_prompt_tokens)

            # In first prompt, we use a "\n" instead of the first inst-tag:
            prompt_for_processing = "\nUSER: " + prompt + " " + system_prompt["prompt"]["inst-tags"][1]
            prompt_tokens = utilize.run_tokenizer(prompt_for_processing, self.hf_processor)
            prompt_token_embeddings = self.get_embeddings(prompt_tokens)

            # Concatenate image and token embeddings into multimodal embeddings:
            multimodal_embeddings = np.concatenate([sys_prompt_token_embeddings, np.squeeze(img_embedding),
prompt_token_embeddings], axis=0)
        else:
```

```
            prompt_tokens = utilize.run_tokenizer(prompt, self.hf_processor)

            multimodal_embeddings = self.get_embeddings(prompt_tokens)


        return multimodal_embeddings
```

> The lookup_table is produced during the preprocess stage, as described in the corresponding section.
> It is an essential file that will be reused throughout the subsequent pipeline stages.
>
> Additionally, utilize.run_tokenizer behaves identically to hf_processor.tokenizer.encode(prompt) in terms of functionality.
> The different naming in this demo is solely for logging and traceability purposes.

← 返回：how to run the visual language model

execute inference using libgenie.so

It is recommended to read the How to Run the Large Language Model chapter before proceeding with this section.

This section builds upon that chapter and describes the required changes needed to complete the final deployment.

The only required change is to replace the original **GenieDialog_Query** function with **GenieDialog_embeddingQuery**.

This function includes two fallback paths that need to be implemented.

The first fallback behaves identically to the original logic.

The second fallback introduces an additional t2e_callback, which can be implemented by following the example below.

```
       EMBEDDING_CALLBACK = ctypes.CFUNCTYPE(None, ctypes.c_uint32,ctypes.c_void_p, ctypes.c_uint32,
ctypes.c_void_p)

        def t2e_callback(token_id, embedding_as_input, embedding_size, user_ptr):

            #print(token_id)

            embedding = self.lookup_table_np[token_id, :]

            float_ptr = ctypes.cast(embedding_as_input, ctypes.POINTER(ctypes.c_float))

            ctypes.memmove(float_ptr, embedding.ctypes.data, embedding.nbytes)

        self.embed_cb = EMBEDDING_CALLBACK(t2e_callback)
```

The example shown above demonstrates one possible implementation, where Python is used to pass data into the C extension library.

For more detailed configuration options and usage parameters, please refer to the documentation provided here.

← 返回：how to run the visual language model

how openwebui is deployed on the iq 9

We use Ollama as an example to describe how we can integrate it into the product application in order to avoid making too many changes to the upper-layer application.
First, we need to install Ollama and then disable its backend server. The default backend server port of Ollama is **11434**, so we will install and start our own backend server using the following commands:

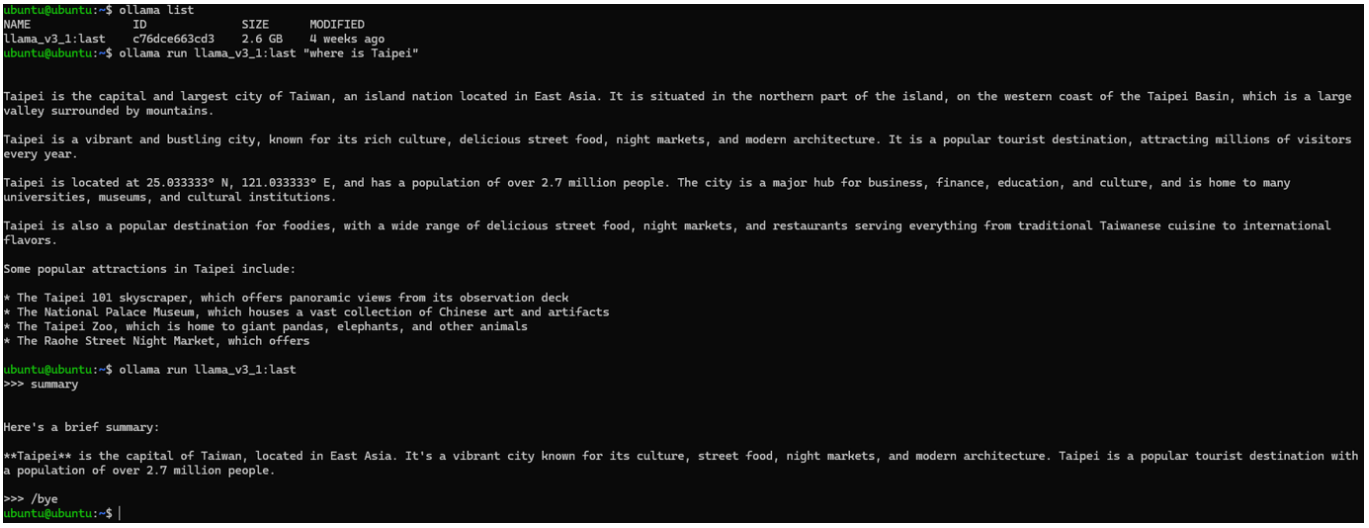
```
#Install
fastapi
uvicorn
snap install ollama
snap install docker
pip install transformers

#Run ollama backend
snap stop ollama
source ollama_env/bin/activate
python -m uvicorn llm_server:app --host 127.0.0.1 --port 11434 --workers 1
snap start ollama
```

Let us start building our own **llm_server**. To ensure that Ollama can function properly, we at least need to implement the following backend server endpoint via FastAPI:

> / [root method: "GET", "HEAD"]
> /api/tags
> /api/show
> /api/generate
> /api/chat

You can follow this to setup the API in our own Ollama backend server

Finaly we can test it by Ollama CLI:



How to configure OpenWebUI to create an interactive interface for LLM usage

We need to install OpenWebUI first, and it can be used through **docker-compose.yml**. But you must first ensure that you have Docker installed.
snap install docker
docker compose up -d
You can refer to the following for the docker-compose.yaml

```
version: '3.8'

services:

  open-webui:

    image: ghcr.io/open-webui/open-webui:main

    container_name: open-webui

    network_mode: "host"

    volumes:

      - ./:/app/backend/data



volumes:

  open-webui-data:
```

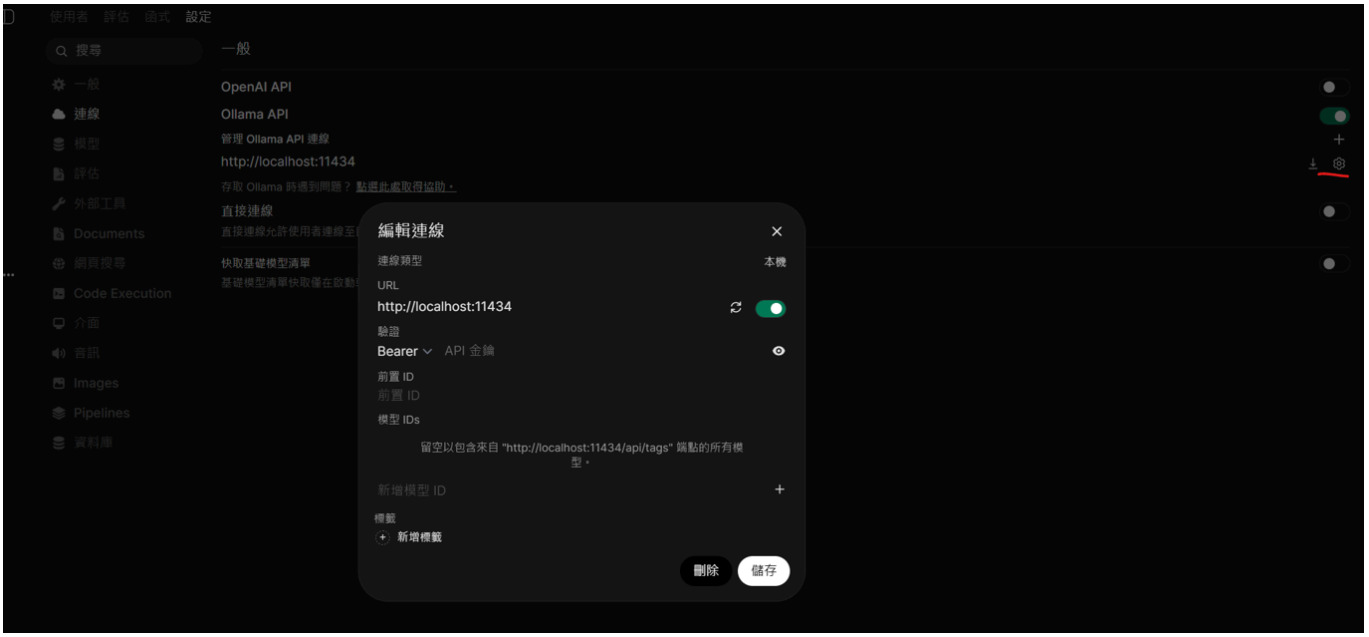
for run the server:

```
docker compose up -d
sudo docker start open-webui
```

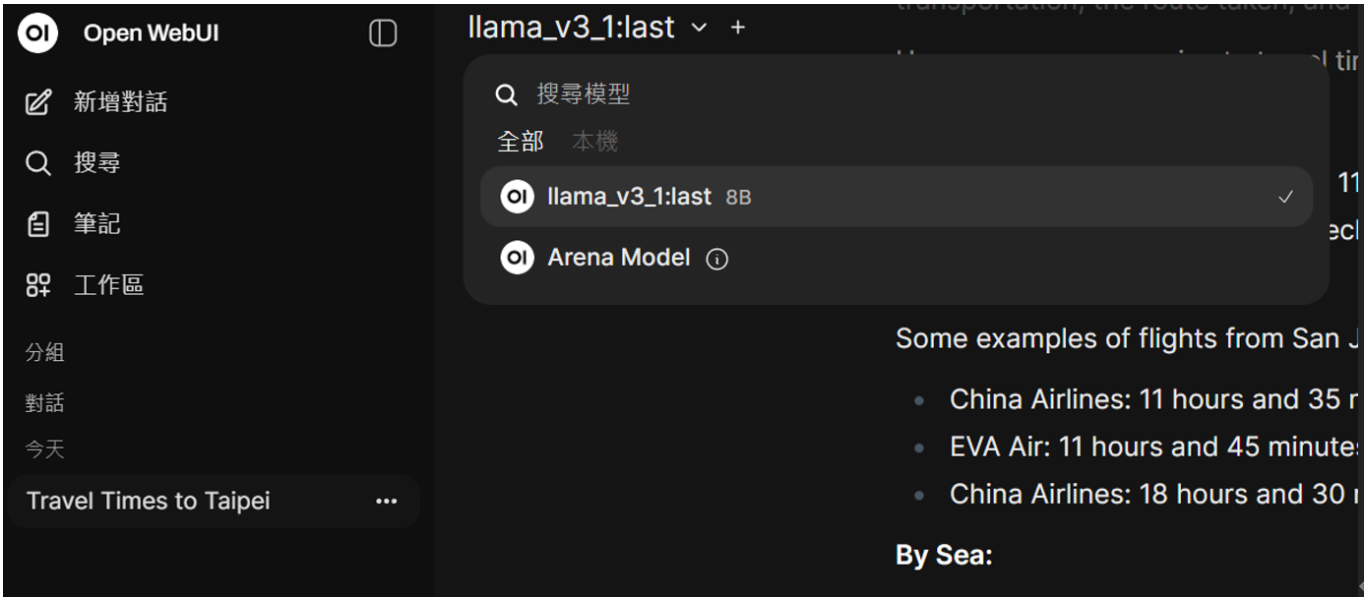
Setting using the Ollama API for OpenWebUI

After entering, click the user avatar in the upper right corner and go to **Settings**.
Then, in the window that appears, find **Manager Settings** in the lower-left corner.
After clicking it, go to **Connections**, and you can disable the OpenAI API, enable the Ollama API, and then set the

URL as shown in the picture below.



After then you can choose the Model like this



LLM_SERVER Example by using the pyGenie, this pyGenie plz flower this How to Run the Large Language Model to build yourself python package

---

← 返回：how to run the visual language model

how to run the cv base model

← 返回：input pre processing

onnx runtime for cv base model

← 返回：input pre processing

how to run the large language model

Running the tutorial from QPM

Find tutorials from QPM for LLama3.1_8B(IOT):
Download the jupyter notebook and run the examples to get the QNN context binary files.

Using AIHub

Run below commands on your x86 devices with python

pip install "qai-hub-models[llama-v3-1-8b-instruct]"

python -m qai_hub_models.models.llama_v3_1_8b_instruct.export --device "Dragonwing IQ-9075 EVK"

Then you will get QNN context binary files.

Using Genie with Python

For Python-based deployment, the primary library required is:

> **libGenie.so**
> **python3.10+**
> huggingface metadata or auth login

Ensure that Python is able to locate this library. You may specify the library path manually if needed. However, if you choose to rely on the **systembuiltin Genie library** no additional environment variable configuration is required.

If you have previously set custom environment variables (for example, LD_LIBRARY_PATH) and want to revert to the system default, you can simply remove them using:

```
unset LD_LIBRARY_PATH
unset QNN_SDK_ROOT
unset ADSP_LIBARY_PATH
```

If you want to use the specify version for Genie, please set env path:

```
export QNN_SDK_ROOT= <your QNN root>
export LD_LIBRARY_PATH = $QNN_SDK_ROOT/lib/aarch64-oe-linux-gcc11.2
export ADSP_LIBRARY_PATH=$QNN_SDK_ROOT/lib/hexagon-v73/unsigned
```

This ensures that the system uses the builtin library path without conflicts

---

build our own Python API

Next, we provide an overview of commonly used functions, and the concrete implementation can be referenced from this document as well as the **pyGenieAPI** example.

First, you can load libGenie.so through CDLL in ctypes. If you are using the system's default installation, the library is located at:

> **/usr/lib/libGenie.so**

Setting config json for dialog and profile

For the profile and dialog JSON mentioned above, you can refer to the document links below.
https://docs.qualcomm.com/doc/80-63442-10/topic/dialog_json.html
https://docs.qualcomm.com/doc/80-63442-10/topic/profile_json.html
Among them, you need to pay attention to the fact that the paths for the tokenizer and model must be absolute paths. In addition, the extensions field inside the backend of the engine is also a JSON path, and it must also be an absolute path.

Use libGenie

Then, following the diagram below,

```
def __init__(self):
        lib = ctypes.CDLL(lib_path)
        self.GenieDialogConfig_createFromJson = lib.GenieDialogConfig_createFromJson
        self.GenieDialogConfig_bindProfiler = lib.GenieDialogConfig_bindProfiler
        self.GenieProfile_create = lib.GenieProfile_create
        self.GenieProfileConfig_createFromJson = lib.GenieProfileConfig_createFromJson
        self.GenieDialog_create = lib.GenieDialog_create
        self.GenieDialog_query = lib.GenieDialog_query
        self.GenieDialog_reset = lib.GenieDialog_reset
        self.GenieDialogConfig_free = lib.GenieDialogConfig_free
        self.GenieDialog_free = lib.GenieDialog_free
        self.GenieProfile_free = lib.GenieProfile_free
```

you can use the common functions in sequence to initialize the model:

> 1. GenieProfileConfig_createFromJson
> 2. GenieProfile_create
> 3. GenieDialogConfig_createFromJson
> 4. GenieDialogConfig_bindProfiler
> 5. GenieDialog_create

```
        with open(config_json_path,"rb") as f:
             config_bytes = f.read()

         with open(profile_json_path,"rb") as f:
             profile_bytes = f.read()

         self.configHandle = ctypes.c_void_p()
         self.profileConfigHandle = ctypes.c_void_p()
         self.profileHandle = ctypes.c_void_p()
         self.dialogHandle_fn = ctypes.POINTER(_Impl)
         self.dialogHandle = self.dialogHandle_fn()

         #pipline
         self.GenieProfileConfig_createFromJson(profile_bytes, ctypes.byref(self.profileConfigHandle)) #1
         self.GenieProfile_create(self.profileConfigHandle, ctypes.byref(self.profileHandle))          #2
         ret = self.GenieDialogConfig_createFromJson(config_bytes, ctypes.byref(self.configHandle))    #3
         ret = self.GenieDialogConfig_bindProfiler(self.configHandle, self.profileHandle)              #4
         ret = self.GenieDialog_create(self.configHandle, ctypes.byref(self.dialogHandle))             #5
```

After these steps, you can call the model using **GenieDialog_query.**
For the function parameters, please refer to the [Genie API documentation](Genie C API - Qualcomm AI Runtime (QAIRT) SDK) below
In addition, GenieDialog_query requires a callback function, which can be wrapped using ctypes.CFUNCTYPE and the userData can be passed using an asyncio queue and asyncio loop events for stream mode.

```
def on_event(msg_ptr, code, user_data_ptr):
    ba = ctypes.cast(user_data_ptr, ctypes.POINTER(ctypes.py_object)).contents.value
    if msg_ptr:
        ba += msg_ptr

CALLBACK = ctypes.CFUNCTYPE(None, ctypes.c_char_p, ctypes.c_int, ctypes.c_void_p)
self.cb = CALLBACK(on_event)


#for stream mode
stream_queue = asyncio.Queue()
stream_loop = asyncio.get_event_loop()
def on_event_stream(msg_ptr, code, user_data_ptr):
    if code == GenieDialog_SentenceCode_Map["GENIE_DIALOG_SENTENCE_END"]:
        stream_loop.call_soon_threadsafe(stream_queue.put_nowait, None)
        print("", flush=True, end="\n")

    ba = ctypes.cast(user_data_ptr, ctypes.POINTER(ctypes.py_object)).contents.value
    if msg_ptr:
        ba += msg_ptr

    msg = msg_ptr.decode("utf-8") if msg_ptr else ""
    print(msg, flush=True, end="")
    stream_loop.call_soon_threadsafe(stream_queue.put_nowait, msg)
self.cb_stream = CALLBACK(on_event_stream)

## some code
promate = f"<|begin_of_text|><|start_header_id|>user<|end_header_id|> {promate} <|eot_id|>
<|start_header_id|>assistant<|end_header_id|>"
async_user_data = ctypes.py_object(bytearray())
b_promate = promate.encode('utf-8')
c_char_p_promate = ctypes.c_char_p(b_promate)
c_user_data_p = ctypes.cast(ctypes.pointer(async_user_data), ctypes.c_void_p)~

chat_thread = threading.Thread(target=self.GenieDialog_query, args=(self.dialogHandle, c_char_p_promate,
GenieDialog_SentenceCode_Map[send_method], self.cb_stream, c_user_data_p,))
chat_thread.start()
response = async_user_data.value.decode("utf-8")
```

For detailed usage, please refer to the **pyGenie** example code.
After encapsulating everything, the final calling pattern will look like the example shown below

```
genie_obj = pyGenieAPI_LLM(config_json_path, profile_json_path, modelInfo_json_path, lib_path = lib_path)

print("[Prompt]: where is Taipei")
genie_obj.chat("where is Taipei")

print("[Prompt]: Summary")
genie_obj.chat("where is Taipei")

genie_obj.shutdown()
```

Using Genie with C++

For customers who prefer to deploy using **C++**, a reference implementation is available under:

> **<QAIRT_ROOT_PATH>/example/Genie**

This directory contains example code demonstrating how to integrate and utilize Genie in a C++ application, which can serve as a helpful starting point for custom development

---

← 返回：how openwebui is deployed on the iq 9