ShopTalk Documentation

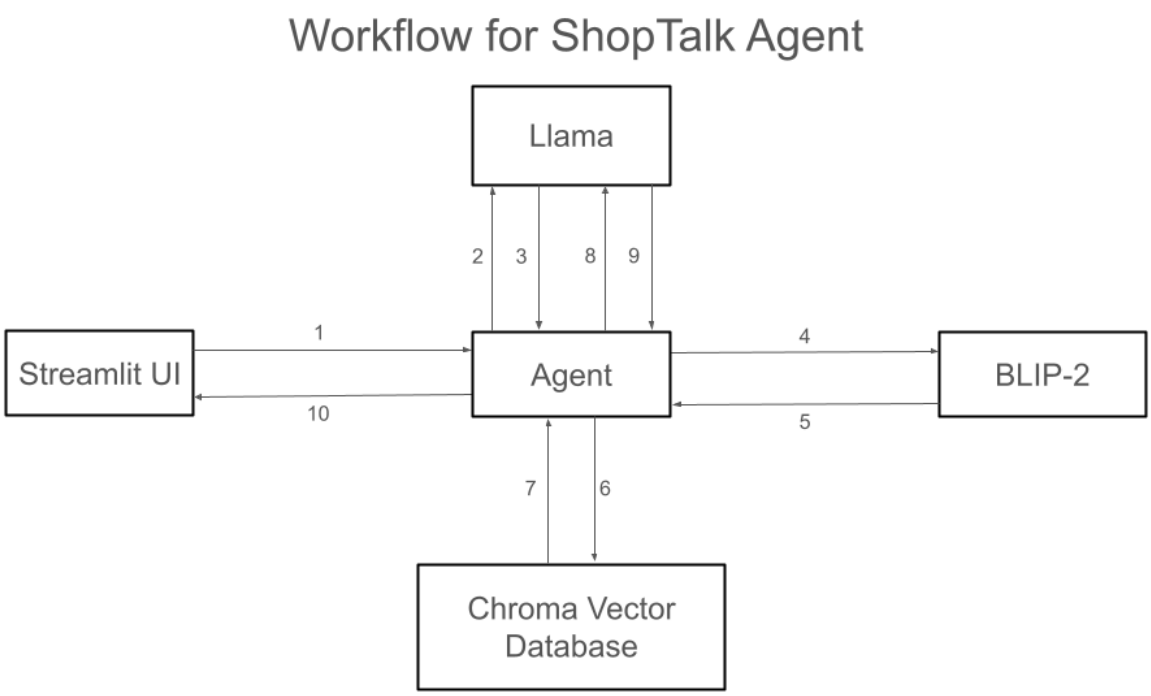
Overview

This implementation of the ShopTalk shopping assistant chat-bot uses a Streamlit front-end and LangGraph to tie together a Llama-3.1-8B-Instruct LLM, a Chroma vector database, and a BLIP-2 embedding model.

Architecture

The workflow for the ShopTalk agent is shown below. Each of the five rectangles corresponds to a Docker container.

1. The user interacts with the Agent through the Streamlit UI (code found in the `code/UI` directory), which passes the prompt and an optional image the user provides to the Agent.
2. The Agent (code found in the `code/Agent` directory) passes the prompt (but not the image) to the Llama LLM.
3. The Llama LLM processes the prompt and tells the Agent to call a tool (BLIP-2 and Chroma database) with a summary.
4. The Agent sends the prompt summary and the optional image to BLIP-2 (API code found in `code/Blip-2_API_endpoint`).
5. BLIP-2 calculates embeddings and returns them to the Agent.
6. The Agent forwards the embeddings to the Chroma Vector database.
7. The Chroma Vector Database returns the *N*k* closest matching product-image pairs, where *N* is the most images any product has and *k* is the number or products we want to return to the user.
8. The Agent pares down the *N*k* product-image pairs to *k* distinct products. It forwards the metadata about those products to the Llama LLM.
9. The Llama LLM produces a summary of the *k* products and passes it back to the Agent.
10. The Agent forwards that summary along with any applicable images of the products to the Streamlit UI, which displays them to the user.



Notes

1. The bot may have to prompt the user for additional information before it can provide useful text for an embedding. This case should only apply if the user doesn't provide an image.
2. There are multiple images for each product, so the multimodal vector database stores an embedding per image-item pair. Therefore, it returns *N*k* results, where *N* is the most images any product has and *k* is the number or products we want to return to the user.

Design Choices

Design Goals

1. Small models that, when combined, fit into the 12GB video RAM on the RTX 4070 Ti. One of our group members has two and the card is fast when compared to the commonly-used T4.
2. Multi-modal embeddings, so that we can create text-only, image-only, and unified text-image embeddings, which can all be compared (goal not completely met).
3. A chat-bot that can have a conversation with the user, not just look up a query and provide a summary of results.

BLIP-2

During the class introducing the capstone projects, there was a mention of multi-modal embeddings when ShopTalk was described. Therefore, we thought to take advantage of them. Unfortunately, most models that can do multimodal embeddings (such as Meta's ImageBind) can't produce a single embedding from multiple modalities; they can only produce an embedding for each modality, all of which lie in the same space. BLIP-2 is different, and can produce a combined text-image embedding.

When we started, we believed that BLIP-2 would put the text-only, image-only, and text-image multimodal embeddings in the same space. This thinking turned out to be wrong, and we ended up using Chroma's (the vector database) default embedding model, all-MiniLM-L6-v2, to produce a separate set of text-only embeddings.

We fine-tuned BLIP-2 on two datasets and evaluate the agent's performance with them (plus the pretrained one and the pre-fine-tuned Coco one) later in the document.

Chroma

ChromaDB, an open-source vector database tailored for AI applications, stands out for its scalability, ease of use, and robust support for machine learning tasks. ChromaDB is finely tuned to store and query vector data, making it the top choice for developing AI-driven applications that rely on semantic search, recommendation systems, and natural language processing.

ChromaDB is a vector database that offers several advantages over the other vector databases. Summary of the highlights are

1. Scalability and Performance
2. Advanced Query Capabilities
3. Support for multiple data types
4. Integration with popular deep learning frameworks
5. Security and data governance

There are several other databases like Milvus, FAISS, Pinecone etc each has unique strengths. The choice between these depends on the specific requirements of the project.

Phi-3.5-mini via Hugging Face to Llama-3.1-8B-Instruct via Ollama

We wanted a well-performing small model. Not having time to make detailed comparisons ourselves, we searched and found an article comparing RAG performance of various small LLMs, which placed Microsoft's recent Phi-mini models at the top of the chart. Since they only have 3.8B parameters and use bfloat16, we decided to use Phi-3.5-mini. A comparison of the speed of three Phi-mini models is made in the notebook `Phi-mini-tests`.

We had to discard the Phi models for two main reasons:

1. The ranking we found turned out to be for models fine-tuned for use with LLMWare, and by that point it was too late to switch frameworks.
2. We had to switch to Ollama to be able to stream output with LangChain/LangGraph, and the versions of Phi available for it do not support tool calling.

Therefore, we tested four other models: Ministral-8B-Instruct-2410, Mistral-7B-Instruct-v0.3, Llama-3.2-3B-Instruct, and Llama-3.1-8B-Instruct, the 1st, 2nd, and 4th with 8-bit quantization (we may quantize further later depending on memory requirements). In our tests, Ministral and Mistral both rarely called the tool despite being told to do so (and often claiming that they did!). Llama-3.2-3B-Instruct often called the tool with incorrect syntax, resulting in the tool not getting

called. So, we settled on Llama-3.1-8B-Instruct, which does call the tool, though perhaps over-eagerly.

LangChain to LangGraph

The tests of LangChain, leading to the decision to use LangGraph, are in the notebook `LangChain_to_LangGraph`. The choice to use LangGraph came down to the limitations of LangChain. As its name suggests, LangChain is a chain. The implication is that we cannot have the LLM call the tools, because there is always a chance that it will not call the tools. Therefore, if we just use a chain, we have to put the user's query through the embedding tool every time. But what if the user sends the chat-bot a prompt that is off-topic? It will just result in a nonsense response.

The solution to the conundrum above is to use LangGraph. It has the LLM provide instructions on which tool to call (if multiple are available) and follows them, or calls no tools if no such instructions are provided. In the latter case, it can send the output directly to the user. More details are found in the notebook.

Memory

The agent does have a memory. Previous chats may be accessed via the list in the left sidebar of the UI. This memory is implemented via LangGraph's MemorySaver class, which keeps track of separate chat threads. It keeps track of not only the text, but the image the user uploaded and the images sent to the user as well.

Note that a current limitation is the lack of unique sessions. Every instance of the interface currently interacts with the same chat memory.

Feedback Loop

The agent can take and save feedback in the form of thumbs-up and thumbs-down. Currently, it just saves the feedback in memory. Perhaps the feedback data could be analyzed in an automated fashion if we prompted the user to tell us what is wrong. A human could then review the results and determine how to act on them. However, without that, a human would need to review each piece of feedback to determine what is wrong and how to correct it.

Currently, most of the problems a user is likely to provide feedback about could be solved with improvements discussed below. Therefore, most feedback would not be very useful until those improvements are implemented.

Poetry Environments

There are two Python environments for pre-deployment work which were tested on Windows and Ubuntu-24.04 (under Windows Subsystem for Linux). The two environments are under the `python_envs` directory, then under the subdirectories `ShopTalk_py3.11` and `ShopTalk_py3.12` for Python 3.11 and 3.12, respectively. All the notebooks and Python files except `ChromaDemo` use one of these environments, though the latter notebook may still work

with them. As a rule of thumb, the `laguage_detection` notebook and anything to do with BLIP-2 use the 3.11 environment. The rest (except for `ChromaDemo`) use the 3.12 environment. This separation is necessary because the two environments require different versions of Transformers.

There are also three deployment environments, which are smaller than the pre-deployment environments, `py3.11_blip-2`, `py3.12_agent`, and `py3.12_ui`. They all have requirements.txt files for use when creating Docker images. We attempted to merge the BLIP-2 and Agent environments early on before using Ollama, but even the fine-tuned versions of BLIP-2 could not be loaded without LAVIS, which requires an early version of transformers that does not support the LLMs we were using, as demonstrated in the notebook `Blip-2-load-test`. We could probably merge them now that we're using Ollama to run the LLM, but since the work to separate them has already been done, we'll leave them separate.

Datasets

We use two datasets in this project. The first is the Amazon Berkeley Objects (ABO) Dataset, which has products found on Amazon along with their images and many other details typically found describing a product on Amazon's website. We use this dataset for both fine-tuning BLIP-2 and as the dataset the agent searches in. The second is the Marqo Google Shopping Dataset (marqo-GS-10M), which we only use for fine-tuning BLIP-2.

Exploratory Data Analysis

ABO Dataset

The EDA for the ABO dataset is in the notebook `ShopTalkEDA`. The notebooks is self-explanatory, but is broken into three parts:

1. Exploration of the dataset and extraction of pieces with English tags
2. English-language checking with a model and Google Translate. Multiple models are explored for this use on a sample of the dataset in the `language_detection` notebook.
3. Category (product_type) checking with Llama-3.2-11B-Vision-Instruct, the details of which are shown on a sample of the dataset in the `Huggingface_Llama_Vision` notebook. The code for running the checking on the full dataset is in the directory `Llama_data_checks`.

marqo-GS-10M Dataset

A brief exploration of the Marqo Google Shopping dataset is in the notebook `Marqo-GS-10M-EDA`. There wasn't much to explore, but the main take-aways are that there are no nulls, that queries have a varying number of products, and that the same product may show up in multiple queries.

Fine-Tuning

We only fine-tune BLIP-2 on both the ABO dataset and the Marqo Google Shopping dataset. From BLIP-2 we only use the Image Encoder (vision transformer) and the Q-Former, of which

we only fine-tune the Q-Former. There is no need for the LLM as we just need embeddings. The fine-tuning turned out to be one of the most difficult tasks as LAVIS, which is the framework Salesforce developed to develop their models, is very badly documented. The results of the saga are documented in the notebook `Blip-2-fine-tune`.

The code used to run the fine-tuning on the full datasets is in the directory `code/Blip-2_fine_tune`. It must be run in the Python 3.11 environment due to BLIP-2 and must be run on Linux due to using the nccl distributed backend, even if only running on one GPU.

There are two versions of the code, one for fine-tuning using the Google Shopping dataset and one for using the Amazon Berkeley Objects dataset. For the Google Shopping dataset, the fine-tuning requires the Marqo Google Shopping dataset in the `marqo_gs_data_dir` defined in `run_gs.py`. Inside that directory, there must be an `images` subdirectory with the images in it, and a subdirectory `marqo-gs-dataset/marqo_gs_full_10m` containing `query_0_product_id_0.csv` for the training annotations and `query_1_product_id_1.csv` for the validation annotations. The code will first fine-tune the model using the training data, saving the checkpoints and the training loss in the `save_dir` (defined in `run_gs.py`) every 5,000 batches along with a final save at the end of the training run. It will then validate it using the validation data. The only result from the validation is the loss, which is also saved in the `save_dir`.

For the Amazon Berkeley Objects dataset, the fine-tuning requires the dataset in the `abo_dataset_dir` defined in `run_abo.py`. Inside that directory, there must be a pickle file `abo_listings_file`, also defined in `run_abo.py`, containing the item metadata produced by one of the first four stages of the data pipeline described below (we took it after the third stage), and an `images/small` subdirectory with the images in it along with the images metadata at `images/metadata/images.csv`. The code will first fine-tune the model using the training data, saving the checkpoints and the training loss in the `save_dir` (defined in `run_gs.py`) every 5,000 batches along with a final save at the end of the training run. There is no validation run for reasons discussed in the `Blip-2-fine-tune` notebook.

After the model is fine-tuned, a database for it must be created using the fifth stage of the data pipeline described below, and the multimodal collection must follow the naming scheme in the `configuration` section of `Running ShopTalk` above. The model itself must also be placed in the `BLIP_2_DIR_LOCAL` directory defined in `docker/.env`. It must also be renamed to follow the aforementioned naming scheme.

Embeddings

When tallying up all the image-text pairs plus all the text-only items, over 500,000 embedding are needed. We need these all in the database before deployment. Therefore, to create the

embeddings, we use the code found in the directory `code/Blip-2_embeddings`. We create embeddings using BLIP-2 pre-fine-tuned (by Salesforce) on the Coco dataset, pretrained BLIP-2, BLIP-2 fine-tuned on the ABO dataset, and BLIP-2 fine-tuned on the Marqo Google Shopping dataset. The embeddings are saved to the Chroma database in the notebook `Chroma_DB_save_embeddings`. The text-only embeddings using the default Chroma model (all-MiniLM-L6-v2) are saved in the notebook `Chroma_DB_save_docs`.

RAG Evaluation

No current method exists to evaluate a RAG system that uses images and text. We also determine that the typical positive vs. negative result metrics do not take into account the subtleties of the problem. For example, if a person is searching for a pink shoe and the model returns a blue one, that's a much better result than a couch. Therefore, we created an ad-hoc metric specific to shopping. Details can be found in the `evaluation` notebook. The result is that the model fine-tuned on the Google Shopping dataset does best for multimodal, and Chroma's default embedding model for text, all-MiniLM-L6-v2, does great for text-only.

An attempt at evaluation using a pre-existing metric (DeepEval) was made in the notebook `RAG-evaluation-DeepEval`. As seen in the notebook, it's just for text and it didn't go well, partly because the input prompts are just way too descriptive.

Latency

We test the latency by calling the agent directly. The code can be found in `run-perf-test.py` with the environmental variables listed in `setenv.bat`.

We run two cases, with and without images. In each case, we cycle through three different model calls 100 times, for a total of 300 calls, each starting a new chat thread. We also test on two different machines, an AWS EC2 instance with an L4 GPU (g6.2xlarge) and a local machine with an RTX 4070 Ti GPU. Due to the 12GB memory limitation on the RTX 4070 Ti, we have to run the LLM with q5_K_M quantization, while we run with the default q8_0 quantization on the L4.

Machine	Percentile	With Image	Without Image
AWS EC2 L4 GPU q8_0 quantization	mean	779 ms	694 ms
	95th	999 ms	765 ms
	99th	1037 ms	915 ms
Local RTX 4070 Ti GPU q5_K_M quantization	mean	330 ms	327 ms
	95th	382 ms	360 ms

	99th	404 ms	378 ms

Aside from the memory (24 GB GDDR6 vs 12 GB GDDR6X), the L4 and RTX 4070 Ti GPUs have similar specs, so most of the performance difference can be attributed to the difference in quantization of the LLM.

Note that if a chat is continued beyond a single prompt, the latency increases.

Data Pipeline

The data pipeline, which takes a dataset and produces a database for use with a model, can be run by running `run_pipeline.sh` script from the root of the ShopTalk directory structure. The pipeline configuration file is `pipeline_config.toml`, where explanations of each variable are provided. The script runs through all the files in `code/Data_pipeline`. It expects a certain directory structure and naming scheme, with variables that can be configured in `blue` (others are explained in comments in the file):

1. A working directory `working_dir` where it keeps intermediate metadata files
2. A dataset directory `abo_dataset_dir` where dataset can be found. There must be three directories inside:
    a. `listings/metadata` containing `.json` or `.json.gz` files (only one kind to avoid duplicates) that contain the metadata in json format.
    b. `images/small` containing all the images referred to in the metadata mentioned above.
    c. `images/metadata` containing `images.csv` or `images.csv.gz` mapping the `image_id`s in the metadata in part `a` above to the images in the directory in part `b` above.
3. A `models_dir` containing the BLIP-2 model for creating the multimodal embeddings. The BLIP-2 model file must have the form `blip-2-model_selection.pt`, where `model_selection` is specified below.
4. The variable `model_selection` is used to specify which BLIP-2 model to use in `3` above. The provided model options are `gs`, `abo`, `pretrain`, and `coco`. It is also used to specify the multimodal Chroma collection, which has the name `blip_2_model_selection_multimodal`.
5. An `embeddings_dir` where the embeddings from BLIP-2 will be saved before they are added to the Chroma database.
6. A `chroma_dir` where the Chroma database is stored.
7. The processor files, `blip-2-processors-pretrain.pkl` and `blip-2-processors-coco.pkl`, along with the language detection model `language_detector.tflite`, in the ShopTalk directory `assets`.

Both the `ShopTalk_py3.11` and `ShopTalk_py3.12` environments must be installed (run `poetry install` in the respective directories. The `python311` and `python312` variables in `run_pipeline.sh` must point to the Python binaries that Poetry creates for each of the respective environments.

The pipeline Python files are each labeled with a number at the end corresponding to the order that they must be run in. Each file is documented in a docstring at the start of the file. Note that the first four files use the Python 3.12 environment, while the last uses the 3.11 environment. Once the last file completes its processing, the produced Chroma database will need to be moved to the directory `CHROMA_DIR_LOCAL` in `docker/.env` points to or the variable changed to point to the appropriate directory for the agent to use it. The listings pickle file produced by the fourth stage of the pipeline must also be placed in `ABO_DIR_LOCAL` and the variable `ABO_LISTINGS_FILE` changed to its name.

The pipeline files perform the following tasks:

1. The script `english_tags_1.py` reads the metadata from the `.json.gz` files (in 2a above) into a Pandas dataframe. It filters out non-English tagged items and saves the result into a pickle file with the suffix `preprocess-1.pkl`.
2. The script `product_type_spaces_2.py` loads the metadata produced by `english_tags_1.py` from the pickle file with suffix `preprocess-1.pkl` and replaces any '_' in product_type in the metadata with a ' ' and adds spaces where they're missing. It saves the result in a pickle file with the suffix `preprocess-2.pkl`.
3. The script `english_check_3.py` loads the metadata saved by `product_type_spaces_2.py` from the file with suffix `preprocess-2.pkl` and runs English-language checks on the metadata using Google's MediaPipe and optionally Google Cloud language detection. If cloud language detection is run, it saves the detection results in a pickle file with suffix `cloud-language-detections.pkl`. It then drops the metadata rows detected as non-English. It saves the metadata result in a pickle file with the suffix `preprocess-3.pkl`.
4. The script `product_type_verification_4.py` loads the metadata saved by `english_check_3.py` from the file with suffix `preprocess-3.pkl`. It runs images (in 2b above) and their corresponding product_types through Llama-3.2-11B-Vision-Instruct to check if the items are categorized correctly. It removes the product_type from a metadata row if most of the model determines that most of the images do not match the product_type. It saves the metadata result in a pickle file with suffix `reprocess-4.pkl`.
5. The script `embed_to_chroma_5.py` loads the metadata saved by the previous script from the file with the suffix `preprocess-4.pkl`. It then transforms the metadata rows into strings and creates a multimodal (text + image) embedding for each item-image pair, creating multiple embeddings files in the `embeddings_dir`. It then adds each embedding to the collection `embeddings_model_selection_multimodal`, with the `image_id` and `item_id` as metadata, in the Chroma database. It also adds each

metadata row (as a text string) to a separate `text_only` collection in the database with the `item_id`s as `ids`.

Other Notebooks

Blip-2-load-test

Having multiple Python environments is a pain, especially for deployment. Therefore, we had the idea to try saving a fine-tuned BLIP-2 model and the preprocessing functions then loading it into another environment without LAVIS installed. As seen in the notebook, PyTorch refused to load the model.

Ollama_Llama_vision

For testing Llama-3.2-11B-Vision-Instruct running through Ollama. Ollama added a few seconds of overhead when compared to running the model loaded through the Hugging Face transformers module. Those few seconds are critical when handling hundreds of thousands of pieces of data, leading us to use Hugging Face to run this model.

Improvements

There are multiple ways this agent could be improved. Some suggestions are below.

1. ESCI (Exact, Substitute, Complement, Irrelevant) classification. No dataset will have everything a user could want, which can be concluded from the fact that some products a user could want simply don't exist. Therefore, the database will, by necessity, sometimes return products that aren't what the user requested. It would be useful to know which of these categories a product falls into. We found at least two datasets for such training on the Internet. One possibility is to train a BERT model with classification output to perform such a task.
2. Fine-tune the BLIP-2 vision transformer. Due to resource limitations, we didn't try fine-tuning the vision transformer, as was done in the paper with the Coco dataset. We did not have time to explore the feasibility of memory-conserving methods such as LORA due to the time spent figuring out how to fine-tune BLIP-2 in the first place.
3. Explore BLIP-3. It was released in August 2024 and didn't show up in our searches until after we had already fine-tuned BLIP-2. It may improve the multimodal embeddings.
4. Explore more LLMs. While Llama-3.2-8B-Instruct does pretty well, it still makes mistakes at times, such as not listing products that were found or listing them out of order. Other LLMs may not make such mistakes.
5. Automated feedback loop. We easily gather feedback from users already, but it's useless if we don't use it. An automated way to use it to further fine-tune our models would be useful. However, as noted above, until other improvements are implemented, most feedback is unlikely to be useful.
6. Integrate the ABO spin and 3D images. We didn't have time to do anything with those.
7. Batch queries. We didn't do this at all, as we were low on time and were already running low on memory on the RTX 4070 Ti. Getting batching up and running on AWS may have taken days, and we didn't want to burn through so many credits. Furthermore, LangGraph doesn't appear to natively support it, so it would require coding outside of LangGraph.

8. Unique user sessions. We currently have no way to distinguish between users. Every user sees the same interface in the same state, so multiple users could interact with the agent via the same chat thread at the same time.
9. More user images per thread/multiple use of the user image. Currently the image submitted to the agent is only used with the next prompt in the chat thread, and only one is allowed per thread. There is no way to submit the image again or submit another. Those abilities would be useful to the user.
10. Feed returned images back to the agent. Images are typically returned to the user with product results, but there is no way for the agent to use them again. Therefore, if the user queries about a product returned, the agent cannot use that image.