React AST Vector Store & Retrieval Pipeline

Overview

This project implements an end-to-end pipeline to:

- Parse React code into ASTs
- Embed them using **Qwen3 Embedding models**
- Store in **Weaviate VectorDB** with graph references for component dependencies
- Retrieve and reconstruct component source code along with its dependencies for downstream code generation tasks.

It uses **LangChain abstractions** for vector storage and retrieval chaining with a custom **Qwen LLM** for final React JSX generation.

Key Components

1. `QwenEmbedding` class

Custom embedding interface using `dengcao/Qwen3-Embedding-8B:Q8_0` model via local API endpoint.

2. `Qwenmodel` class

LLM wrapper for `qwen-2.5-32b-instruct-zlabs` model to generate React JSX code from ASTs.

3. WeaviateDB setup

Creates a collection `reactcode_snippet` with properties:

- `component_name`: Name of the React component
- `ast_code`: AST representation

- `org_code`: Original source code
- `uses_components`: Graph reference to other components used

4. `import_with_relationship()` function

- Embeds each AST
- Inserts into WeaviateDB
- Adds graph references for used components
- Tracks embedding time

5. `print_component_with_dependencies(component_name)` function

Retrieves a component and traverses its `uses_components` graph to fetch all dependency source codes.

6. Retriever & Query flow

Uses LangChain retriever with **similarity score threshold** to fetch the closest matching AST from the database.

7. The Training Data used to build the DataBase

Link: https://github.com/rafaelalmeidatk/twitter-fullstack-clone.git

8. The Test Data used to test the working model

Link: https://github.com/kundanhere/netflix-clone.git

9. To view and visualize the vector embeddings follow the below link:

link: https://projector.tensorflow.org/

Setup Instructions

Dependencies

- **Python packages**
  - weaviate

- requests
- langchain_weaviate
- langchain
- react_ast (custom module for AST parsing & normalisation)

Environment

Ensure:

- Local Weaviate DB is running
- Run the provided docker file in order to set Weaviate local host
- Link to create your own Docker File : https://docs.weaviate.io/deploy/installation-guides/docker-installation
- Qwen embedding API accessible at: http://ai-test-3.csez.zohocorpin.com:11435/api/embeddings
- Qwen chat completion API accessible at: http://infinity-sm4:8091/v1/chat/completions

Docker-File:

```
---
services:
  weaviate:
    command:
    - --host
    - 0.0.0.0
    - --port
    - '8080'
    - --scheme
    - http
    image: cr.weaviate.io/semitechnologies/weaviate:1.30.0
    ports:
    - 8080:8080
    - 50051:50051
```

```
    volumes:
    - weaviate_data:/var/lib/weaviate
    restart: on-failure:0
    environment:
      QUERY_DEFAULTS_LIMIT: 25
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      PERSISTENCE_DATA_PATH: '/var/lib/weaviate'
      DEFAULT_VECTORIZER_MODULE: 'none'
      ENABLE_MODULES: ''
      CLUSTER_HOSTNAME: 'node1'
volumes:
  weaviate_data:
...
```

WeaviateDB.py

1. Import Statements

```
import weaviate, requests, os, pprint
from weaviate.classes.config import Property, DataType, ReferenceProperty
from langchain_weaviate import WeaviateVectorStore
from langchain.embeddings.base import Embeddings
from react_ast import code_snippet, parser, normalize_ast
from langchain.chains.retrieval import create_retrieval_chain
from langchain.llms.base import LLM
from langchain_core.prompts import ChatPromptTemplate
from langchain.chains.combine_documents import create_stuff_documents_c
hain
from weaviate.classes.query import Filter, QueryReference
import time
```

Explanation

- **weaviate**: Client for interacting with Weaviate Vector DB.

- **requests**: For API calls to embedding & LLM endpoints.
- **pprint, os, time**: Utility modules for printing, environment, and timing.
- **langchain modules**: Used for integrating embeddings, vector store, retrieval chains, and prompts.
- **react_ast**: Custom module containing:
  - `code_snippet`: Dataset of React components
  - `parser` & `normalize_ast`: Functions to parse and normalise ASTs for embedding.

---

2. Qwen Embedding Class

```
class QwenEmbedding(Embeddings):
    def embed_documents(self, code):
        return [self.embedding(t) for t in code]
    def embed_query(self, query):
        return self.embedding(query)
    def embedding(self, text):
        body = {
            'model': 'dengcao/Qwen3-Embedding-8B:Q8_0',
            'prompt': text
        }
        headers = {'content-type': 'application/json'}
        response = requests.post('http://ai-test-3.csez.zohocorpin.com:11435/ap
i/embeddings',
                                 headers=headers,
                                 json=body)
        return response.json()['embedding']

embedding = QwenEmbedding()
```

Explanation

Defines a custom **embedding interface** using Qwen 8B embedding model:

- **embed_documents**: Embeds a list of code snippets.

- **embed_query**: Embeds a single query.
- **embedding**: API call to Qwen embedding endpoint, returns the vector representation.

---

3. Qwen LLM Class

```
class Qwenmodel(LLM):
    def _call(self, prompt: str, stop=None):
        body = {
            "model": "qwen-2.5-32b-instruct-zlabs",
            "messages": [
                {"role": "system", "content": "You are a helpful code generator. Conv
ert the retrieved AST to a working React JSX code snippet."},
                {"role": "user", "content": prompt}
            ]
        }
        headers = {'content-type': 'application/json'}
        response = requests.post("http://infinity-sm4:8091/v1/chat/completions",
                                 headers=headers,
                                 json=body)
        output = response.json()
        return output['choices'][0]['message']['content']

    @property
    def _llm_type(self) -> str:
        return "qwen-custom"

llm = Qwenmodel()
```

Explanation

Defines a custom **LLM interface** to the Qwen 32B instruct model:

- Sends system + user prompts for code generation.
- Returns the **generated React JSX code** from the model.

4. Weaviate Connection and Collection Setup

```
client = weaviate.connect_to_local()
collection_name = 'reactcode_snippet'

client.collections.delete(collection_name)
print('collection deleted')

if collection_name not in client.collections.list_all():
    client.collections.create(
        name=collection_name,
        description="AST of react code",
        vectorizer_config=None,
        properties=[
            Property(name='component_name', data_type=DataType.TEXT),
            Property(name='ast_code', data_type=DataType.TEXT),
            Property(name='org_code', data_type=DataType.TEXT),
        ]
    )
    print("Collection created.")
else:
    print("Collection already exists.")

collection = client.collections.get(collection_name)
```

Explanation

- **Connects** to local Weaviate instance.
- Deletes existing collection `reactcode_snippet` to ensure a fresh setup.
- Creates a new collection with properties:
  - `component_name`: React component name
  - `ast_code`: AST representation (string)

- `org_code`: Original source code

---

5. Adding Graph References

```
collection.config.add_reference(
    ReferenceProperty(
        name='uses_components',
        target_collection=collection_name
    )
)
```

Explanation

Adds a **cross-reference property** `uses_components` to create **Graph relationships** between components that use each other. Enables dependency traversal.

---

6. Import Data with Relationships

```
elapsed_time = 0
def import_with_relationship():
    ids = {}
    for item in code_snippet:
        start_time = time.time()
        vec = embedding.embed_query(item['ast_code'])
        end_time = time.time()
        obj = collection.data.insert(
            properties={
                'component_name': item['component_name'],
                'org_code': item['org_code'],
                'ast_code': item['ast_code']
            },
            vector=vec
        )
        ids[item['component_name']] = str(obj)
```

```
        global elapsed_time
        elapsed_time += end_time - start_time
    print(f"Embedding took: {elapsed_time:.4f} seconds")

    for item in code_snippet:
        source_uuid = ids[item['component_name']]
        for dep in item.get('used_components', []):
            if dep in ids:
                collection.data.reference_add(
                    from_uuid=source_uuid,
                    from_property='uses_components',
                    to=ids[dep]
                )
import_with_relationship()
```

Explanation

- **Embeds** each AST code and inserts into Weaviate with its metadata.
- Tracks **embedding time** for performance metrics.
- Adds cross-references between components based on `used_components` to build dependency graph.

---

7. Function: Print Component with Dependencies

```
def print_component_with_dependencies(component_name):
    """Retrieve a component and all its dependencies' source code"""
    try:
        collection = client.collections.get("reactcode_snippet")
        response = collection.query.fetch_objects(
            limit=1,
            filters=Filter.by_property("component_name").equal(component_nam
e),
            return_properties=["component_name", "org_code"],
            return_references=[
```

```
            QueryReference(
                link_on="uses_components",
                return_properties=["component_name", "org_code"]
            )
        ]
    )
    if not response.objects:
        return f"Component {component_name} not found"

    main_component = response.objects[0]

    dependencies = []
    seen = set()
    queue = []

    if "uses_components" in main_component.references:
        for ref in main_component.references["uses_components"].objects:
            dep = ref.properties
            if dep["component_name"] not in seen:
                seen.add(dep["component_name"])
                dependencies.append(dep)
                queue.append(dep["component_name"])

    while queue:
        current_name = queue.pop(0)
        current_response = collection.query.fetch_objects(
            limit=1,
            filters=Filter.by_property("component_name").equal(current_name),
            return_properties=["component_name"],
            return_references=[
                QueryReference(
                    link_on="uses_components",
                    return_properties=["component_name", "org_code"]
                )
            ]
        )
```

```
            if not current_response.objects:
                continue

            current_component = current_response.objects[0]
            if "uses_components" in current_component.references:
                for ref in current_component.references["uses_components"].object
s:
                    dep = ref.properties
                    if dep["component_name"] not in seen:
                        seen.add(dep["component_name"])
                        dependencies.append(dep)
                        queue.append(dep["component_name"])

        output = []
        output.append(f"=== {main_component.properties['component_name']}
===\n")
        output.append(main_component.properties['org_code'])
        output.append("\n\n")

        for dep in dependencies:
            output.append(f"=== {dep['component_name']} (used by {component
_name}) ===\n")
            output.append(dep['org_code'])
            output.append("\n\n")

        return "".join(output)
    finally:
        client.close()
```

Explanation

- Retrieves a component by name from Weaviate.
- Performs **Breadth-First Search (BFS)** traversal over its `uses_components` graph to collect all dependencies.

- Returns formatted output with main component code followed by dependency codes.

---

8. Vector Store & Retriever Setup

```
vectordb = WeaviateVectorStore(client=client, index_name=collection_name, t
ext_key='ast_code', embedding=embedding)
retriever = vectordb.as_retriever(search_type='similarity_score_threshold', sea
rch_kwargs={'score_threshold':0.5,'k':2})
```

Explanation

- Wraps Weaviate collection as a **LangChain VectorStore** for retrieval.
- Configures retriever to perform semantic search with:
  - **score_threshold**: 0.5
  - **k**: Top 2 results

---

9. User Query & Parsing

```
user_query = input('Ask Something:')

def parse_query(user_query):
    try:
        tree = parser.parse(user_query.encode('utf-8'))
        root = tree.root_node
        if root.child_count == 0 or 'ERROR' in root.sexp():
            print("Input code is incomplete or invalid JS code.")
            return None
        normalized_user_query = normalize_ast(root)
        return normalized_user_query
    except Exception:
        return None
```

```
ast_query = parse_query(user_query)
```

Explanation

- Takes user input for a code snippet.
- Parses and normalises it into AST form for embedding and retrieval.

---

10. Retrieve & Display Results

```
docs = retriever.invoke(user_query)
print(docs)
retrieve_start_time = time.time()
if docs:
    component_name = docs[0].metadata.get('component_name')
    print(component_name)
    print(len(docs))
    result = print_component_with_dependencies(component_name)
    print(result)
else:
    print('No relevant Data found')
retrieve_end_time = time.time()
elapsed_time_retrieve = retrieve_end_time - retrieve_start_time
print(f"Embedding took: {elapsed_time_retrieve:.4f} seconds")
client.close()
```

Explanation

- Uses retriever to find matching documents based on user query .
- Prints the component name and total retrieved documents.
- Calls `print_component_with_dependencies` to display the component and its full dependency tree.
- Tracks retrieval time for performance monitoring.

React_ast.py

1. Import Statements

```
from tree_sitter import Language, Parser
import os, pprint
from ast_relationship import import_map, parse_component_name
```

Explanation

- **tree_sitter**: Used for parsing JavaScript/TypeScript code into ASTs.
- **os, pprint**: Filesystem traversal and pretty printing for debugging.
- **ast_relationship**: Custom module containing:
  - `import_map`: Generates a mapping of component imports and usage relationships.
  - `parse_component_name`: Extracts component name from the code.

---

2. Initialize Import Map

```
base_dir = 'D:/downloads1/vectordb_ast/react_data/components'
imap = import_map(base_dir)
```

Explanation

- **base_dir**: Root directory containing React components. Please change the location according to your base directory
- **imap**: Stores mappings of each component to its dependencies and usage, used later for setting relationships in the dataset.

---

3. Compile Tree-sitter JavaScript Grammar

```
Language.build_library(
    'build/my-languages.so',  # Output shared library
    [
        './tree-sitter-javascript'  # Path to grammar
    ]
)
```

Explanation

- Builds a shared library for JavaScript grammar parser using **tree-sitter**.
- Required **only once** unless grammar is updated.

---

4. Load Compiled Language & Initialize Parser

```
JS_LANGUAGE = Language('build/my-languages.so', 'javascript')
parser = Parser()
parser.set_language(JS_LANGUAGE)
```

Explanation

- Loads the compiled JavaScript grammar.
- Sets parser language to **JavaScript** for parsing React code files.

---

5. Function: normalize_ast

```
def normalize_ast(node):
    # if node.type == "identifier":
    #     return "var"
    # elif node.type == "string":
    #     return "str"
    if node.child_count == 0:
        return node.text.decode("utf-8")
    else:
```

```
        return f"({node.type} {' '.join([normalize_ast(child) for child in node.childr
en])})"
```

Explanation

- Recursively converts AST nodes to a normalised string format.
- **Commented lines** indicate an earlier approach to mask all identifiers as `var` and strings as `str`, which was disabled to retain actual names for precise retrieval.
- Returns either:
  - Terminal node text, or
  - A string representation with node type and children.

---

6. Prepare Code Snippet Dataset

```
code_snippet = []

for folder_path, dirs, files in os.walk(r'D:/downloads1/vectordb_ast/react_data/
components'):
    for x in files:
        if x.endswith(('.js', '.jsx','.ts','.tsx')):
            file_path = os.path.join(folder_path, x)
            with open(file_path, 'r', encoding='utf-8') as file:
                code = file.read()
                ast_code = parser.parse(code.encode('utf-8'))
                root_node = ast_code.root_node
                normalized_ast_string = normalize_ast(root_node)
                comp_name = parse_component_name(code)
                used = imap[comp_name]['used'] if comp_name in imap else []
                code_snippet.append({
                    'component_name': comp_name,
                    'ast_code': normalized_ast_string,
                    'org_code': code,
```

```
            'used_components': used
        })
```

Explanation

- Traverses **all React component files** under the specified base directory.
- For each file:
  - Reads the source code.
  - Parses it into an AST using Tree-sitter.
  - Normalises the AST into a string representation.
  - Extracts the component name.
  - Fetches its used components from `import_map`.
- Appends a dictionary to `code_snippet` containing:
  - **component_name**: Name of the component.
  - **ast_code**: Normalised AST string.
  - **org_code**: Original source code.
  - **used_components**: List of other components used/imported by this component.

---

7. Output Check (Commented)

```
# pprint.pprint(code_snippet)
```

Explanation

- Uncomment this line to **print and inspect** the prepared dataset for verification before database insertion.

---

ast_relationship.py:

---

1. Import Statements

```
import os, re
```

Explanation

- **os**: For filesystem traversal.
- **re**: For regular expression matching in component parsing.

---

2. Function: parse_component_name

```
def parse_component_name(code):
    match = re.search(r'(function|class)\s+(\w+)\s*\(', code)
    if match:
        return match.group(2)
    match = re.search(r'export\s+default\s+(\w+)', code)
    if match:
        return match.group(1)
    return None
```

Explanation

- **Purpose**: Extracts the component name from the source code.
- **Approach**:
  - Checks for function or class declarations (e.g. `function MyComp(` or `class MyComp {`).
  - Checks for `export default MyComp` syntax.
- **Returns**:
  - The parsed component name as a string, or `None` if not found.

---

3. Function: code_parser

```
def code_parser(code):
    imports = {}
    used_components = set()
    lines = code.split('\n')
    for line in lines:
        line = line.strip()
        if line.startswith('import'):
            parts = line.split('from')
            if len(parts) == 2:
                name = parts[0].replace('import', '').strip()
                path = parts[1].strip().strip(';').strip('"').strip("'")
                name = name.replace('{', '').replace('}', '').strip()
                imports[name] = path
    for x in imports.keys():
        if x in code:
            used_components.add(x)
    return imports, used_components
```

Explanation

- **Purpose**:
  - Parses JavaScript import statements.
  - Extracts all component names imported in the code.
  - Identifies which imported components are actually used.
- **Returns**:
  - A dictionary of `imports` mapping names to paths.
  - A set of `used_components` containing names of components referenced in the code.

---

4. Function: import_map

```
def import_map(base_dir):
    import_map = {}
    for filedir, file_path, filenames in os.walk(base_dir):
        for filename in filenames:
            if filename.endswith(('.js','.jsx','.ts','.tsx')):
                file_path = os.path.join(filedir, filename)
                with open(file_path, 'r', encoding='utf-8') as f:
                    code = f.read()
                imports, used_components = code_parser(code)
                exported_name = parse_component_name(code)
                name_without_ext = os.path.splitext(filename)[0]
                import_map[name_without_ext] = {
                    'imports': imports,
                    'used': used_components,
                    'path': file_path,
                    'exported': exported_name
                }
    return import_map
```

Explanation

- **Purpose**: Traverses a base directory to build a mapping of each file to:
  - Its imported components
  - Components used within
  - File path
  - Exported component name
- **Process**:
  - Walks through all `.js` , `.jsx` , `.ts` , `.tsx` files in the directory.
  - Reads each file's content.
  - Parses import statements and used components using `code_parser`.
  - Extracts the exported component name using `parse_component_name`.

    - Stores this data in a dictionary keyed by filename (without extension).
- **Returns**: `import_map` dictionary containing metadata for all components.

---

5. Module Summary

This module is a **utility parser** for:

- Generating a comprehensive mapping of React components, their imports, and usage relationships.
- Used in pipeline to establish **graph references** for dependency-based retrieval.

VectorDB Tasks and Learnings:

Setting up a Vector Database

My initial task involved setting up a vector database and performing simple queries to familiarise myself with vector search fundamentals. This laid the groundwork for upcoming tasks requiring efficient semantic retrieval.

Storing React Code Snippets as ASTs for Retrieval

The next objective was to store React code snippets in the database such that querying with any piece of React code would retrieve the entire corresponding document. To achieve this, I decided to convert the React code into its Abstract Syntax Tree (AST) representation, as ASTs preserve structural and logical information, making retrieval more robust.

Problem Faced

Initially, I faced difficulties in retrieving the entire document accurately. The retrieval failed because I was not using a **code embedding model**, which meant the semantic meaning of the code was not captured effectively.

Solution

To address this, I modified the vectorization step to embed both the AST code and the original source code together. By combining them, the database could match the user query against both representations, significantly improving retrieval accuracy.

Retrieving Component Dependencies

Once retrieval of a single component worked, my next goal was to extend the functionality to retrieve **not only the queried component but also all its dependent components**. This required establishing relationships between the stored components.

Problem Faced

In Weaviate, each inserted data item is treated as an object. Traditional RAG (Retrieval-Augmented Generation) approaches were insufficient here, as they do not support direct relational queries between objects. I needed a way to model the dependency relationships between React components.

Solution

I explored **GraphRAG**, which allows traversing linked data objects to retrieve related context. Weaviate provides cross-reference properties for creating such relationships. I implemented these cross-references to link each component with its dependencies. You can read more about Weaviate cross-references here.

Additionally, I wrote a **graph traversal function** to print the entire dependency tree alongside the main component, ensuring the code generation system has all necessary context.

Improving AST Conversion Precision

During the AST conversion step, I encountered a critical bug. Initially, my AST normalisation logic replaced all identifiers and string literals with generic tokens like `"var"` and `"str"`, under the assumption that the LLM would only require the logic structure to regenerate the code.

Problem Faced

However, this approach reduced retrieval precision significantly, as the unique identifiers are crucial for accurate matching.

Solution

I resolved this by commenting out the replacement logic in the `react_ast.py` file. After this fix, retrieval results became **highly precise**, as the AST retained the unique identifiers needed for semantic matching.

Integrating Qwen Code Embedding Model

Despite improvements, further enhancement was possible because I was still embedding both AST and original code. After gaining access to the **Qwen 3 8B embedding model**, which is specifically optimised for multilingual text and code embeddings, I modified the insertion step to embed **only the code itself**.

Outcome

With the Qwen code embedding model integrated, semantic search using partial code snippets achieved **high precision retrieval**, fulfilling the initial task requirement effectively.

Known Limitation & Outstanding Bug

During testing, an unresolved bug was identified:

- When printing a main component and its dependencies, if the imported component name does not exactly match the registered component name in the database, retrieval fails.

Example

```
import Watch from 'components/WatchPage'
```

Here, the used component name is `Watch`, while the stored component name is `WatchPage`. Although logically the same, the system requires an exact match between the dependency name and the component name.

Next Steps

This bug is partially resolved and requires further testing with different data. Now it successfully matches its dependencies but it might not perfrom well in future because the logic used behing this bug is FUZZY LOGIC MATCH. so it will work for the above example but when it comes to import name as ‘xyz‘ or any other irrelevant name to the component name it might not work . It needs further fixing

The bug was resolved using the following code snippet:

```
import difflib

def import_with_relationship():
    ids = {}
    for item in code_snippet:
        vec = embedding.embed_query(item['ast_code'])
        obj = collection.data.insert(
            properties={
                'component_name': item['component_name'],
                'org_code': item['org_code'],
                'ast_code': item['ast_code']
            },
            vector=vec
        )
        ids[item['component_name']] = str(obj)

    # Filter out any None keys from component_names
    component_names = [name for name in ids.keys() if name]

    for item in code_snippet:
        source_uuid = ids[item['component_name']]
        for dep in item.get('used_components', []):
            if not dep:
                continue  # skip None or empty used components

            target_uuid = None
```

```
            # First try exact match
            if dep in ids:
                target_uuid = ids[dep]
            else:
                # Perform partial/fuzzy match using difflib
                # Ensure dep is a string
                if isinstance(dep, str):
                    matches = difflib.get_close_matches(dep, component_names, n=1, c
                    if matches:
                        target_uuid = ids[matches[0]]

            if target_uuid:
                collection.data.reference_add(
                    from_uuid=source_uuid,
                    from_property='uses_components',
                    to=target_uuid
                )
```

This is the original code snippet that produced the bug If you want you can look into it :

```
def import_with_relationship():
    ids={}
    # code_snippet_unique = {item['component_name']: item for item in code_snip
    for item in code_snippet:
        # combined_text = f"{item['ast_code']} {item['org_code']}"
        start_time=time.time()
        vec=embedding.embed_query(item['ast_code'])
        end_time=time.time()
        obj=collection.data.insert(
        properties={
            'component_name':item['component_name'],
            'org_code':item['org_code'],
            'ast_code':item['ast_code']
```

```
        },
        vector=vec
        )
        ids[item['component_name']] = str(obj)
        _time=end_time-start_time
        global elapsed_time
        elapsed_time+=_time
    print(f"Embedding took: {elapsed_time:.4f} seconds")

    for item in code_snippet:
        source_uuid=ids[item['component_name']]
        for dep in item.get('used_components', []):
            if dep in ids:
                collection.data.reference_add(
                    from_uuid=source_uuid,
                    from_property='uses_components',
                    to=ids[dep]
                )
                # collection.data.reference_add(
                #     from_uuid=ids[dep],
                #     from_property='used_by',
                #     to=source_uuid
                # )

import_with_relationship()
```