Code Comprehension Assistant using - GenAI

Course Project on
DS246 - Gen AI and Agentic AI in practice

Ila Ananta (24956)
Vanshika Jindal (24721)
Abhishek Ayyappa Bhat B (24900)
Ankit Negi (24877)
Ashik Raveendran (24205)

Code Comprehension Assistant using GenAI

Unify CPG, graph retrieval, and LLM reasoning for semantic code understanding

Code Property Graph (CPG)

Unified semantic graph combining syntax, control flow, and data dependencies.

Integration & Tooling

Bridges to existing codebases and developer tools for practical workflow adoption.

Developer Interface

Natural language queries, interactive code exploration, and explainable answers.

Graph-based Retrieval

Fast, precise retrieval of relevant code regions via graph queries and similarity.

LLM Reasoning

Context-aware natural language answers, summarization, and navigation guidance.

Problem Statement

To develop an AI system enabling developers to interact with large codebases via natural language using CPGs, graph retrieval, and LLM reasoning

Goal: Develop an AI system allowing developers to interact with large codebases via natural language

Approach: Leverage Code Property Graphs (CPGs) to model code structure and semantics

Retrieval: Use graph-based retrieval to surface relevant code regions and structural relations

Reasoning: Combine retrieved graph context with LLM reasoning for semantic explanations and tasks

Benefits: Faster comprehension, fewer errors, and robust support across large, evolving codebases

Motivation: Faster, Safer Code Exploration

Reduce onboarding time and debugging friction by surfacing code structure and semantics

Developers face weeks-long learning curves on unfamiliar projects due to missing structural awareness

Difficulty tracing **function calls**, **control flow**, and **data dependencies** across large codebases

Documentation often **incomplete** or **outdated**, blocking accurate troubleshooting

Current tools lack integrated **code graphs**, **semantic retrieval**, and LLM reasoning

This project unifies those capabilities to boost **productivity** and **accuracy** in maintenance and debugging

GraphRAG Architecture

GraphRAG is an enhanced Retrieval-Augmented Generation (RAG) framework that incorporates graph structure instead of retrieving only text chunks. It retrieves nodes, edges, and relationships relevant to the query to provide better reasoning and context for LLMs.

1

Code as Graph

Retrieve nodes, edges, and relationships from the code/property graph rather than only text chunks.

2

Contextual Assembly

Assemble relevant graph substructures to provide richer context and explicit connections for the query.

3

LLM Reasoning

Feed structured graph context into the LLM to enable improved chain-of-thought and more informed responses.

4

Improved Responses

Generate answers grounded in graph relationships for more accurate, explainable outputs.

Tools Used — Scalable Semantic Code Retrieval

Key technologies powering CPG generation, embedding, graph processing, storage, and UI

- **Joern** for generating the unified Code Property Graph (CPG)
- ***Python*** for parsing, embedding, graph processing, and retrieval pipeline
- **ChromaDB** for vector storage and similarity search
- **Nomic-Embed** and **GraphCodeBERT** to generate code embeddings
- LLMs **Qwen2.5-Coder** and **Llama** for code explanation
- **NetworkX** for graph structure handling
- **Streamlit** to build a user-friendly frontend interface
- **Conda** to manage virtual environments and dependencies

Code Property Graph (CPG)

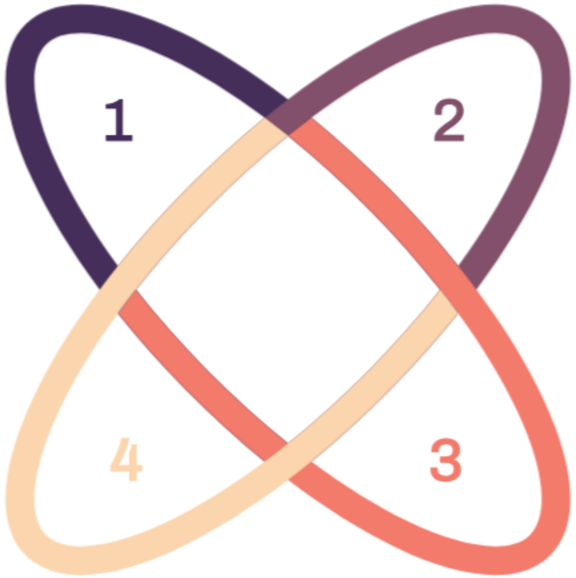
Unified graph merging **AST**, **CFG**, and **PDG** to enrich code context for better LLM reasoning

AST — Abstract Syntax Tree

Captures code structure and syntax nodes

CFG — Control Flow Graph

Represents execution order and control edges



PDG — Program Dependence Graph

Shows data dependencies between operations

CPG — Combined Graph

Integrates syntax, control, and data flow for richer context

Core Architecture — Two Analysis Methods

Our system combines Two Analysis Methods

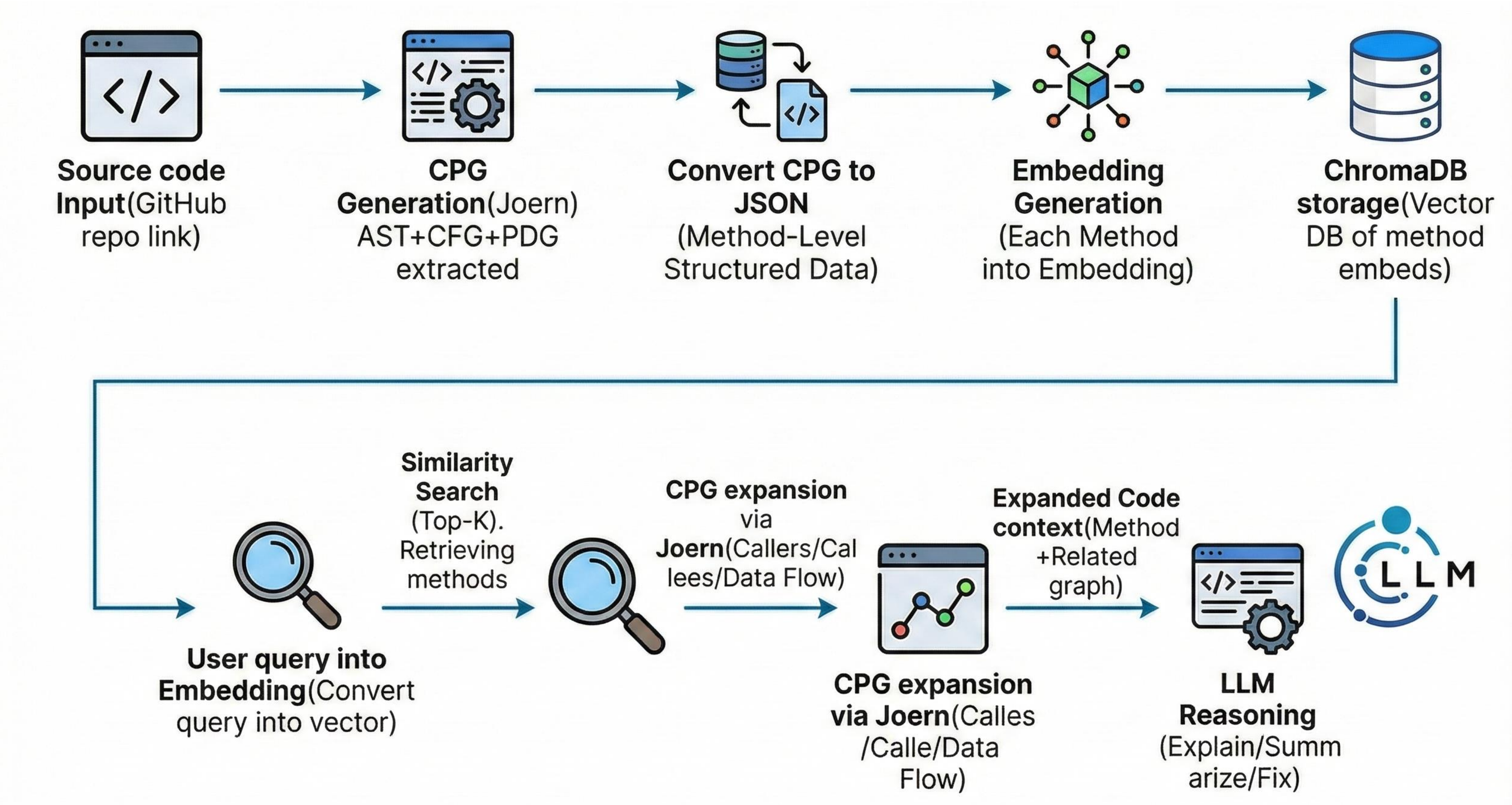
Method 1. Query Specific Understanding

- User query-Embedding
- Similarity Search- Top K-relevant Methods
- CPG expansion via Joern
- LLM explains:
  - What code does
  - Flow of Logic
  - Function Specific Behavior
  - Input/Output
  - Executution Steps

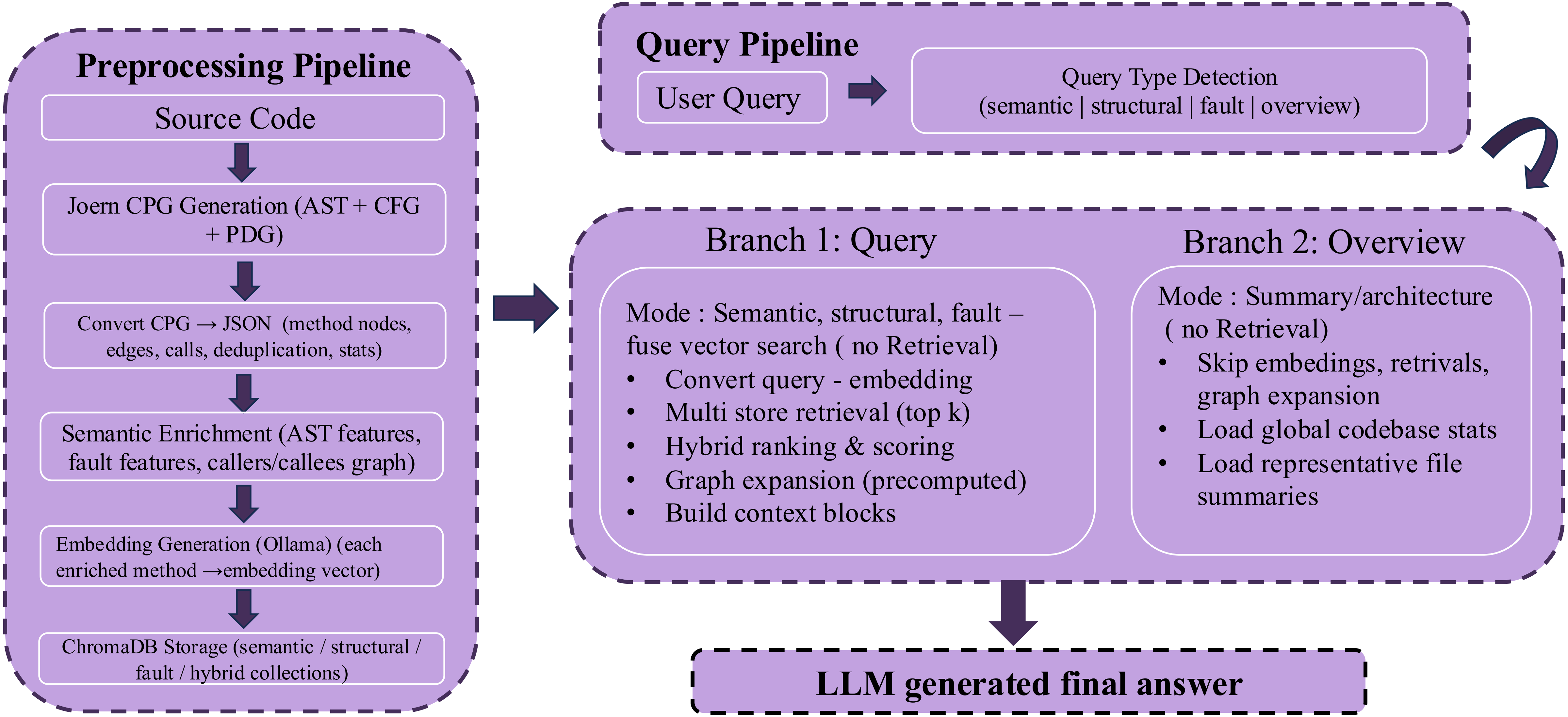
Method 2. Code Overview

- User Query → Overview/Summary/Structure
- Codebase Statistics (Total files, Total methods, Total lines, Largest modules, Largest functions)
- Overview Mode Activation: Detects “overview”, “summarize”, “architecture”
  Switches from retrieval → global summary
- LLM-only processing
- LLM explains :
  1. Purpose of codebase
  2. Main components
  3. Module responsibilities
  4. High-level workflow
  5. Overall architecture
  6. Fault detection

Query-Specific Code Understanding - Method 1



Fault detection and overview Code Understanding - Method 2



CPG Generation & Graph Representation Challenges

Key technical blockers that degraded performance and retrieval accuracy

CPG Generation Issues : Joern Produced Very Large CPG Files For Medium-Sized Codebases - Exporting CPG To DOT/JSON Caused Slow Processing .
Some Nodes/Edges Were Incomplete Or Not Linked As Expected. AST / CFG / PDG Extraction Was Sometimes Inconsistent

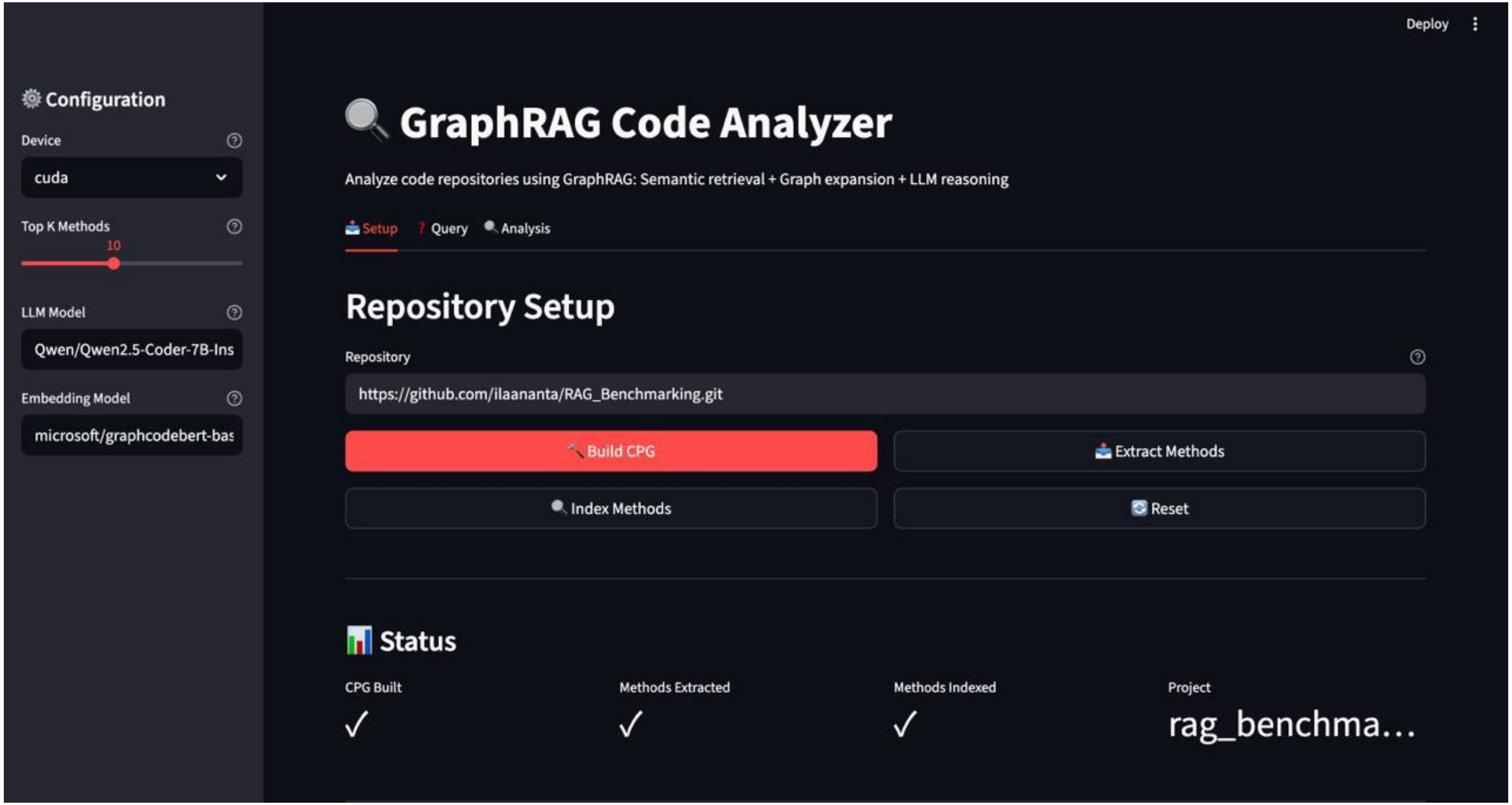
Graph Representation Challenges : JSON Output For Large Graphs Became Heavy To Parse - Reconstructing Graphs In Python Was Slow.
Needed To Decide Which Representation Preserved Structure Best

Embedding & Retrieval Problems : Some Code Methods Produced Poor Embeddings (Too Short Or Boilerplate) . Vector DB Similarity Search Sometimes Returned Irrelevant Methods .
Query Embeddings Did Not Always Match Code Semantics

Graph Expansion Issues : Joern Queries Occasionally Returned Empty Caller/Callee Sets, Missing Data-Flow Relationships Were Observed, Duplicate Nodes Appeared In Expanded Graphs.
Expanding Neighbors For Multiple Methods Became Slow

LLM Reasoning Limitations:
Some Explanations Lacked Depth Due To Incomplete Graph Context
LLM Hallucinated When Graph Expansion Failed
Needed Careful Formatting Of Context For Better Answers

Results — MedSAM Evaluation Highlights



Results — MedSAM Evaluation Highlights

Enter your question

how segmentation is done using medsam model?

Generate Answer

Answer

Segmentation with the MedSAM model involves loading the pre-trained SAM model, applying it to an input image to generate masks, and then processing these masks to obtain the final segmented result. The process typically includes initializing the model, preprocessing the image, running inference to predict masks, post-processing to refine the segmentation, and finally saving or displaying the segmented output.

Enter your question

what does this code base do?

Generate Answer

Answer

This code base contains multiple scripts designed for various medical image segmentation tasks. It includes functionalities for loading models, processing images, and saving segmentation results. Scripts like `MedSAM_Inference.py` handle inference using the Segment Anything Model (SAM) for different dimensions (2D and 3D), while others like `infer_nnunet_2D.py`, `infer_nnunet_3D.py`, and `infer_deeplabv3_res50_2D.py` are tailored for specific models such as nnU-Net and DeepLabV3+. Additionally, there are utility scripts for handling data directories, initializing models, and performing operations like finding the largest connected component in segmentations. The overall purpose is to provide tools for researchers and practitioners to perform efficient and accurate medical image segmentation across various applications and datasets.

Results — MedSAM Evaluation Highlights

Enter your question

How does the code distinguish between MedSAM checkpoints and the original SAM checkpoints, and where does the conversion between formats happen?

🚀 Generate Answer

Answer

Assistant: To understand how the code distinguishes between MedSAM checkpoints and the original SAM checkpoints and where the conversion between formats happens, let's analyze the provided code and relationships step-by-step: 1. **Checkpoint Loading**: Both MedSAM and original SAM checkpoints can be loaded using the `segment_anything.sam_model_registry` function. This function takes a model name and loads the corresponding pre-trained model. The key difference lies in the specific model configurations and architectures used. 2. **Model Registration**: - For MedSAM, the model registry might include a specific entry for MedSAM models. This could involve custom model classes or configuration files that differentiate MedSAM from standard SAM models. - For standard SAM models, the registry would have entries specifically for SAM models. 3. **Conversion Between Formats**: - The conversion between MedSAM and original SAM checkpoints typically happens when a user wants to switch between these models for inference or training. This could be handled in scripts like `MedSAM_Inference.py` or similar inference scripts. - When a user specifies the checkpoint path, the script checks whether the path corresponds to a MedSAM checkpoint or an original SAM checkpoint. Depending on the path, the script initializes the appropriate model. 4. **Script Examples**: - In `MedSAM_Inference.py`, there is no explicit distinction made between MedSAM and original SAM checkpoints in the provided code snippet. However, in other scripts (like those mentioned in the relationships section), the script might look for specific patterns or directories indicating the type of checkpoint. For instance, in the script `comparisons/SAM/infer_SAM_2D_npz.py`, the checkpoint path might be checked against known paths for MedSAM models. If the path matches, the script might load a MedSAM-specific model configuration. # Example pseudo-code to illustrate checkpoint loading based on path if 'medsam' in checkpoint_path: model = sam_model_registry[medsam] model = sam_model_registry[default_sam] model.load_state_dict(torch.load(checkpoint_path)) In summary, the code distinguishes between MedSAM and original SAM checkpoints through specific model registrations and potentially through path-based logic in scripts that handle checkpoint loading. The actual conversion between formats occurs within these scripts when initializing the model based on the provided checkpoint path.

Enter your question

Where in the training code (e.g. train_one_gpu.py or related modules) are the prompt encoder parameters excluded from optimization, and how is that implemented in PyTorch?

🚀 Generate Answer

Answer

In the training code, specifically within the `train_multi_gpus.py` module, the prompt encoder parameters are excluded from optimization. This is achieved by setting the `requires_grad` attribute of these parameters to `False`. By default, all model parameters have `requires_grad=True`, meaning they are included in the computation graph and their gradients will be computed during backpropagation. To exclude the prompt encoder parameters from being optimized, the code iterates through them and sets `requires_grad=False`. This tells PyTorch not to track operations on these parameters for gradient computation, effectively freezing them during training.

Results — MedSAM Evaluation Highlights

Enter your question

The README says pre_CT_MR.py performs soft-tissue windowing (L=40, W=400), max–min normalization, and resamples to 1024×1024 before saving .npy files. How does the actual code implement CT windowing?

🚀 Generate Answer

Answer

The actual code for implementing CT windowing in pre_CT_MR.py involves adjusting pixel values based on a specified window level (L=40) and window width (W=400). This process transforms the original CT image data so that only the soft tissue region within the defined window is emphasized, effectively highlighting structures of interest while suppressing other regions. After applying the windowing, the code performs max-min normalization to scale the pixel values between 0 and 1, ensuring that the data is properly normalized for further processing or analysis.

Enter your question

In train_multi_gpus.py, what distributed training backend is used (torch.distributed, DistributedDataParallel, etc.), and how are rank, world size, and device mapping handled?

🚀 Generate Answer

Answer

In `train_multi_gpus.py`, the distributed training backend used is `torch.distributed`. The script sets up multiple processes for parallel training using `torch.multiprocessing`'s `mp.spawn`. Each process has its own GPU assigned based on `ngpus_per_node`, which is determined by `torch.cuda.device_count()`. Rank and world size are managed within the `main_worker` function, which is called by `mp.spawn`. The rank of each process is passed as an argument to `main_worker`, and it is used to ensure that only one process writes to the log file at a time. World size is implicitly defined by the number of GPUs available (`ngpus_per_node`). Device mapping is handled by specifying the device inside the `main_worker` function where models and data are moved to the appropriate GPU using `.to(device)`. Overall, the script leverages PyTorch's distributed training capabilities to scale across multiple GPUs efficiently.

Enter your question

How does the sam_model_registry (imported in MedSAM_Inference.py) determine which backbone ("vit_b", etc.) to construct, and where in the segment_anything subpackage is this registry implemented?

🚀 Generate Answer

Answer

The sam_model_registry in MedSAM_Inference.py determines which backbone (such as vit_b) to construct based on a configuration passed during the initialization or invocation of SamPredictor from the segment_anything module. This registry maps specific configurations to their corresponding backbone implementations within the segment_anything.subpackage.modeling.sam module.

Sample Prompts used in Code Analysis

```
Answer:(base) nidhi@user-MS-7D98:~/Ila/ai/prompts$ cat query_prompt.txt
You are a code analysis assistant.

Question: What is the code doing?

Below is relevant code and relationship information to help answer this question.

================================================================================
QUESTION
================================================================================

What is the code doing?

================================================================================
RELEVANT CODE METHODS
================================================================================
(Methods are ordered by semantic relevance to your question)
IMPORTANT: Review ALL methods below - even if a method appears later in the list, it may still be highly relevant to your question.

--- Method 1: <module> ---
File: MedSAM_Inference.py
Line: 3

Code:
File: MedSAM_Inference.py
Code:
"""
usage example:
python MedSAM_Inference.py -i assets/img_demo.png -o ./ --box "[95,255,190,350]"

"""

# %% load environment
import numpy as np
import matplotlib.pyplot as plt
import os

join = os.path.join
import torch
from segment_anything import sam_model_registry
from skimage import io, transform
import torch.nn.functional as F
import argparse


# visualization functions
# source: https://github.com/facebookresearch/segment-anything/blob/main/notebooks/predictor_example.ipynb
# change color to avoid red and green
Calls: <operator>.assignment, add_argument, eval, imsave, show_box, show_mask, show, <operator>.fieldAccess, __init__, parse_args

--- Method 2: <module> ---
File: segment_anything/predictor.py
Line: 8
```

Sample Prompts used in Code Analysis

```
================================================================================
CODE RELATIONSHIPS
================================================================================
IMPORTANT: The relationships below show which methods call the methods above.
Pay special attention to class names and file paths in the 'Called by' section —
they often reveal the main components and algorithms in the codebase.

--- Method 1: <module> ---
Calls: add_argument, eval, imsave, show_box, show_mask, show, __init__, parse_args, to, imread

--- Method 2: <module> ---

--- Method 3: getLargestCC ---
Called by: extensions/seg_3dnii_sparse_marker/medsam_infer_3Dbox_adrenal.py:<module>
Calls: label, astype, argmax, bincount

--- Method 4: <fakeNew> ---
Called by: comparisons/DeepLabV3+/train_deeplabv3_res50.py:<module>.NpyDataset.<metaClassCallHandler>, extensions/point_prompt/train_point_prompt.py:<module>.NpyDataset.<metaClassCallHandler>, extensions/poin
t_prompt/train_point_prompt.py:<module>.MedSAM.<metaClassCallHandler>, extensions/text_prompt/train_text_prompt.py:<module>.NpyDataset.<metaClassCallHandler>, extensions/text_prompt/train_text_prompt.py:<modu
le>.TextPromptEncoder.<metaClassCallHandler>, extensions/text_prompt/train_text_prompt.py:<module>.MedSAM.<metaClassCallHandler>, gui.py:<module>.Window.<metaClassCallHandler>, segment_anything/automatic_mask
_generator.py:<module>.SamAutomaticMaskGenerator.<metaClassCallHandler>, segment_anything/modeling/common.py:<module>.MLPBlock.<metaClassCallHandler>, segment_anything/modeling/common.py:<module>.LayerNorm2d.
<metaClassCallHandler>
Calls: __init__

--- Method 5: dice_coefficient ---
Called by: comparisons/DeepLabV3+/infer_deeplabv3_res50_2D.py:<module>, comparisons/DeepLabV3+/infer_deeplabv3_res50_3D.py:<module>, comparisons/nnU-Net/infer_nnunet_2D.py:<module>, comparisons/nnU-Net/infer_
nnunet_3D.py:<module>, utils/SurfaceDice.py:<module>
Calls: sum

--- Method 6: _get_device ---
Called by: extensions/text_prompt/train_text_prompt.py:<module>.TextPromptEncoder.forward, segment_anything/modeling/prompt_encoder.py:<module>.PromptEncoder.<body>, segment_anything/modeling/prompt_encoder.p
y:<module>.PromptEncoder.forward, segment_anything/modeling/prompt_encoder.py:<module>.PromptEncoder._get_device<metaClassAdapter>

--- Method 7: _pe_encoding ---
Called by: segment_anything/modeling/prompt_encoder.py:<module>.PositionEmbeddingRandom.<body>, segment_anything/modeling/prompt_encoder.py:<module>.PositionEmbeddingRandom.forward, segment_anything/modeling/
prompt_encoder.py:<module>.PositionEmbeddingRandom.forward_with_coords, segment_anything/modeling/prompt_encoder.py:<module>.PositionEmbeddingRandom._pe_encoding<metaClassAdapter>, segment_anything/utils/onnx
.py:<module>.SamOnnxModel._embed_points
Calls: cat, sin, cos

--- Method 8: dice_coefficient ---
Called by: comparisons/DeepLabV3+/infer_deeplabv3_res50_2D.py:<module>, comparisons/DeepLabV3+/infer_deeplabv3_res50_3D.py:<module>, comparisons/nnU-Net/infer_nnunet_2D.py:<module>, comparisons/nnU-Net/infer_
nnunet_3D.py:<module>, utils/SurfaceDice.py:<module>
Calls: sum

================================================================================
INSTRUCTIONS
================================================================================
You are a code analysis assistant. Based on the code methods and relationships shown above, answer the following question in natural language.

IMPORTANT: Write a natural language answer. Do NOT copy or repeat:
- Method names with '--- Method X: name ---' format
- 'Called by:' or 'Calls:' lines
- Code fragments or variable names

Instead, explain what the functions do in plain English.

Question: What is the code doing?
```

Future Scope: Security, UI, Scalability & Language Growth

Improve CPG Generation
Optimize Joern Export To Reduce Extremely Large CPG Sizes.
Explore Protobuf-Based CPG Extraction For Faster And Lossless Structure Retrieval.
Implement Incremental CPG Updates Instead Of Full Graph Regeneration.

Better Graph Expansion & Parsing
Build A Faster And More Consistent Parser For DOT/JSON Outputs.
Improve Caller–Callee & Data-Flow Extraction Reliability.
Cache Graph Neighborhoods To Avoid Repeated Joern Queries.

Enhanced Semantic Retrieval
Improving Embedding Generation
Fine-Tune Embedding-Generation Model (E.G., GraphCodeBERT / StarCoder Embeddings).

- Use another LLM to generate method explanations.
- Index the embeddings for these explanations to ChromaDB.

Stronger LLM Reasoning
Train Task-Specific LLM Prompts To Reduce Hallucinations When Graph Context Is Incomplete.
Add Structured Explanation Templates For Debugging, Data-Flow, And Security Analysis.
Explore Lightweight Local LLMs For Offline Code Analysis.

Future Scope: Security, UI, Scalability & Language Growth

Security & Vulnerability Detection

Extend The System To Detect :

Taint Flows

Insecure API Calls

Buffer Overflows / Data Leaks

Integrate Existing SAST Tools With GraphRAG Analysis.

Scalable System Integration

Improve Streamlit UI To Support :

Full-Repo Exploration

Function Dependency Heatmaps

Graph Visualizations (AST/CFG/PDG).

Deploy As A Cloud Service With Async Retrieval + Caching.

Multi-Language Support

Expand Beyond Python/C++ To Java, Rust, Golang Using Joern Or Similar CPG Generators.