Inside GraphRAG: Analyzing Microsoft's Innovative Framework for Knowledge Graph Processing

Calvin Ku · Follow
Published in Percena
11 min read · Jul 10, 2024

Listen Share More

Introduction

Microsoft recently open-sourced GraphRAG, a new Retrieval-Augmented Generation (RAG) system based on knowledge graphs. This framework leverages Large Language Models (LLMs) to extract structured data from unstructured text, build labeled knowledge graphs, and support various applications such as dataset question generation and summarized Q&A. A key feature of GraphRAG is its use of graph machine learning algorithms for semantic aggregation and hierarchical analysis of datasets. This enables it to answer relatively high-level abstract or summary questions, addressing a common shortcoming of conventional RAG systems.

Having followed this framework for some time, I recently spent several days studying its source code. This article presents my interpretation of GraphRAG's source code, combined with insights from previous technical documents. My goal is to provide a deeper understanding of its system architecture, key concepts, and core workflows.

The GraphRAG project source code analyzed in this article corresponds to commit ID a22003c302bf4ffeefec76a09533acaf114ae7bb, last updated on July 5, 2024.

Framework Overview

What problem does it solve (What & Why)?

Before delving into the code, let's briefly examine the goals and positioning of the GraphRAG project. In their paper, the authors identify an application scenario that conventional RAG systems struggle to handle:

> *"However, RAG fails on global questions directed at an entire text corpus, such as 'What are the main themes in the dataset?', since this is inherently a query-focused summarization (QFS) task, rather than an explicit retrieval task."*

These high-level summary questions, like "What are the themes of this dataset?", represent a challenge for traditional RAG systems. The authors argue that this scenario is essentially a Query-Focused Summarization (QFS) task, which cannot be solved by data retrieval alone.

> *"In contrast with related work that exploits the structured retrieval and traversal affordances of graph indexes (subsection 4.2), we focus on a previously unexplored quality of graphs in this context: their inherent modularity (Newman, 2006) and the ability of community detection algorithms to partition graphs into modular communities of closely-related nodes (e.g., Louvain, Blondel et al., 2008; Leiden, Traag et al., 2019). LLM-generated summaries of these community descriptions provide complete coverage of the underlying graph index and the input documents it represents. Query-focused summarization of an entire corpus is then made possible using a map-reduce approach: first using each community summary to answer the query independently and in parallel, then summarizing all relevant partial answers into a final global answer."*

The solution employs community detection algorithms (such as the Leiden algorithm) to divide the entire knowledge graph into modular communities containing highly related nodes. Large language models are then used to summarize these communities from bottom to top. Finally, a map-reduce approach implements QFS: each community first executes the query in parallel, and the results are summarized into a comprehensive global answer.

How is it implemented (How)?

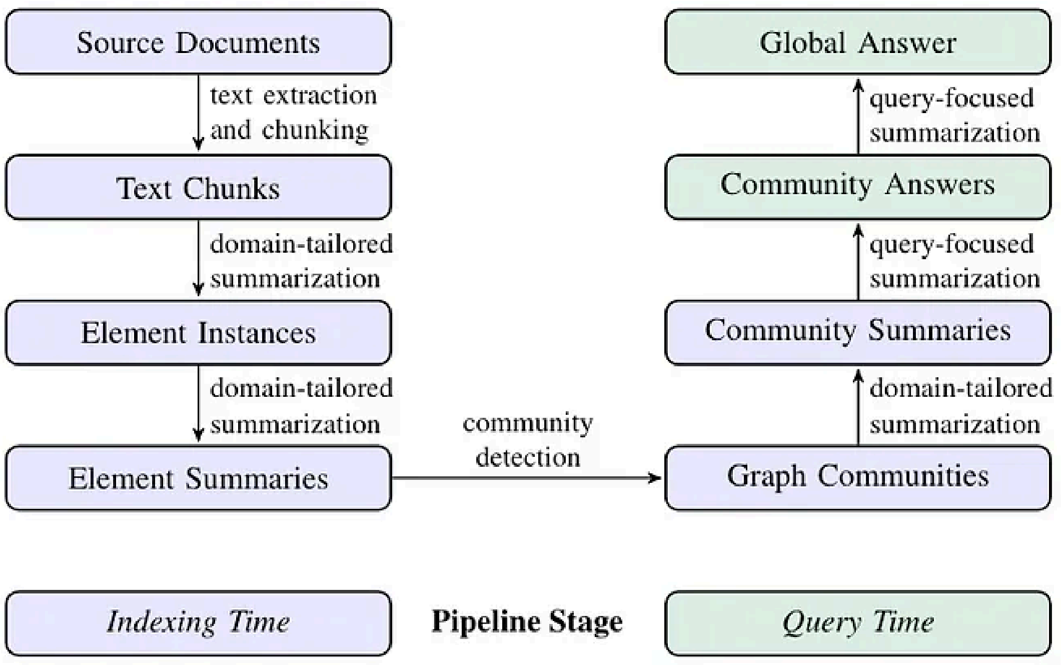


Figure 1: Graph RAG pipeline using an LLM-derived graph index of source document text. This index spans nodes (e.g., entities), edges (e.g., relationships), and covariates (e.g., claims) that have been detected, extracted, and summarized by LLM prompts tailored to the domain of the dataset. Community detection (e.g., Leiden, Traag et al., 2019) is used to partition the graph index into groups of elements (nodes, edges, covariates) that the LLM can summarize in parallel at both indexing time and query time. The "global answer" to a given query is produced using a final round of query-focused summarization over all community summaries reporting relevance to that query.

uses community detection technology to partition the entire knowledge graph and further summarizes using LLMs. For specific queries, it can summarize all related community summaries to generate a global answer.

Source Code Analysis

While the official documentation is quite comprehensive, delving into the source code helps us understand some implementation details. Let's examine the specific code implementation. The project source code structure is as follows:

```
.
├── cache
├── config
├── emit
├── graph
```

```
│   ├── embedding
│   ├── extractors
│   │   ├── claims
│   │   ├── community_reports
│   │   ├── graph
│   │   └── summarize
│   ├── utils
│   └── visualization
├── input
├── llm
├── progress
├── reporting
├── storage
├── text_splitting
├── utils
├── verbs
│   ├── covariates
│   │   └── extract_covariates
│   │       └── strategies
│   │           └── graph_intelligence
│   ├── entities
│   │   ├── extraction
│   │   │   └── strategies
│   │   │       └── graph_intelligence
│   │   └── summarize
│   │       └── strategies
│   │           └── graph_intelligence
│   ├── graph
│   │   ├── clustering
│   │   │   └── strategies
│   │   ├── embed
│   │   │   └── strategies
│   │   ├── layout
│   │   │   └── methods
│   │   ├── merge
│   │   └── report
│   │       └── strategies
│   │           └── graph_intelligence
│   ├── overrides
│   └── text
│       ├── chunk
│       │   └── strategies
│       ├── embed
│       │   └── strategies
│       ├── replace
│       └── translate
│           └── strategies
└── workflows
    └── v1
```

This file structure contains many documents worth careful study, which we'll explain in detail later in conjunction with the code.

Demo

Before diving into specific functions, let's first run the official demo. It's quite straightforward to get started; you can directly refer to the Get Started guide.

> *Warning: Although it's just a simple demo, the token consumption is not trivial. Despite having expectations and deleting more than half of the original document content in advance, it still cost us about $3 to run it completely. Running the official complete demo document is estimated to consume $5-$10.*

The actual runtime here is relatively slow, as the large model is going back and forth through the entire document. Some of the more important points are as follows:

```
├── cache
│   ├── community_reporting
│   │   ├── create_community_report-chat-v2-0d811a75c6decaf2b0dd7b9edff02389
│   │   ├── create_community_report-chat-v2-1205bcb6546a4379cf7ee841498e5bd4
│   │   ├── create_community_report-chat-v2-1445bd6d097492f734b06a09e579e639
│   │   ├── ...
│   ├── entity_extraction
│   │   ├── chat-010c37f5f6dedff6bd4f1f550867e4ee
│   │   ├── chat-017a1f05c2a23f74212fd9caa4fb7936
│   │   ├── chat-09095013f2caa58755e8a2d87eb66fc1
│   │   ├── ...
│   ├── summarize_descriptions
│   │   ├── summarize-chat-v2-00e335e395c5ae2355ef3185793b440d
│   │   ├── summarize-chat-v2-01c2694ab82c62924080f85e8253bb0a
│   │   ├── summarize-chat-v2-03acd7bc38cf2fb24b77f69b016a288a
│   │   ├── ...
│   └── text_embedding
│       ├── embedding-07cb902a76a26b6f98ca44c17157f47f
│       ├── embedding-3e0be6bffd1c1ac6a091f5264858a2a1
│       ├── ...
├── input
│   └── book.txt
├── output
│   └── 20240705-142536
│       ├── artifacts
│       │   ├── create_base_documents.parquet
│       │   ├── create_base_entity_graph.parquet
│       │   ├── create_base_extracted_entities.parquet
│       │   ├── create_base_text_units.parquet
│       │   ├── create_final_communities.parquet
│       │   ├── create_final_community_reports.parquet
│       │   ├── create_final_documents.parquet
│         │    ├── create_final_entities.parquet
│         │    ├── create_final_nodes.parquet
│         │    ├── create_final_relationships.parquet
│         │    ├── create_final_text_units.parquet
│         │    ├── create_summarized_entities.parquet
│         │    ├── join_text_units_to_entity_ids.parquet
│         │    ├── join_text_units_to_relationship_ids.parquet
│         │    └── stats.json
│         └── reports
│              ├── indexing-engine.log
│              └── logs.json
├── prompts
│    ├── claim_extraction.txt
│    ├── community_report.txt
│    ├── entity_extraction.txt
│    └── summarize_descriptions.txt
└── settings.yaml
```

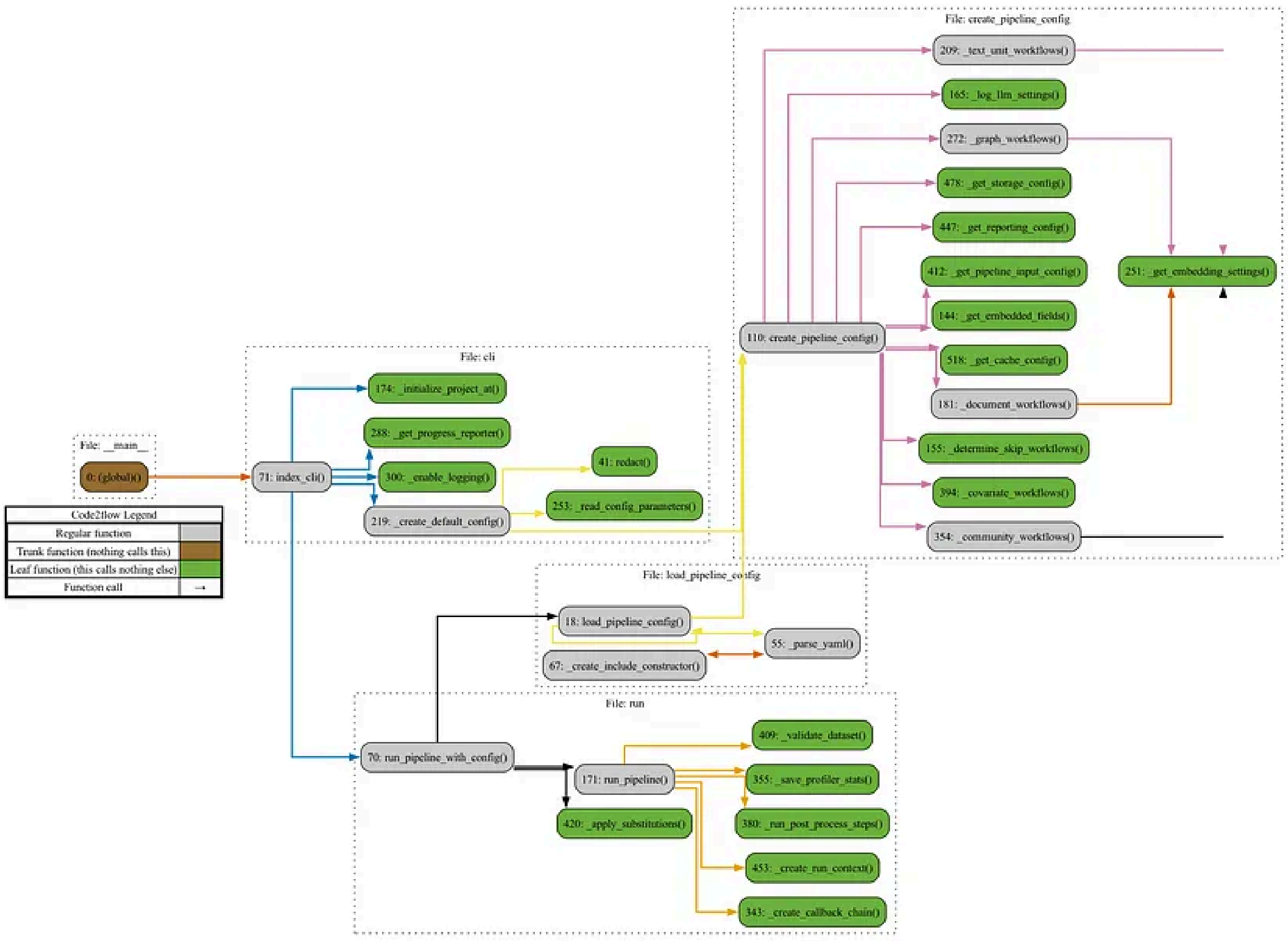
Additionally, the console will print many runtime logs, among which a particularly important one is the complete workflows, involving the complete pipeline orchestration:

```
⠹ GraphRAG Indexer
├── Loading Input (InputFileType.text) - 1 files loaded (0 filtered) ━━━━━━━━━
├── create_base_text_units
├── create_base_extracted_entities
├── create_summarized_entities
├── create_base_entity_graph
├── create_final_entities
├── create_final_nodes
├── create_final_communities
├── join_text_units_to_entity_id
├── create_final_relationships
├── join_text_units_to_relationship_ids
├── create_final_community_reports
├── create_final_text_units
├── create_base_documents
└── create_final_documents
🚀 All workflows completed successfully.
```

Indexing

The indexing phase is considered the core of the entire project, and the overall process is quite complex. The statements to execute indexing are as follows:

```
python -m graphrag.index --init --root ./ragtest
python -m graphrag.index --root ./ragtest
```


File: create_pipeline_config
209: _text_unit_workflows()
165: _log_llm_settings()
272: _graph_workflows()
478: _get_storage_config()
447: _get_reporting_config()
412: _get_pipeline_input_config()
251: _get_embedding_settings()
144: _get_embedded_fields()
110: create_pipeline_config()
518: _get_cache_config()
181: _document_workflows()
155: _determine_skip_workflows()
394: _covariate_workflows()
354: _community_workflows()
File: cli
174: _initialize_project_at()
288: _get_progress_reporter()
File: __main__
0: (global)()
71: index_cli()
300: _enable_logging()
41: redact()
253: _read_config_parameters()
219: _create_default_config()
Code2flow Legend
Regular function
Trunk function (nothing calls this)
Leaf function (this calls nothing else)
Function call
File: load_pipeline_config
18: load_pipeline_config()
55: _parse_yaml()
67: _create_include_constructor()
File: run
409: _validate_dataset()
70: run_pipeline_with_config()
171: run_pipeline()
355: _save_profiler_stats()
420: _apply_substitutions()
380: _run_post_process_steps()
453: _create_run_context()
343: _create_callback_chain()

These commands call the main function in `graphrag/index/__main__.py`, using argparse to parse input parameters, and ultimately calls the `index_cli` function in `graphrag/index/cli.py`.

Let's examine the call chain of related functions, focusing on the key components:

- The `cli.py::index_cli()` function first determines whether to initialize the current folder based on user input parameters (e.g., `--init`). It checks for the existence of configuration files, prompts, and `.env` in the directory, creating them if they don't exist. These files include `settings.yaml` and various prompt files.

- For the actual indexing operation, it executes an internal function `cli.py::index_cli()._run_workflow_async()`, which primarily involves two functions: `cli.py::_create_default_config()` and `run.py::run_pipeline_with_config()`.

Due to space constraints, we'll focus on the default configuration process:

Default Configuration Generation:

- `cli.py::_create_default_config()` checks the root directory and `settings.yaml`.
- It then executes `cli.py::_read_config_parameters()` to read system configurations (LLM, chunks, cache, storage, etc.).
- The crucial step is creating a pipeline configuration based on current parameters, implemented in `create_pipeline_config.py::create_pipeline_config()`. This is one of the most complex modules in the project.

The core logic of `create_pipeline_config.py::create_pipeline_config()` is:

```
result = PipelineConfig(
    root_dir=settings.root_dir,
    input=_get_pipeline_input_config(settings),
    reporting=_get_reporting_config(settings),
    storage=_get_storage_config(settings),
    cache=_get_cache_config(settings),
    workflows=[
        *_document_workflows(settings, embedded_fields),
        *_text_unit_workflows(settings, covariates_enabled, embedded_fields),
        *_graph_workflows(settings, embedded_fields),
        *_community_workflows(settings, covariates_enabled, embedded_fields),
        *(_covariate_workflows(settings) if covariates_enabled else []),
    ],
)
```

This code generates a complete workflow sequence based on different functions, using templates from the `workflows/v1` directory. Note that dependencies between workflows are not considered at this stage.

- Pipeline Execution:

1. `run.py::run_pipeline_with_config()` loads the existing pipeline configuration.
2. It creates subdirectories (cache, storage, input, output, etc.).
3. It uses `run.py::run_pipeline()` to execute each workflow sequentially and return results.

The `run.py::run_pipeline()` function is crucial, with two main components:

1. Loading workflows: `workflows/load.py::load_workflows()` creates regular workflows and handles topological sorting.
   – `workflows/load.py::create_workflow()` : Creates workflows using existing templates.
   – `graphlib::topological_sort()` : Calculates DAG topological sorting based on workflow dependencies.

2 Executing operations like `inject_workflow_data_dependencies()` , `write_workflow_stats()` , and `emit_workflow_output` for dependency injection, data writing, and saving.

```
await dump_stats()

for workflow_to_run in workflows_to_run:
    # Try to flush out any intermediate dataframes
    gc.collect()

    workflow = workflow_to_run.workflow
    workflow_name: str = workflow.name
    last_workflow = workflow_name

    log.info("Running workflow: %s...", workflow_name)

    if is_resume_run and await storage.has(
        f"{workflow_to_run.workflow.name}.parquet"
    ):
        log.info("Skipping %s because it already exists", workflow_name)
        continue

    stats.workflows[workflow_name] = {"overall": 0.0}
    await inject_workflow_data_dependencies(workflow)

    workflow_start_time = time.time()
```

```
        result = await workflow.run(context, callbacks)
        await write_workflow_stats(workflow, result, workflow_start_time)

        # Save the output from the workflow
        output = await emit_workflow_output(workflow)
        yield PipelineRunResult(workflow_name, output, None)
        output = None
        workflow.dispose()
        workflow = None

    stats.total_runtime = time.time() - start_time
    await dump_stats()
```

In summary, the indexing phase workflow consists of:

1. Initialization: Generating necessary configuration files, caches, and directories.
2. Indexing: Creating a series of pipelines using workflow templates, adjusting execution order based on dependencies, and executing them sequentially.

This process ensures a systematic and efficient approach to building the knowledge graph and preparing for subsequent query operations.

Workflow

Up to this point, we've focused on GraphRAG's built-in pipeline orchestration system rather than the business logic of the indexing phase. Let's examine how a specific workflow runs, using `index/workflows/v1/create_final_entities.py` as an example.

DataShaper

Before diving into the workflow, it's important to understand DataShaper, another framework used in the project. DataShaper is an open-source library from Microsoft for executing workflow processing, featuring many built-in components (called "Verbs").

Think of DataShaper as an assembly line, where each step defines an operation on the input data, similar to clip, rotate, and scale operations in PyTorch image transformations. If this concept is still unclear, I recommend running the `examples/single_verb` demo in the official documentation for a better understanding. Functionally, it's somewhat similar to Prefect.

Knowledge Graph Construction

The workflow we're examining is `create_final_entities.py`. This workflow depends on `workflow:create_base_extracted_entities` and defines operations such as `cluster_graph` and `embed_graph`. The `cluster_graph` operation uses the Leiden strategy, implemented in `index/verbs/graph/clustering/cluster_graph.py`:

```
from datashaper import TableContainer, VerbCallbacks, VerbInput, progress_itera

@verb(name="cluster_graph")
def cluster_graph(
    input: VerbInput,
    callbacks: VerbCallbacks,
    strategy: dict[str, Any],
    column: str,
    to: str,
    level_to: str | None = None,
    **_kwargs,
) -> TableContainer:
```

As you can see, it's essentially a function decorated with a `@verb` decorator. The Leiden algorithm used here comes from the graspologic library, a Python package for graph statistics.

Pipeline

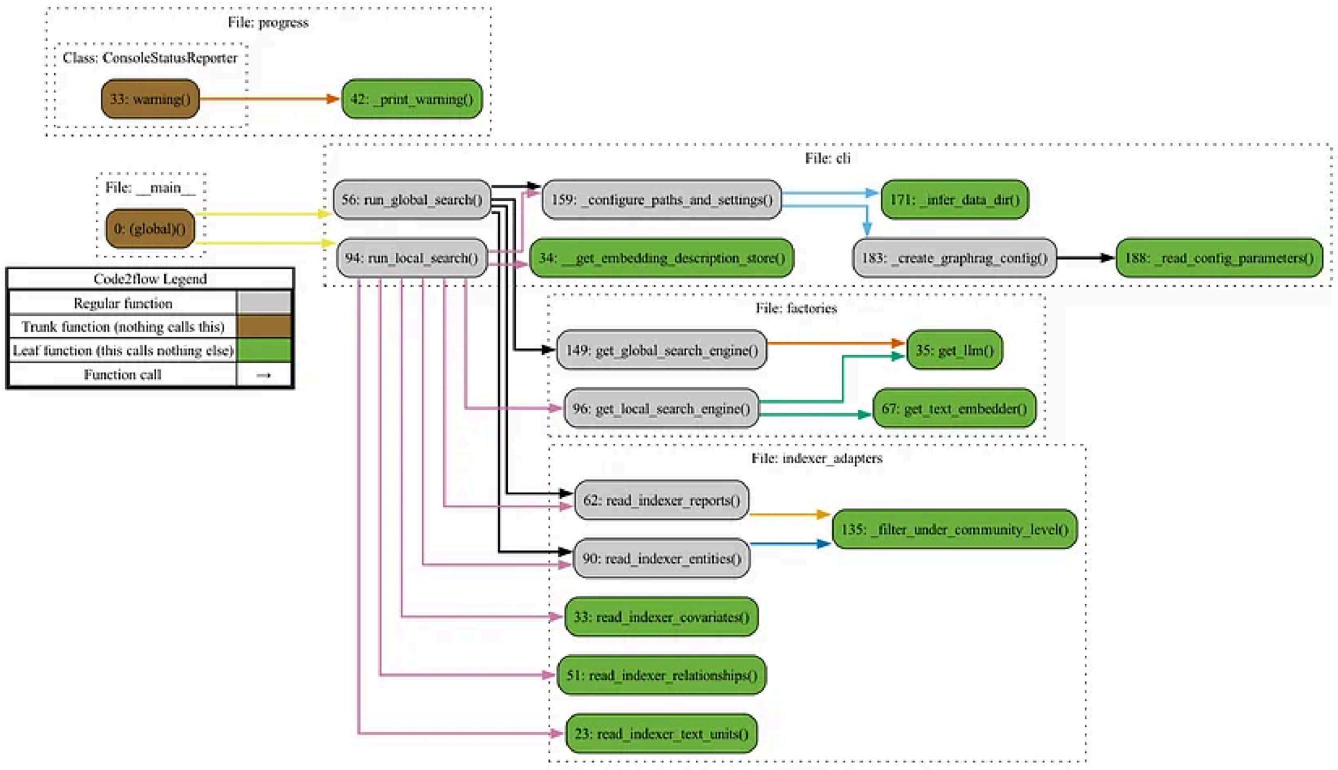
With an understanding of the workflow execution logic, we can piece together the complete workflow based on the orchestration logs or the `artifacts/stats.json` file. The official documentation provides detailed explanations in the Indexing Dataflow section. However, the pipeline I've extracted from the source code shows some differences. I encourage those with related experience to share their thoughts on this.

Query

The query phase pipeline is relatively simpler. Here are the commands to execute queries:

```
# Global search
python -m graphrag.query \
--root ./ragtest \
--method global \
"What are the top themes in this story?"

# Local search
python -m graphrag.query \
--root ./ragtest \
--method local \
"Who is Scrooge, and what are his main relationships?"
```


File: progress
Class: ConsoleStatusReporter
33: warning()
42: _print_warning()
File: __main__
0: (global)()
File: cli
56: run_global_search()
94: run_local_search()
159: _configure_paths_and_settings()
171: _infer_data_dir()
34: __get_embedding_description_store()
183: _create_graphrag_config()
188: _read_config_parameters()
File: factories
149: get_global_search_engine()
35: get_llm()
96: get_local_search_engine()
67: get_text_embedder()
File: indexer_adapters
62: read_indexer_reports()
135: _filter_under_community_level()
90: read_indexer_entities()
33: read_indexer_covariates()
51: read_indexer_relationships()
23: read_indexer_text_units()
Code2flow Legend
Regular function
Trunk function (nothing calls this)
Leaf function (this calls nothing else)
Function call

There are two search modes: Global and Local. The main function in `graphrag/query/__main__.py` routes to either `cli::run_local_search()` or `cli::run_global_search()` based on the parameters.

Global Search

`cli::run_global_search()` primarily calls `factories.py::get_global_search_engine()` , which returns a `GlobalSearch` class. This class, like `LocalSearch` , is created using the factory pattern. Its core method, `structured_search/global_search/search.py::GlobalSearch.asearch()` , employs a map-reduce approach. It generates answers for each community's summary in parallel using a large language model, then aggregates these answers for the final result. The map/reduce prompts provided by the author offer detailed explanations of this process.

This map-reduce mechanism is the reason global search consumes a large number of tokens.

Local Search

Similarly, `cli::run_local_search()` calls `factories.py::get_local_search_engine()`, returning a `LocalSearch` class. Its `asearch()` method is simpler, providing responses directly based on the context. This mode is more akin to conventional RAG semantic retrieval strategies and consumes fewer tokens.

Unlike global search, the Local mode integrates multiple data sources including nodes, community_reports, text_units, relationships, entities, and covariates. The official documentation provides a comprehensive explanation of this aspect

Some Thoughts

- GraphRAG's core innovation lies in its approach to the Query-Focused Summarization (QFS) task, which, to my knowledge, is a pioneering effort in this field. The closest conceptual approach prior to this was likely RAPTOR, although it wasn't specifically designed for knowledge graphs. QFS and Multi-Hop Q&A are currently challenging areas that traditional RAG systems struggle to address. However, they have wide-ranging applications, particularly in data analysis. While the current implementation of GraphRAG is resource-intensive, it opens up new possibilities in this domain.

- One of GraphRAG's standout features is its comprehensive built-in workflow orchestration system, which is uncommon in other Knowledge Graph-based RAG frameworks. This approach, defining workflows based on templates with configurable flexibility and traceable steps, could be a promising direction for future development. It offers more control and transparency compared to systems that rely entirely on large language models for all operations. The conventional RAG components, such as embedding and retrieval, don't break new ground, although the addition of a community detection algorithm is noteworthy.

- GraphRAG's fine-grained processing of knowledge graphs, including the use of the Leiden algorithm for community detection and local search integrating multi-source data, is impressive. An interesting aspect is the project's approach to entity extraction. Unlike some Pydantic-based approaches, GraphRAG relies entirely on large language models without constraining the schema of triples. The author explains that due to similarity clustering, variations in model extractions don't significantly impact the final community generation. This approach markedly improves robustness.

There are areas where GraphRAG could be improved. The project introduces several potentially confusing terms (Emit, Claim, Verbs, Reporting, etc.) and incorporates some relatively niche Microsoft libraries, which can make the system more challenging to understand. Additionally, there's room for improved modularization, particularly in decoupling the OpenAI components.

In conclusion, Microsoft's GraphRAG framework is a substantial contribution to the field. It introduces novel approaches to complex problems in natural language processing and knowledge graph manipulation. Despite some areas for potential improvement, it's a project worth studying in depth, offering valuable insights and innovative solutions in the realm of advanced information retrieval and summarization.

Reference

- Welcome to GraphRAG (microsoft.github.io)
- GraphRAG: LLM-Derived Knowledge Graphs for RAG (youtube.com)
- GraphRAG is now on GitHub | Hacker News (ycombinator.com)
- GraphRAG & Ollama — intelligent Search of local data : r/LocalLLaMA (reddit.com)
- GraphRAG on narrative private data : r/LocalLLaMA (reddit.com)

Knowledge Graph Retrieval Augmented Gen AI Microsoft Llm

Follow

Written by Calvin Ku

127 Followers · Editor for Percena

Sapere aude.

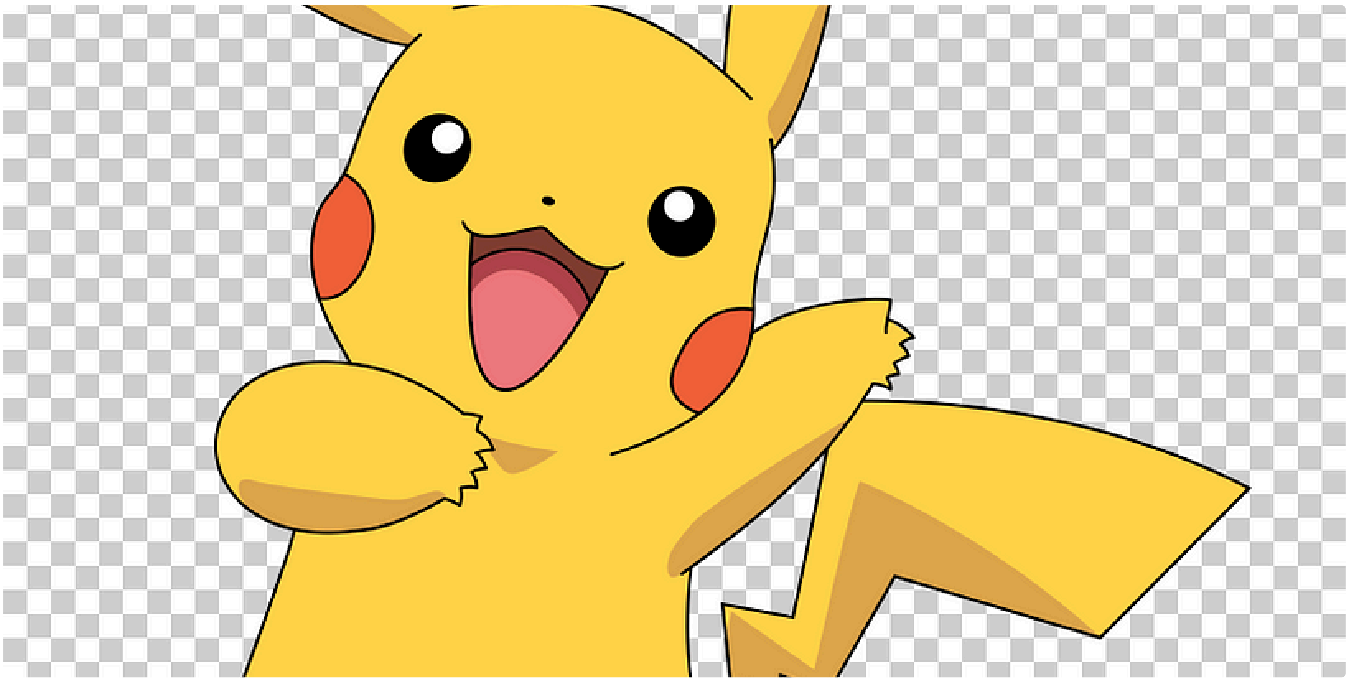
More from Calvin Ku and Percena

Calvin Ku in Noumena

How does DataLoader work in PyTorch?

Why use DataLoader?

Sep 9, 2018 178 2

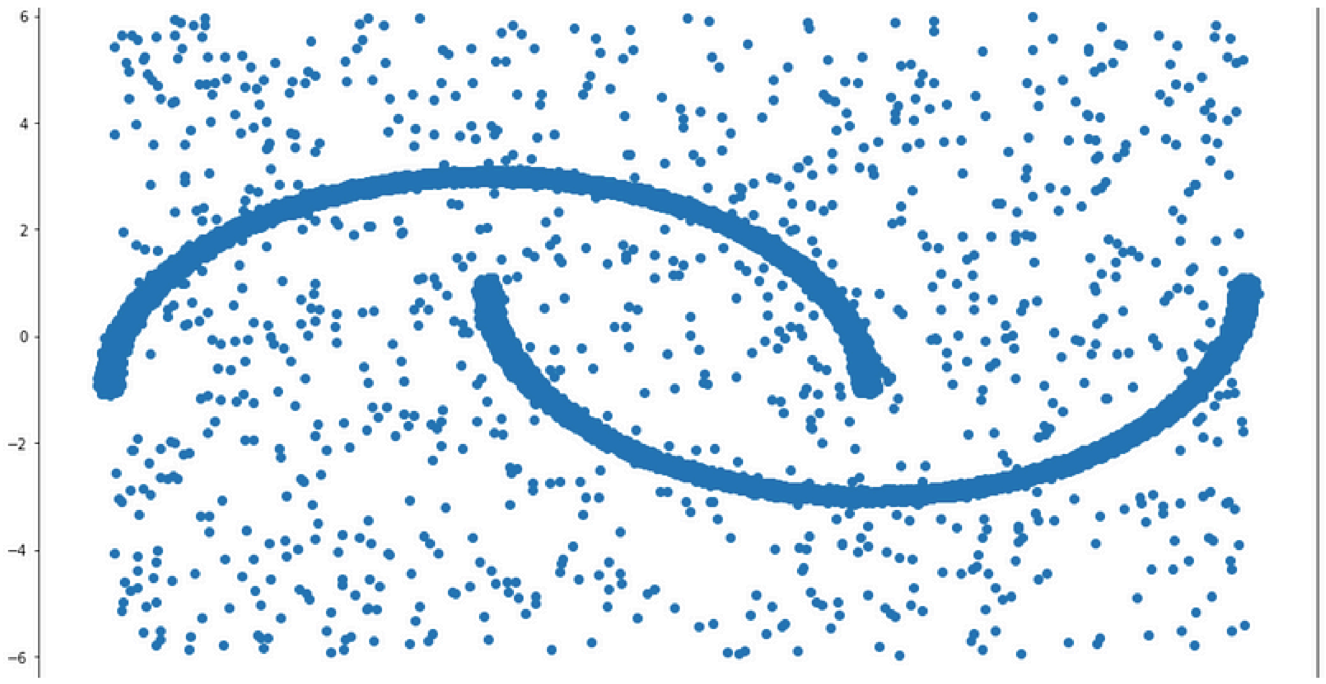


Calvin Ku

Best practice for combining an image with text on a button for Android

I decided to write an article about this because after hours of research I couldn't find a clear, straightforward article for noobs like...

Oct 4, 2021 345



Calvin Ku

Trying Out Machine Learning on Elasticsearch using Python

Recently I've been doing some research on Elasticsearch to see if it suits my needs at work. One thing that I care about most is its...

Mar 29, 2021 218

Calvin Ku

Beating the Naive Model in the Stock Market

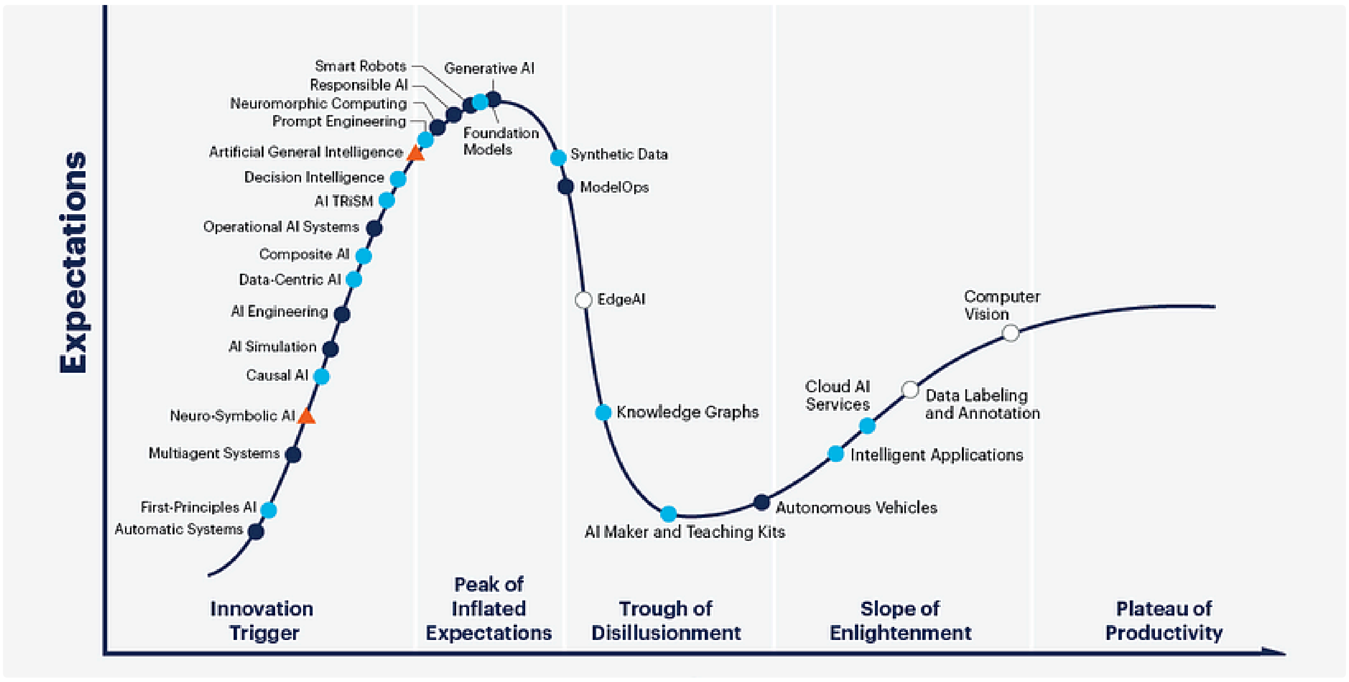
Recently while studying for the Self-Driving Car Nanodegree from Udacity, I came across something really amazing, called the Kalman filter...

May 23, 2017 89 2

See all from Calvin Ku

See all from Percena

Recommended from Medium

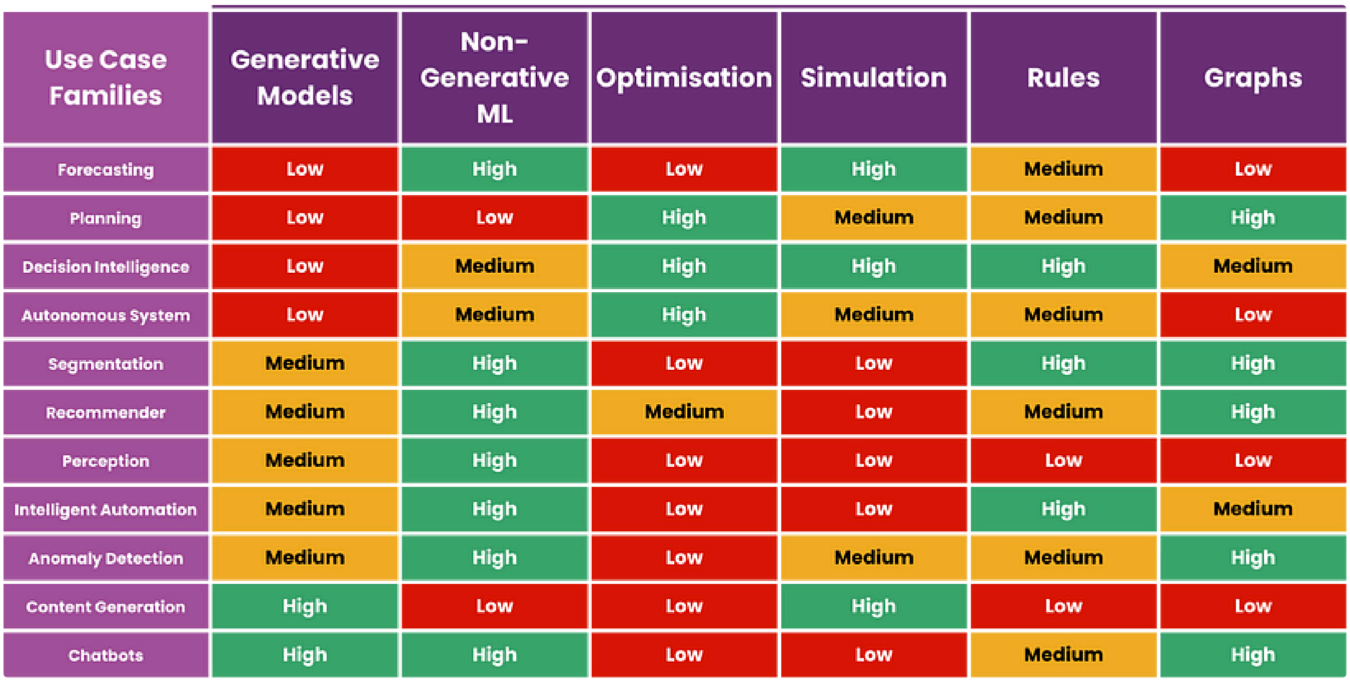


Vishal Rajput in AIGuys

Why GEN AI Boom Is Fading And What's Next?

Every technology has its hype and cool down period.

Sep 4 1.97K 60



Use Case Families	Generative Models	Non-Generative ML	Optimisation	Simulation	Rules	Graphs
Forecasting	Low	High	Low	High	Medium	Low
Planning	Low	Low	High	Medium	Medium	High
Decision Intelligence	Low	Medium	High	High	High	Medium
Autonomous System	Low	Medium	High	Medium	Medium	Low
Segmentation	Medium	High	Low	Low	High	High
Recommender	Medium	High	Medium	Low	Medium	High
Perception	Medium	High	Low	Low	Low	Low
Intelligent Automation	Medium	High	Low	Low	High	Medium
Anomaly Detection	Medium	High	Low	Medium	Medium	High
Content Generation	High	Low	Low	High	Low	Low
Chatbots	High	High	Low	Low	Medium	High

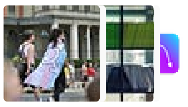
Christopher Tao in Towards AI

Do Not Use LLM or Generative AI For These Use Cases

Choose correct AI techniques for the right use case families

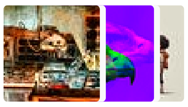
Aug 10 3.6K 39

Lists



Generative AI Recommended Reading

52 stories · 1416 saves



What is ChatGPT?

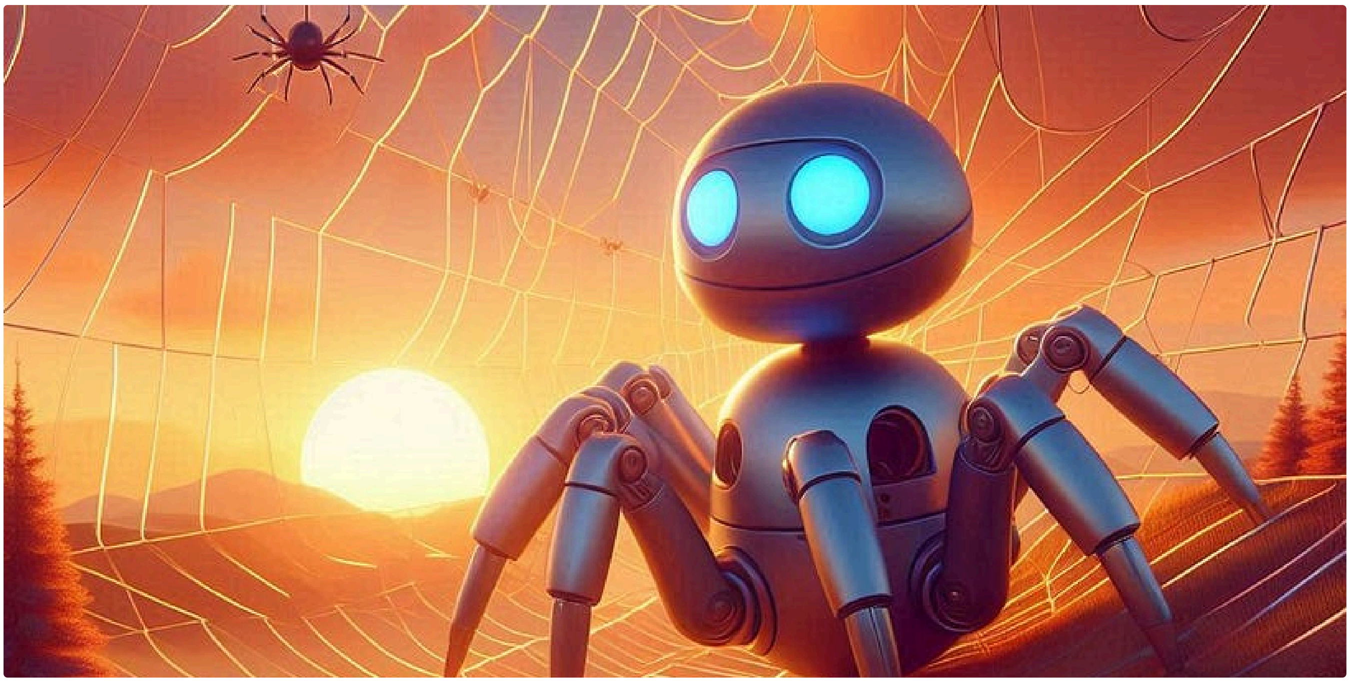
9 stories · 444 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 471 saves

Natural Language Processing

1741 stories · 1324 saves



Valentina Alto in Microsoft Azure

Introducing GraphRAG with LangChain and Neo4j

Part 1: Getting Started with Graph Databases in the LLMs Era

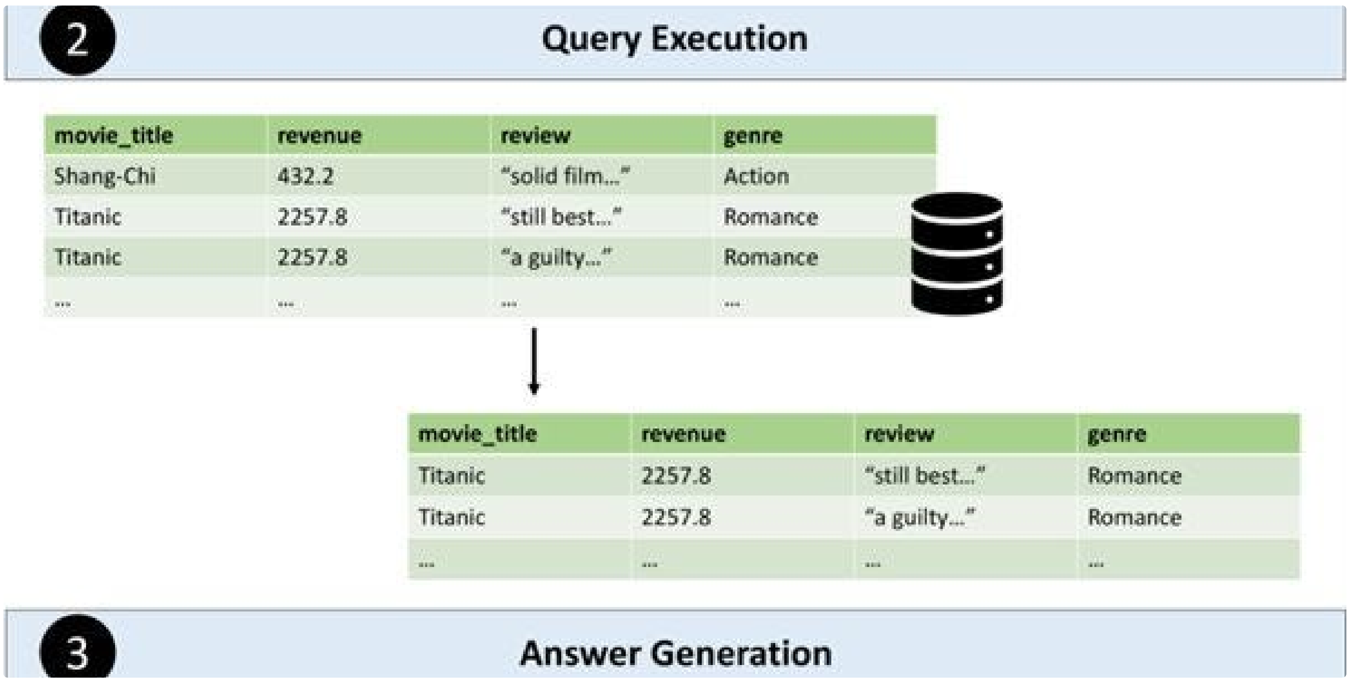
Apr 28 836 14

Steve Hedden in Towards Data Science

How to Implement Knowledge Graphs and Large Language Models (LLMs) together at the Enterprise Level

A survey of the current methods of integration

Apr 20 630 8



movie_title	revenue	review	genre
Shang-Chi	432.2	"solid film..."	Action
Titanic	2257.8	"still best..."	Romance
Titanic	2257.8	"a guilty..."	Romance
...	...	...	...

movie_title	revenue	review	genre
Titanic	2257.8	"still best..."	Romance
Titanic	2257.8	"a guilty..."	Romance
...	...	...	...

Pavan Emani in Artificial Intelligence in Plain English

Goodbye, Text2SQL: Why Table-Augmented Generation (TAG) is the Future of AI-Driven Data Queries!

Exploring the Future of Natural Language Queries with Table-Augmented Generation.

Sep 11 984 16

Prakash Joshi Pax

Fabric: The Best AI Tool That Nobody is Talking About

An open-source AI tool to automate every day tasks

Aug 26 2K 18

See more recommendations