GraphContainer: A Unified Platform for Comparing and Debugging Graph RAG Methods

Seonho An*
KAIST
Daejeon, Republic of Korea
asho1@kaist.ac.kr

Chaejeong Hyun*
KAIST
Daejeon, Republic of Korea
hchaejeong@kaist.ac.kr

Min-Soo Kim†
KAIST
Daejeon, Republic of Korea
minsoo.k@kaist.ac.kr

ABSTRACT

Graph RAG mitigates hallucinations and stale knowledge in LLMs, particularly for multi-hop question answering. However, existing approaches remain highly fragmented and incompatible. The structural heterogeneity of graph formats across different frameworks and the lack of granular visualization tools make it exceedingly difficult to evaluate and compare retrieval behaviors. To bridge this gap, we propose **GraphContainer**, a novel platform designed to unify and visualize diverse graph RAG workflows. GraphContainer features two key components: (1) a **Unified Graph Representation (UGR) layer** that seamlessly standardizes multi-format graphs, and (2) a **Graph Recorder** that tracks and visually renders the step-by-step retrieval process. Through an interactive web interface, we demonstrate GraphContainer's ability to import heterogeneous graphs and perform live, traceable visual debugging of graph RAG methods. Ultimately, we show how GraphContainer enables controlled comparisons of various graph formats and retrieval strategies, lowering the barrier for researchers and practitioners to design optimal graph RAG pipelines. A demonstration video is available at https://youtu.be/O02eNJLwkU0.

PVLDB Reference Format:
Seonho An, Chaejeong Hyun, and Min-Soo Kim. GraphContainer: A Unified Platform for Comparing and Debugging Graph RAG Methods. PVLDB, 19(12): 4550-4553, 2026.
doi:10.14778/3827998.3828063

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/asmath472/GraphContainer.

1 INTRODUCTION

Retrieval-augmented generation (RAG) methods have emerged as a foundational technology for addressing the hallucination and out-of-date information problems inherent in large language models (LLMs) [4]. Recently, graph RAG methods—which leverage structured graph data into the retrieval pipeline—demonstrate superior effectiveness, particularly in multi-hop question answering [10].

The proliferation of graph RAG methods poses a challenge in identifying the best approach for specific datasets or tasks [2], a research area that remains largely unexplored due to two fundamental bottlenecks. First, comparative analysis is severely hindered by **structural heterogeneity**. As shown in Table 1, representative graph formats utilize highly divergent data notation. Since these formats are tightly coupled to their native graph RAG method, it is difficult to cross-evaluate methods across datasets. Second, **analyzing the graph retrieval process is elusive**. Tracking the step-by-step selection of nodes and edges during a query requires dedicated visualization tools. Although some methods (e.g. LightRAG [5]) offer built-in visualizers, they are strictly isolated to their own architectures. Consequently, researchers and practitioners lack a cohesive environment to compare, evaluate, and debug diverse graph RAG strategies [2].

Table 1: Notation of four representative graph formats. We use u, v for node identifiers, x_u for node content, $\mathbf{z}_u \in \mathbb{R}^d$ for node embeddings, e for edge identifiers, x_e for edge content, and $\mathbf{z}_e \in \mathbb{R}^d$ for edge embeddings.

Graph Format	Notation
Component Graph [1]	$\mathcal{G}_{\text{cmp}} = (\mathcal{N}, \mathcal{E}, \mathcal{I})$ where $\mathcal{N} = \{(u, x_u)\}$, $\mathcal{E} = \{(u, v)\}$, $\mathcal{I} = \{(u, \mathbf{z}_u)\}$
Attribute Bundle Graph [3, 5]	$\mathcal{G}_{\text{bun}} = (\mathcal{N}^*, \mathcal{E}^*)$ where $\mathcal{N}^* = \{(u, x_u, \mathbf{z}_u)\}$, $\mathcal{E}^* = \{(e, u, v, x_e, \mathbf{z}_e)\}$
Topology–Semantic Graph [6]	$\mathcal{G}_{\text{ts}} = (\mathcal{T}, \mathcal{X}_N, \mathcal{X}_E, \{\mathcal{I}_N, \mathcal{I}_E\})$, where $\mathcal{T} = \{(e, u, v)\}$, $\mathcal{X}_N = \{(u, x_u)\}$, $\mathcal{X}_E = \{(e, x_e)\}$ $\mathcal{I}_N = \{(u, \mathbf{z}_u)\}$, $\mathcal{I}_E = \{(e, \mathbf{z}_e)\}$
Subgraph Union Graph [7, 8, 11]	$\mathcal{G}_{\text{sub}} = \{(\mathcal{N}_i, \mathcal{E}_i, \{\mathcal{I}_{N,i}, \mathcal{I}_{E,i}\})\}_{i=1}^{m}$ where $\mathcal{N}_i = \{(u, x_u)\}$, $\mathcal{E}_i = \{(e, u, v)\}$ $\mathcal{I}_{N,i} = \{(u, \mathbf{z}_u) \mid u \in \mathcal{N}_i\}$, $\mathcal{I}_{E,i} = \{(e, \mathbf{z}_e) \mid e \in \mathcal{E}_i\}$

To bridge this gap, we propose **GraphContainer**, a novel platform for the comparison and visual debugging of graph RAG methods. GraphContainer addresses the above challenges through two key components: (1) a **Unified Graph Representation (UGR) layer** and (2) a **Graph Recorder**. First, the UGR layer maps disparate graph formats (e.g., four major formats detailed in Table 1) into a standardized structure called the **Unified Graph State (UGS)**. The UGS comprehensively captures both structural and semantic attributes across all formats, enabling the underlying retrievers to operate over a shared representation. Second, the **Graph Recorder** tracks the step-by-step execution of retrieval methods operating on the UGS. This recorded process is then rendered through our Live Visualizer, enabling users to inspect exact retrieval behaviors and interactively refine their graph RAG pipelines.

*Co-first author.
†Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828063

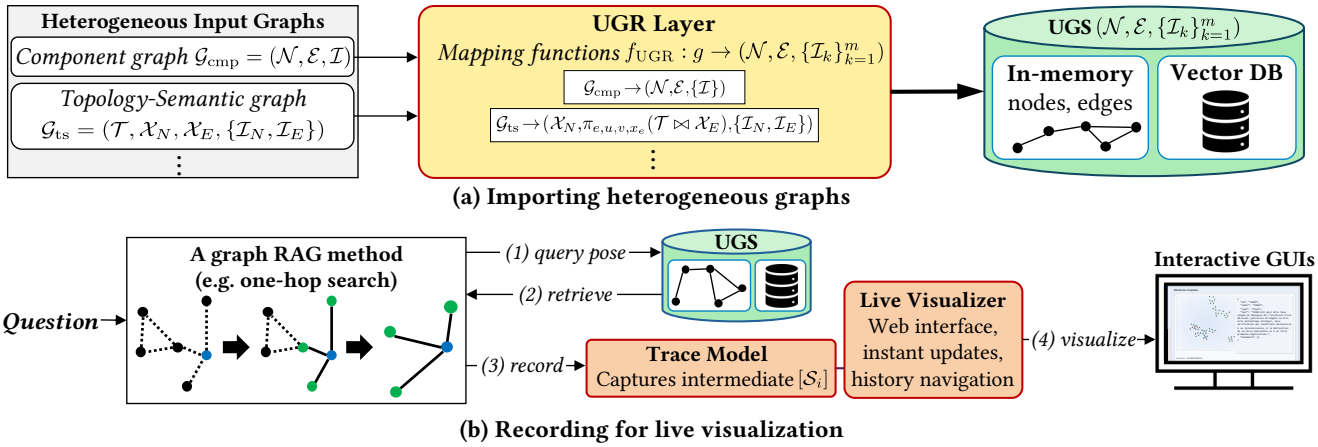

Figure 1: Overview of GraphContainer. (a) UGS transformation via UGR. (b) Live visualization for Graph RAG.

We demonstrate GraphContainer's capabilities through an end-to-end scenario where users import heterogeneous graphs into a unified representation and visualize the entire retrieval process via a live web interface. Finally, we present an empirical study illustrating how GraphContainer facilitates controlled, rigorous comparisons across varying graph construction and retrieval methods.

2 SYSTEM OVERVIEW

As illustrated in Figure 1, our GraphContainer provides two core functionalities: (a) the ingestion and standardization of heterogeneous graphs via the UGR layer, and (b) dynamic graph tracking and live visualization via Graph Recorder. Figure 1(a) depicts functionality (a), where the UGR layer employs format-specific mapping functions to decode varying graph structures into a single canonical UGS. Meanwhile, for functionality (b), the Graph Recorder captures the dynamic execution state of the retrievers operating over this shared UGS. Figure 1(b) shows the step-by-step process (1–4) that translates these recorded results into a live visualization within the graph RAG workflow.

2.1 Unified Graph Representation Layer

Existing graph RAG methods encode structural knowledge using diverse *graph formats*. Formally, a graph format is defined as:

Definition 2.1 (Graph Format). A graph format $\mathcal{G}$ is defined as a data schema that encodes nodes $\mathcal{N}$, edges $\mathcal{E}$, and m optional vector indices $\{\mathcal{I}_k\}_{k=1}^m$.

These encoded elements serve as fundamental data components for retrieval in graph RAG methods [1, 10].

Although various graph formats hold logically equivalent information, their structural discrepancies make systematic comparison of graph RAG methods difficult. To resolve this, we propose the **Unified Graph Representation (UGR)** layer. The UGR layer acts as a translator, taking disparate graph formats and standardizing them into a common structure referred to as the **Unified Graph State (UGS)**. Here, the UGS serves as the materialized representation of $\mathcal{N}$, $\mathcal{E}$, and $\{\mathcal{I}_k\}_{k=1}^m$. Nodes, edges, and their attributes are maintained in the *in-memory UGS*, while vector indices are managed via an external vector database. Formally, the UGR layer is defined as follows:

Definition 2.2 (Unified Graph Representation Layer). Given a graph instance g encoded in format $\mathcal{G}$, the UGR is defined by a set of mapping functions, $f_{\text{UGR}} : g \to (\mathcal{N}, \mathcal{E}, \{\mathcal{I}_k\}_{k=1}^m)$, that transform format-specific components into the UGS.

In practice, our proposed UGR layer supports four major graph formats categorized from representative graph RAG frameworks: (1) *Component Graph* in FastInsight [1], (2) *Attribute Bundle Graph* in LightRAG [5] and PathRAG [3], (3) *Topology-Semantic Graph* in HippoRAG [6], (4) *Subgraph Union Graph* in G-Retriever [7], GRAG [8], and KG2RAG [11]. Although we initially focus on these four formats, users can easily extend this to custom graph formats by adding corresponding UGR mapping functions. We summarize the format-specific decoding mappings induced by f_{UGR} in Table 2.

Table 2: Summary of decoding mappings from each graph format to the UGS. Here, π denotes projection, $\bowtie$ denotes join, and ϕ_N, ϕ_E denote parsing functions that convert textual triplets into node and edge records.

Graph Format	Decoding Mapping
Component Graph	$(\mathcal{N}, \mathcal{E}, \{\mathcal{I}\})$
Attribute Bundle Graph	$(\pi_{u,x_u}(\mathcal{N}^*), \pi_{e,u,v,x_e}(\mathcal{E}^*), \{\pi_{u,z_u}(\mathcal{N}^*), \pi_{e,z_e}(\mathcal{E}^*)\})$
Topology–Semantic Graph	$(\mathcal{X}_N,\ \pi_{e,u,v,x_e}(\mathcal{T} \bowtie \mathcal{X}_E),\ \{\mathcal{I}_N, \mathcal{I}_E\})$
Subgraph Union Graph	$(\bigcup_{i=1}^m \mathcal{N}_i,\ \bigcup_{i=1}^m \mathcal{E}_i,\ \{\bigcup_{i=1}^m \mathcal{I}_{N,i}, \bigcup_{i=1}^m \mathcal{I}_{E,i}\})$

2.2 Graph Recorder

The Graph Recorder is comprised of two components: a **Trace Model** that captures intermediate retrieval results (i.e., retrieved nodes and edges), and a **Live Visualizer** that renders them to analyze retrieval processes on the UGS.

During retrieval, a query is first processed by a selected graph RAG method, which retrieves relevant node and edge identifiers from the UGS (steps (1)–(2) in Figure 1(b)). The retriever then calls the Trace Model to organize each retrieval progress per query as a *session* and record the current retrieval results (step (3)) by capturing the nodes and edges selected at each step. As retrieval progresses, these recorded results accumulate into a sequence $[\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_m]$ for each session, where each $\mathcal{S}_i$ denotes the subgraph at the i-th record call induced by the retrieved nodes and edges in the in-memory UGS. By recording each step as a subgraph $\mathcal{S}_i$ that stores

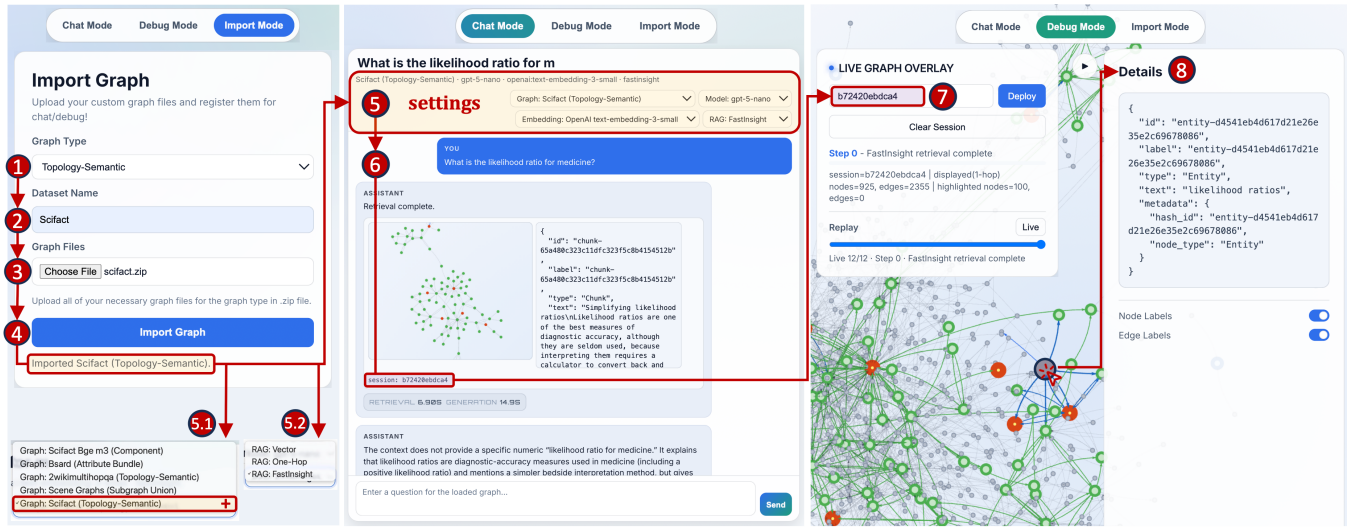

Figure 2: A demonstration flow for our GraphContainer.

only incremental overlay information, the Trace Model minimizes storage overhead while preserving the structural context required for inspection.

Next, the Live Visualizer consumes the recorded sequence produced by the Trace Model and renders it through a web-based interactive interface (step (4)). Each invocation of step (3) appends a new subgraph $\mathcal{S}_i$ to the sequence and triggers an immediate update of the visualized graph to reflect the latest retrieval step. The visualizer preserves this sequence as the retrieval history within a session, allowing users to navigate $\mathcal{S}_i$ and inspect how the retrieved subgraph evolves. For example, users can monitor the evolving retrieval process while interacting with standard debugging environments (e.g., Python debuggers), or modify the graph structure or retrieval logic to observe the changes immediately in the interface.

3 DEMONSTRATION SCENARIOS

We demonstrate the debuggability and comparability of GraphContainer through two scenarios: **Interactive Visual Debugging** and **Comparative Retrieval Analysis**. In the first scenario, users experience the end-to-end GraphContainer workflow by *importing* a sample graph through the GUI, *executing* Graph RAG queries, and *visualizing* the step-by-step retrieval process. In the second scenario, users compare various retrieval methods—specifically vector search, one-hop search, and FastInsight—to systematically analyze how different retrieval mechanisms influence generated answers.

3.1 Scenario 1: Interactive Visual Debugging

The first scenario consists of three sequential phases: Importing, Executing, and Visualizing. These phases correspond to steps (1)–(4), (5)–(6), and (7)–(8) in Figure 2, respectively.

Importing. In this phase, we provide users with two sample graphs: (a) BSARD in the Attribute Bundle format, and (b) SciFact in the Topology-Semantic format. To begin the demonstration session, users first navigate to the *Import Mode* on the website. Users select one of the provided graphs and upload it to the GraphContainer platform. Specifically, users select the format of the chosen graph ((1) in Figure 2), input the name of the dataset to be displayed ((2)), upload the graph files ((3)), and click the *Import Graph* button ((4)). Subsequently, the server utilizes UGR to decode the uploaded graph into the canonical UGS.

Executing. Switching to the *Chat Mode* interface at the top of the web, users configure the graph RAG pipeline ((5)) by selecting the target graph dataset ((5.1)), the embedding and LLM models, and the retrieval method ((5.2)). Users then submit a natural language query and review the generated response ((6)). To facilitate the demonstration, we provide default queries (e.g., "What do I risk if I violate professional confidentiality?" for BSARD, and "What is the likelihood ratio for medicine?" for SciFact) and set FastInsight and one-hop search as the default retrieval methods. Users can also interactively submit their own custom queries. Once the query completes, users can thoroughly inspect the underlying retrieval mechanics. The interface details the selected nodes (shown as red dots in the displayed subgraph), the retrieval and generation latencies, the final generated response, and the *session_id*.

Visualizing. In this final phase, users transition to the *Debug Mode*. Upon submitting the *session_id* from the *Executing* phase into the Live Graph Overlay panel and clicking *Deploy*, the system dynamically renders the retrieved graph ((7)). Users can then interactively inspect specific node and edge attributes or utilize the replay slider to visualize the step-by-step retrieval sequence ((8)).

3.2 Scenario 2: Comparative Retrieval Analysis

In the second scenario, users execute and evaluate various RAG methods using three pre-defined retrievers: (1) vector search, which retrieves the top-K nodes using an embedding model; (2) one-hop search, which expands the initial top-K selection to include immediate neighboring nodes; and (3) FastInsight [1].

Returning to the *Chat Mode* interface, users execute an identical query while varying the selected retrieval method ((5.2)). By using the unique *session_id* from each execution, users can render and systematically contrast the resulting subgraphs in the *Debug Mode*. This process illustrates how GraphContainer seamlessly facilitates side-by-side comparative analysis across diverse graph RAG methods.

4 EXPERIMENTAL STUDY

Beyond the initial demonstration, we conduct an empirical study to illustrate the experiment types and analyses enabled by GraphContainer. Specifically, we investigate how different combinations of graph construction and retrieval choices affect overall graph RAG performance. While the heterogeneous formats produced by multiple graph RAG methods typically hinder direct comparisons, GraphContainer overcomes this challenge by unifying these representations into a common UGS.

Settings. We utilize BSARD [9] and SciFact document corpora $(d_1, \ldots, d_n)$ to construct experimental graphs. Specifically, we use three different construction methods: (1) LightRAG, which produces an *Attribute Bundle Graph*; (2) HippoRAG, which produces a *Topology-Semantic Graph*; and (3) a standard k-NN approach that constructs a *Component Graph* $(\mathcal{N}, \mathcal{E}, \{\mathcal{I}\})$ using document embeddings. The UGR layer converts all graphs into UGS, enabling evaluation through a unified interface.

We evaluate two graph retrieval methods introduced in Section 3.2: *One-hop search* and *FastInsight*. These retrievers are used in two corresponding RAG pipelines to generate answers from the retrieved information. The outputs are evaluated using an LLM-as-a-Judge protocol. Overall, we evaluate six combinations (three graph construction methods × two graph RAG methods), resulting in 15 pairwise comparisons for each dataset.

For all experiments, we use OpenAI's text-embedding-3-small for embeddings, Gemma3 (12B) via Ollama for generation, OpenAI's GPT-5-mini as the judge model, and ChromaDB as the vector database. To mitigate positional bias, each comparison is performed twice with reversed answer ordering, yielding a mean win ratio. We use an initial retrieval size of ten and select the final five nodes in FastInsight; all other parameters follow the default settings.

Experimental Results and Discussions. Table 3 reports the 30 pairwise comparisons and reveals two main findings. First, under the same graph construction (yellow cells in Table 3), FastInsight outperforms one-hop, though the margins of improvement exhibit considerable variance. Second, across both retrieval methods, the KNN-constructed graph yields the strongest overall performance on both datasets, followed by the HR graph.

These results suggest that the effectiveness of a graph RAG pipeline depends jointly on the interplay between the graph construction strategy and the retrieval method. More specifically, the significant performance variance between KNN and LLM-mediated restructuring methods (HR and LR) highlights the need for cross-evaluation. GraphContainer facilitates this process by enabling effortless comparisons, allowing researchers to thoroughly test robustness and practitioners to evaluate diverse pipelines with minimal engineering overhead.

Table 3: Pairwise overall win rate. Top: BSARD, Bottom: SciFact. (KNN)/(HR)/(LR) denote k-NN/HippoRAG/LightRAG construction; FI/OH denote FastInsight/One-hop retrieval. Yellow cells indicate FI vs. OH for the same construction.

	Win Rates (%) of Column Method				
vs. Row Method	(KNN)+FI	(KNN)+OH	(HR)+FI	(HR)+OH	(LR)+FI
BSARD Dataset					
(KNN) + OH	59.01	–	–	–	–
(HR) + FI	68.47	61.71	–	–	–
(HR) + OH	75.45	71.33	62.39	–	–
(LR) + FI	95.72	93.92	87.39	83.56	–
(LR) + OH	95.05	88.74	84.91	81.31	50.23
SciFact Dataset					
(KNN) + OH	54.50	–	–	–	–
(HR) + FI	63.06	57.21	–	–	–
(HR) + OH	71.17	66.22	57.66	–	–
(LR) + FI	95.95	94.14	84.68	83.33	–
(LR) + OH	94.14	87.84	81.98	80.18	55.86

5 CONCLUSIONS AND FUTURE WORKS

In this paper, we presented **GraphContainer**, a unified visual analytics platform designed to evaluate and debug graph RAG methods across heterogeneous formats. By integrating a **Unified Graph Representation** layer with an interactive **Graph Recorder**, GraphContainer facilitates standardized graph conversion alongside step-by-step inspection of retrieval behaviors. Through interactive scenarios and an empirical study, we demonstrated the platform's capability to support controlled comparisons across different graph construction and retrieval methods. We expect GraphContainer to lower the barrier to adopting graph RAG by making complex workflows more transparent, comparable, and highly accessible.

For future work, we plan to transition from in-memory graph to a disk-based graph for massive, enterprise-scale support, and integrate LLM-driven automated graph RAG debugging to provide users with natural language explanations.

REFERENCES

[1] Seonho An, Chaejeong Hyun, and Min-Soo Kim. 2026. FastInsight: Fast and Insightful Retrieval via Fusion Operators for Graph RAG. *arXiv preprint arXiv:2601.18579* (2026).

[2] Yukun Cao, Zengyi Gao, Zhiyang Li, Xike Xie, S. Kevin Zhou, and Jianliang Xu. 2025. LEGO-GraphRAG: Modularizing Graph-Based Retrieval-Augmented Generation for Design Space Exploration. *Proc. VLDB Endow.* 18, 10 (June 2025), 3269–3283. https://doi.org/10.14778/3748191.3748194

[3] Boyu Chen, Zirui Guo, Zidan Yang, Yuluo Chen, Junze Chen, Zhenghao Liu, Chuan Shi, and Cheng Yang. 2026. Pathrag: Pruning graph-based retrieval augmented generation with relational paths. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 30183–30191.

[4] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, Haofen Wang, et al. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023), 32.

[5] Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. 2025. LightRAG: Simple and Fast Retrieval-Augmented Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 10746–10761. https://doi.org/10.18653/v1/2025.findings-emnlp.568

[6] Bernal Jiménez Gutiérrez, Yiheng Shu, Weijian Qi, Sizhe Zhou, and Yu Su. 2025. From RAG to Memory: Non-Parametric Continual Learning for Large Language Models. In *International Conference on Machine Learning*. PMLR, 21497–21515.

[7] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *Advances in Neural Information Processing Systems* 37 (2024), 132876–132907.

[8] Yuntong Hu, Zhihan Lei, Zheng Zhang, Bo Pan, Chen Ling, and Liang Zhao. 2025. Grag: Graph retrieval-augmented generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*. 4145–4157.

[9] Antoine Louis and Gerasimos Spanakis. 2022. A statutory article retrieval dataset in French. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 6789–6803.

[10] Yingli Zhou, Yaodong Su, Youran Sun, Shu Wang, Taotao Wang, Runyuan He, Yongwei Zhang, Sicong Liang, Xilin Liu, Yuchi Ma, and Yixiang Fang. 2026. In-Depth Analysis of Graph-Based RAG in a Unified Framework. *Proc. VLDB Endow.* 18, 13 (Jan. 2026), 5623–5637. https://doi.org/10.14778/3773731.3773738

[11] Xiangrong Zhu, Yuexiang Xie, Yi Liu, Yaliang Li, and Wei Hu. 2025. Knowledge graph-guided retrieval augmented generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 8912–8924.