Master thesis report

Enhancing Knowledge Graph Reasoning via Semantic Alignment: An Improved EoG Framework

Haorui Wang haoruiw@chalmers.se

Supervisor: Shuai Wang
Examiner: Risat Pathan

June 10, 2026

Contents

List of Figures

List of Tables

1 Introduction

In this thesis project, we propose a new scoring optimization architecture to enhance the performance and structural reliability of Knowledge Graph Question Answering (KGQA) systems. By establishing a dual-channel decision framework that combines generative language model scores with dense geometric vector similarities, our approach successfully resolves the critical reasoning bottlenecks commonly found in open-source large language models during multi-hop graph exploration tasks. In recent years, the rapid development of Large Language Models (LLMs) has completely transformed the landscape of natural language processing and question-answering systems. Modern large language models possess incredible text generation capabilities and store massive amounts of general historical knowledge within their frozen parametric weights. However, despite their strong conversational behaviors, pure language models still suffer from fundamental engineering limitations when applied to precision-critical tasks. These models frequently produce factual hallucinations, struggle to provide real-time knowledge updates, and cannot easily verify the underlying sources of their generated responses. To overcome these structural limitations, integrating external structured knowledge sources—such as massive, open-domain knowledge graphs—has become a dominant research direction. Knowledge graphs, such as the full-scale Freebase database repository, store factual data as explicit entity nodes connected by typed relational edges. This relational schema provides a highly stable, verifiable, and precise foundation for truth tracing. Traditional Retrieval-Augmented Generation (RAG) frameworks attempt to link these two worlds by cutting long textual documents into flat text chunks and appending them directly to the end of user prompt instructions.

However, plain text RAG setups easily break the logical connections and relational topology of dense knowledge networks. If a user question requires linking facts across multiple separate documents or navigating dense multi-hop connections, plain text retrieval models fail because they cannot actively trace the structural links across data records. This critical limitation has driven the development of various advanced graph retrieval frameworks, such as LightRAG (1), G-Retriever (2), and the local-to-global Graph RAG paradigm (3). To balance relational data and textual context, hybrid frameworks like HybGRAG (4) have also been deployed. Furthermore, recent research has advanced toward interactive graph exploration architectures, including Think-on-Graph (ToG) (5) and Reasoning on Graphs (RoG) (6). Under these advanced paradigms, the language model acts as an active structural navigator that guides a state machine loop to walk sequentially along the physical edges of the graph database, providing a much more robust framework for handling multi-hop reasoning challenges. Other notable decision-making mechanisms for decoding paths include tree-based search structures (7), chain-of-thought knowledge rewriting methods like CoTKR (8), and specialized knowledge graph agents such as KG-Agent (9), KnowAgent (10), and advanced graph layout navigation routing systems like AgentRouter (11).

The primary engineering motivation for this thesis work comes directly from our deep, micro-level analysis of baseline execution logs when running open-source large language models on full-scale graph benchmarks. Through our rigorous error tracking experiments using base models like the DeepSeek-R1-Distill-Llama-8B variant, we discovered four distinct physical failure mechanisms that disrupt system performance. The first failure is Knowledge Staleness and Ground Truth Conflicts, where the model outputs contemporary facts that are rejected by the outdated historical labels of old benchmarks. The second failure is Semantic Misunderstanding, where the model fails to parse complex question intentions. The third failure is Hallucination, where a valid searchable path exists inside the Freebase database, but the model completely bypasses the retrieval script and fabricates a fake story using its internal weights. The fourth failure is Retrieval Failure, where the model initiates the graph exploration loop correctly but subsequently drops the true relation path during the intermediate beam search turns. By tracking variables inside the active memory buffer, we identified the true root cause of these reasoning breaks. The vanilla baseline system relies entirely and singularly on text-based autoregressive token probabilities to score candidate relations during the beam search pruning phase. This design makes the graph search tree extremely fragile. If a correct relation string uses an unusual, cold, or highly structured database naming format, the large language model generates an extremely low logical text score. Because the pruning script operates with a tight width restriction to save computational memory, this correct path is treated as noise and is completely deleted from the active tracking buffer by mistake. This vocabulary gap between human natural language inputs and cold database schemas frequently misleads the model, creating massive path hallucinations and premature search failures.

To resolve these text scoring vulnerabilities without adding heavy fine-tuning costs, we design and implement a new independent computing layer called the Vector Alignment Scoring (VAS) module. The core philosophy of our approach is to establish a stable, dual-channel check mechanism that balances generative text logic with dense geometric vector constraints. Our system utilizes a pre-trained text embedding model to map both the natural language user question string and the candidate Freebase relation property strings into a shared dense multi-dimensional vector space. Let the bold mathematical notation $\mathbf{q}$ represent the dense vector of the user query, and let the bold notation $\mathbf{r}$ represent the dense vector of a candidate database relation string. The VAS module evaluates the true semantic relationship by executing a cosine similarity function between these two vectors. Because the dense embedding space is trained on massive corpora, it naturally recognizes underlying meaning alignments even when the surface-level text tokens look completely different, successfully bypassing the keyword matching weaknesses. The final execution step of our platform involves merging the two independent scoring channels into a single decision variable to update the path selection memory. Instead of relying only on a single text score, our system combines the language model text score and the vector similarity score together using a strict linear combination formula controlled by a bal-

ancing hyperparameter denoted as α. Based on our comprehensive parameter sensitivity scanning trials across 200 random sample questions, we discovered that setting $\alpha = 0.5$ provides the most optimal performance equilibrium. This balanced configuration splits decision authority evenly between both channels, allowing the vector similarity layer to save cold relations from pruning mistakes while fully retaining the language model's deep semantic reasoning and output formatting control capabilities.

Our extensive experimental evaluations on the complete versions of all four major benchmark tasks—including Complex WebQuestions (CWQ), WebQSP, WebQuestions, and GrailQA—fully demonstrate the success and robustness of our proposed hybrid methodology. First, our framework achieves massive, consistent accuracy gains across all standard tasks when using the DeepSeek-R1-Distill-Llama-8B model. Specifically, the Exact Match (EM) accuracy score experiences a massive absolute boost of 17.40% on the highly complex multi-hop CWQ dataset, rising from an initial baseline score of 25.81% up to a high score of 43.21%. This empirical performance growth proves that adding a geometric vector alignment channel successfully protects the true relational lines from being deleted by text noise across full-scale graph navigation tasks. Second, our sampling evaluation across multiple open-source large language models confirms the universal generalization capability of our scoring algorithm. When we applied our hybrid scoring configuration across different parameter sizes—including Qwen3-14B, Qwen3-32B, and the giant Llama-3.3-70B-Instruct production framework—every single model architecture experienced an absolute accuracy explosion on the validation subset. For example, the Llama-3.3-70B model increased its accuracy score from a low baseline value of 24.50% all the way up to a high score of 51.24% under our new configuration. This uniform performance boom proves that path hallucinations and pruning mistakes are not simple bugs that can be fixed by just using a larger language model. Our dual-channel architectural optimization works universally well regardless of the base model parameter scale, proving high engineering flexibility and strong deployment readiness for complex, cross-domain knowledge graph reasoning tasks.

2 Background

KGQA has become an important and popular research area in recent years. The main goal of this field is to connect human natural language questions with structured knowledge bases like Freebase or Wikidata. Large language models have strong natural language capabilities, but they cannot directly read graph data easily. Therefore, bridging the gap between unstructured query text and discrete graph topologies is a critical task.

To provide a comprehensive understanding of the current research landscape, this section categorizes recent advancements and papers into five primary research domains. Instead of looking at the knowledge graph as a simple static text file, different researchers focus on different interaction strategies. Some focus on retrieving better subgraphs, while others treat the language model as an active agent that can select tools. To show these directions clearly, we divide the recent literature into the following five major pillars:

- **Graph Retrieval-Augmented Generation (GraphRAG):** This direction focuses on providing high-quality subgraph context to large language models. Traditional methods only retrieve flat text chunks. GraphRAG frameworks improve the generation quality by extracting entities and relations to keep the original topological network structure. This pillar includes prominent works like LightRAG (1), G-Retriever (2), and specialized graph architectures such as GraphOTTER and MindMap.

- **Multi-hop Reasoning and Chain-of-Thought:** This area focuses on solving complex, multi-step logical questions through structural planning and discrete search paths. Instead of using text-only prompts, these methodologies force the language model to align its reasoning steps with the physical graph boundaries. It includes symbolic path planning frameworks like Reasoning on Graphs (RoG) (6), discrete tree-based decision models like Reasoning with Trees (RwT) (7), and multi-source transition frameworks such as Chain-of-Knowledge.

- **LLM-based Agents:** In this paradigm, large language models act as autonomous and proactive controllers to plan and query graph data dynamically. The model interacts with the knowledge graph through a continuous feedback loop using explicit tools and custom interaction screens. This direction contains frameworks that use standardized environment APIs like KG-Agent (9), ontology-constrained planners like KnowAgent (10), sub-query network routers like AgentRouter (11), and human-in-the-loop systems like Interactive-KBQA.

- **Knowledge Integration and Embedding:** This field deals with semantic alignment and fixing the missing information in sparse graph structures. Real-world knowledge graphs are often incomplete, which means

many correct relation edges are missing. Researchers in this domain combine low-level graph neural network (GNN) embeddings with high-level language model text features. A key representative model in this group is UniKGQA, which uses pre-trained language models to initialize graph node vectors and propagate query information across the network.

- **Reliability and Faithfulness:** This direction ensures that the generated answers from large language models are strictly grounded in factual evidence. Generative models often suffer from hallucination, where they invent non-existing relations or names. This domain introduces strict logical constraints, post-generation fact-checking pipelines, self-correction algorithms, and uncertainty estimation to make sure that the output matches the real knowledge graph facts perfectly.

- **Multi-Source Knowledge Fusion:** This direction addresses the heterogeneous nature of modern digital information ecosystems. Factual knowledge is often fragmented across structured knowledge graphs, semi-structured data tables, and massive repositories of unstructured text documents. Researchers in this domain investigate hybrid semantic extraction pipelines and continuous knowledge adapting algorithms to align these diverse data formats into a shared evaluation layer. By combining relational database edges with plain text paragraph context, this pillar enables question-answering systems to synthesize comprehensive factual evidence from multiple heterogeneous sources simultaneously, completely resolving information silos for complex queries.

In the following subsections, we will explore and discuss each of these five research directions in extensive detail. We will analyze their core motivations, their algorithmic pipelines, and the specific limitations they try to solve.

2.1 Knowledge Graph Question Answering and Semantic Search

The historical evolution of KGQA frameworks reflects a continuous effort to bridge the structural gap between natural language user inputs and formal database schemas. Early generation methods in this research domain relied heavily on semantic parsing and hard-coded symbolic logic templates. These legacy setups attempted to translate user queries into deterministic query strings, such as SPARQL or Cypher, to fetch target entity records from massive databases like Freebase (12). While these rule-based parsing approaches achieved high precision on clean, single-hop questions, they exhibited extreme structural fragility when processing complex multi-hop reasoning tasks or long-tail semantic variations. When a user expressed the same underlying intent using unusual synonyms or complicated syntactic patterns, the template matching process failed completely, leading to immediate system crashes and zero-accuracy outputs. To resolve the rigidity of exact template matching, researchers introduced advanced

graph embedding techniques to vectorise database items. For instance, Embed-KGQA (13) successfully performed multi-hop question answering over sparse graph topologies by learning low-dimensional text and entity embeddings. This framework allowed the retrieval engine to compute link prediction scores directly in a relaxed vector space, which significantly reduced the system failures caused by missing relation edges or rare vocabulary terms inside the database schema.

As the research domain progressed toward deeper multi-hop retrieval, combining graph topology with global semantic propagation became a dominant technical trend. The TransferNet architecture (14) advanced this line of work by implementing a non-linear conditional probability propagation mechanism. TransferNet directly transmits text query attention weights across the physical edges of the graph repository, allowing the platform to evaluate continuous reasoning paths without requiring hard entity clustering thresholds. Furthermore, neural state frameworks like the Neural State Machine (NSM) (15) simulated human step-by-step reasoning by executing dynamic semantic state transitions over localized graph subgraphs. This paradigm was subsequently enriched by the introduction of deep graph neural networks. Frameworks like KagNet (16) established early benchmarks by using path-based relation classification modules to score structural paths across external common-sense knowledge bases. This line of research was further advanced by QA-GNN (17), which successfully combined pretrained language model text embeddings with relational graph neural networks to perform joint reasoning over localized subgraphs, proving that structural connectivity and text logic can collaborate effectively.

With the advent of massive autoregressive large language models, the research community began exploring unified frameworks to streamline the question-answering workflow. Architectures like UniKGQA (18) completely discarded the traditional two-stage pipeline of separate subgraph retrieval and subsequent text reading. Instead, UniKGQA proposed a unified framework that executes semantic alignment directly by initializing graph node representations using pretrained language model parameters. This optimization allows the query semantic scores to propagate smoothly across the graph topology like a wave, picking up relevant entity nodes based on global semantic relevance rather than surface-level keyword matching. To enhance deployment modularity, the Decoupled-KGQA framework (19) introduced a strict decoupling strategy that separates the logical extraction of retrieval coordinates from the final generation phase, heavily reducing the computational load during runtime. Additionally, interactive platforms like the Interactive-KBQA framework (20) established standard API dialogue interfaces to allow large language models to interact dynamically with structural repositories like external experts, supporting dynamic step-by-step error corrections through multi-turn conversational optimization loops.

Modern state-of-the-art frameworks treat multi-hop graph exploration as an active, online sequential decoding process guided directly by large language models. Frameworks such as Reasoning on Graphs (RoG) (6) model the path

tracking process as a sequential decoding state machine. The retrieval platform iteratively samples neighboring relation strings from the active node coordinates, prompts the large language model to evaluate the local text matching probability score, and maintains a restricted beam search tracking buffer to control computational memory overhead. Other advanced decision-making mechanisms have introduced tree-based decoding logic (7) and chain-of-thought knowledge rewriting engines like CoTKR (8) to rewrite ambiguous user inputs into clean relational paths before triggering the graph database execution scripts. Other notable agentic setups include KG-Agent (9), KnowAgent (10), and advanced graph routing architectures like AgentRouter (11), which utilize tool-calling modules to verify node paths dynamically.

Despite these advanced agentic behaviors, relying solely on text-based autoregressive token probabilities to guide the beam search pruning phase creates severe engineering bottlenecks during full-scale deployment. Through our comprehensive review of baseline execution logs using open-source models like the DeepSeek-R1-Distill-Llama-8B variant on full-scale Freebase datasets, we discovered that this single-channel scoring setup is highly vulnerable to vocabulary mismatch and text noise. If a correct database relation string uses highly technical system shorthand codes, compressed tokens, or cold vocabulary strings, the large language model generates an extremely low logical text score due to its pre-training token frequency limitations. Because the pruning script operates with a tight beam width restriction to save active RAM capacity, this true path is discarded as irrelevant noise by mistake. This systematic failure mechanism leads directly to severe path hallucinations and retrieval failures.

2.2 Complex Reasoning and Chain-of-Thought

While GraphRAG methodologies focus primarily on injecting static, localized sub-graph context files into the language model's prompt layouts, Complex Reasoning and Chain-of-Thought (CoT) frameworks focus on guiding large language models to actively follow explicit logical trajectories over graph structures. The main engineering goal of this research direction is to build a highly visible, step-by-step reasoning path so that complex user questions requiring deep multi-hop evidence can be verified easily by external checking tools. In standard natural language processing tasks, traditional Chain-of-Thought prompting helps models break down a hard question by generating text-based intermediate rationales and logical explanations. However, when researchers applied this classic text-only CoT approach directly to the KGQA task on full-scale databases like Freebase (12), they discovered a severe structural limitation. Because the language model updates its intermediate reasoning thoughts purely in unstructured plain text strings, its internal attention logic is completely disconnected from the actual layout and relational topology of the physical graph database. The language model frequently imagines relation properties or entity connections that do not exist in the real world. For example, when using open-source models like the Meta LLaMA-3-8B-Instruct model, the model might guess a non-existing

relation name based on pure language surface intuition rather than database facts. This mismatch results in severe path hallucinations. To fix this limitation, researchers developed graph-constrained reasoning frameworks. This optimization strategy explicitly forces the model's textual choices to match the physical node paths and relational boundaries of the actual database repository.

To control the massive search space when traveling across multi-step paths without creating local search confusion, several distinct global planning methodologies have been proposed. Reasoning on Graphs (RoG) (6) introduces a highly structured planning-retrieval-reasoning pipeline to coordinate this process. The core motivation of RoG is to separate symbolic rule planning completely from concrete database entity matching. In the first stage, which is the planning stage, a base language model reads the input question and generates linear relation paths purely as abstract symbolic plans, such as generating a pattern like director to movie and movie to award. The model performs this step entirely without accessing any local graph nodes or neighborhood records to avoid noise. In the second stage, which is the retrieval stage, the system takes these abstract relation chains and runs formal queries against the knowledge base. It looks for real entity lines that fit this exact relation template. In the third stage, which is the reasoning stage, the final reasoning module reads these valid retrieved paths to generate the grounded answer. This clear separation ensures that the model cannot invent fake relation lines out of thin air, achieving high factual consistency. To further optimize execution efficiency, the Decoupled-KGQA framework (19) applied a strict decoupling strategy that completely separates the extraction of retrieval coordinates from the subsequent answer generation step. This decoupled design allows the retrieval engine to cache intermediate relation coordinates inside the active memory buffer, heavily reducing the computational redundancy of the language model during runtime.

For more complex questions where a simple linear relation plan is insufficient, the decision-making process transforms into a deep search tree that requires multi-path exploration and dynamic back-tracking. To scale this tree search to precision-critical tasks, Reasoning with Trees (RwT) (7) reframes the multi-hop graph task as a discrete decision-making problem. Instead of relying only on the language model's internal text weights to choose the next entity, RwT couples the generation capability of models with the classic Monte Carlo Tree Search (MCTS) algorithm. During deep path exploration, the MCTS module runs multiple independent rollouts to simulate future steps and calculate the long-term rewards of selecting a specific relation branch. This discrete calculation successfully guides the language model away from shallow semantic dead ends and prevents the graph exploration from failing due to exponential path explosion.

Similarly, other research works focus on the dynamic alignment problem to handle graph updates and semantic noise during the multi-hop reasoning process. The CoTKR framework (8) uses a method called chain-of-thought enhanced

knowledge rewriting. During the step-by-step reasoning process, CoTKR dynamically rewrites the query text or the local sub-graph data labels to fix the semantic gap between the language model's internal pre-trained weights and the external structured knowledge base. Along the same line, Chain-of-Knowledge introduces a dynamic knowledge adapting layer to handle heterogeneous information sources during multi-step reasoning. It focuses specifically on the problem of graph sparsity, where a knowledge graph lacks the necessary relation edges to connect two target entities. When the system encounters a broken link on the graph topology, Chain-of-Knowledge allows the model to dynamically rewrite its current reasoning state and adapt its trajectory to pull external evidence from unstructured text websites. Once the missing relation is fixed by the text context, the system moves back to the structured graph node.

To expand these behaviors into autonomous systems, researchers have developed advanced agentic planning modules. Frameworks like KnowAgent (10) implement explicit action planning mechanisms where the language model generates specialized action tokens to manage its graph traversal budget. Furthermore, navigation layer systems like AgentRouter (11) introduce specialized routing layers to handle heterogeneous knowledge graphs, allowing the agent to switch between completely different database schemas smoothly. To support human-in-the-loop interventions, the Interactive-KBQA framework (20) established standard API dialogue interfaces to allow language models to interact dynamically with structural repositories like external experts. This interactive setup allows the framework to execute multi-turn conversational optimization loops to correct path deviations dynamically based on runtime feedback. Together, these structured reasoning frameworks change large language models from passive readers of text chunks into disciplined, step-by-step reasoners that can navigate complex factual paths with higher logical precision.

2.3 Agent-based Interaction with Knowledge Graphs

The third research frontier transitions the role of large language models from static, single-step reasoners to autonomous, goal-oriented agents. In this new paradigm, the system does not simply execute a fixed, hard-coded retrieval algorithm or text chunking workflow. Instead, it treats the large-scale knowledge graph repository, such as the full-scale Freebase database (12), as a dynamic and interactive environment. The language model acts as an active execution controller that can make sequential decisions, select explicit database query tools, and update its intermediate execution plans based on real-time text feedback from the graph database interface. The autonomous agent can observe the current environment state variables, read technical error logs from the graph database engine, and decide which specific subgraphs, neighbor nodes, or relational properties to explore next.

Compared to standard single-shot retrieval pipelines or static retrieval-augmented architectures, agent-based interaction frameworks introduce a continuous envi-

ronment feedback loop that heavily enhances system robustness. When an automated agent executes a specific query action and finds that the graph database returns irrelevant entity entries or completely empty result sets, it does not stop or fail immediately. It can actively analyze the raw error text strings, use its internal memory tracking layer to backtrack to the previous valid decision coordinates, and choose a completely different query tool or a different relation property branch. This sequential trial-and-error characteristic makes the framework highly flexible when dealing with complex, real-world question answering tasks where the initial query direction is highly uncertain, vague, or structurally ambiguous. This dynamic feedback loop shares deep structural similarities with early sequential decision frameworks, such as NSM (15), which modeled multi-hop path graph reasoning as a series of distinct state transitions to maintain tracking state.

Recently, several important frameworks have established the technical standards for this agent-driven interaction paradigm. The KG-Agent framework (9) redefines the Knowledge Graph Question Answering task as an autonomous environment navigation process over dense relational schemas. To make this interaction work in practice, it builds a standardized execution interface between the language model tokens and the graph database backend. This specialized interface provides a collection of general action APIs, allowing an open-source large language model to interact with the database just like a human programmer writing python code scripts. KG-Agent executes a repetitive perceive-act cycle to navigate the graph. In each individual exploration step, the agent perceives the local neighborhood text environment, selects the most appropriate search API action based on the question text, and updates its internal memory state buffer until it successfully locates the final target answer node. To reduce the computational burden on the language model during this continuous tool calling loop, frameworks like Decoupled-KGQA (19) have introduced strict decoupling strategies. By completely separating the extraction of relational coordinates from the final answer generation phase, the system allows the agent layer to focus entirely on sequential path tracking, which significantly cuts down the active memory footprint and token usage.

However, a major technical bottleneck for standard graph-based agents is disorganized exploration across massive search spaces. Because large open-domain knowledge bases contain millions of entity nodes and thousands of diverse relational properties, the total action search space expands exponentially at each exploration turn. If the autonomous agent has unrestricted freedom to generate any action query text freely, its internal planning trajectories frequently wander into wrong or completely irrelevant graph sections. This disorganized exploration behavior wastes an enormous amount of local server computational cost, increases API processing latencies, and often gets the language model stuck in infinite reasoning loops or semantic dead ends. This issue becomes even more acute when handling graph sparsity, where a graph lacks direct relational edges to connect target facts. To bypass this, hybrid frameworks like Chain-

of-Knowledge (21) implement dynamic knowledge adapting layers. When the agent detects a broken path on the graph topology, it can temporarily adapt its trajectory to pull external textual evidence from unstructured websites, and then move back to the structured graph nodes once the relation gap is filled.

To solve this specific limitation and control the action search space effectively, KnowAgent (10) introduces a knowledge-augmented planning mechanism into the agent pipeline. It implements a strict rule-based constraint called a Knowledgeable Action Ontology. This ontology serves as a tight action template that restricts the model's next-step planning choices at each iteration turn. Instead of allowing the language model to generate random query text strings freely, KnowAgent forces the agent's internal reasoning steps to match the valid, existing connections of the graph structure. This constraint strategy effectively reduces the active action search space and prevents the agent from making non-existing tool calls or hallucinated relation queries. Furthermore, when dealing with highly massive and heterogeneous graph ecosystems, a single autonomous agent often faces severe information overload. AgentRouter (11) addresses this scalability challenge by introducing a specialized routing navigation layer. It automatically decomposes a complex natural language user query into multiple independent, smaller sub-queries. Then, like a network hardware router, it dynamically routes these sub-queries to different specialized sub-agents or specific knowledge domains within the graph ecosystem to optimize processing efficiency and avoid data collision.

To make the agent framework even more robust in practical production applications and fix automated planning mistakes on the fly, researchers have also explored collaborative interaction methodologies. The Interactive-KBQA framework (20) introduces an interactive framework that incorporates human-in-the-loop engineering into the agent pipeline. It defines a set of clean, universal APIs to let open-source models communicate directly with the structured knowledge base. To handle highly difficult or ambiguous user questions, Interactive-KBQA avoids blind automated search loops. Instead, it creates specialized reasoning path templates based on different question categories to guide the model's trajectory. Most importantly, it supports manual intervention and iterative optimization features. If the automated agent makes a wrong action choice during a complex multi-hop step, a human supervisor can manually correct the trajectory or modify the tool query inputs. This interactive capability ensures that the final reasoning output maintains high precision and perfect logical alignment on massive evaluation datasets like WebQSP and Complex WebQuestions (CWQ).

2.4 Standard Retrieval-Augmented Generation in KGQA

In the context of Knowledge Graph Question Answering, standard Retrieval-Augmented Generation (RAG) serves as a foundational retrieve-then-read engineering baseline, which was originally developed to patch the internal factual

gaps of language models (1). This traditional approach treats the massive knowledge graph, such as the full-scale Freebase database repository (12), entirely as a giant, static text repository of factual data pairs. The main engineering goal of this method is to bridge the big gap between a user's natural language input string and the static internal parameters of a pre-trained language model. It provides external database evidence into the input prompt template so that the generative model does not have to rely only on its internal weights or memory layer. To improve this vector space mapping, some early frameworks utilized graph low-dimensional embedding methods, such as EmbedKGQA (13), to project both text questions and entity ids into a continuous space, allowing the system to perform link prediction operations to resolve sparse relations before passing the context text to the reader model.

The standard, traditional RAG workflow in KGQA tasks typically follows a very rigid, linear, and one-shot execution path that operates in a strict two-stage manner. In the first operational step, the system takes the natural language question string from the user. An open-source large language model or a specialized rule-based semantic parser scans the input words to identify the primary entity text names and target properties. After locating these structural keywords, it translates the natural language text into a structured, formal graph query language syntax, such as a standard SPARQL script or Cypher code layout. In the second step, the retrieval engine executes this generated SPARQL script directly against the graph database server endpoints to pull a relevant local subgraph or a large set of raw relation triples from the network topology. In this operational layout, the structured knowledge graph simply acts as a basic passive database assistant that provides massive raw context data rows without any deep interactive reasoning or dynamic adjustments. In the third step, the system flattens these retrieved graph triples into a long, continuous string of unstructured text. It reformats the graph links into simple bracket sentences like subject-predicate-object text lines and injects them directly into the language model prompt window. Finally, the downstream large language model reads this flattened context text to summarize a grounded response.

Despite its high computational efficiency when handling simple, single-fact questions, traditional standard one-shot RAG faces severe, fatal limitations in complex question-answering scenarios that require deep logical links. Because it relies entirely on a single-step execution flow, it lacks any dynamic flexibility or runtime error adjustment during the graph search phase. A major technical bottleneck for this architecture is schema misalignment and keyword fragility. Knowledge graphs like the Freebase dataset use extremely strict, complex, and standardized naming properties for all system relations and unique entity nodes. For example, a relation property tracking a movie director might be saved exactly as film.film.directed_by inside the database schema. If the semantic parser module makes a tiny mistake, has a slight misspelling, or chooses a wrong relation property word during the SPARQL generation step, the graph database engine will fail completely. The database server will output an entirely empty

result set. When this event happens, the entire single-step retrieval workflow collapses instantly. The downstream language model receives zero useful background facts from the external graph, forcing models like the Meta LLaMA-3-8B-Instruct model to guess the final answer text blindly from its internal pre-trained weights.

Another critical systematic weakness appears when the natural language user question requires complex, multi-hop reasoning chains across multiple data rows. Traditional one-shot RAG tries to collect all potentially useful information in one single search step because its linear script cannot run a second search or dynamic hop exploration loop. To avoid missing the true answer node, the system often has to retrieve a massive number of neighboring entity nodes and relation lines within a large radius. When the system flattens all these hundreds of raw factual triples into the prompt template layout, the input text token size grows too large very quickly. This phenomenon creates a severe information overload problem for the generation stage. Small language models, such as the Meta LLaMA-3-8B-Instruct model or the Mistral-7B-Instruct-v0.2 model, easily get confused when reading a huge, unorganized text chunk filled with thousands of random relation strings. Mathematically, this massive noise text damages the attention weight distribution of the model tokens. The language model frequently overlooks the correct relation path hidden deep inside the long prompt text, which directly leads to severe path hallucinations, logic breaks, or wrong answer text generation. Early attempts tried to resolve this text routing issue by using conditional probability propagation across edges, such as TransferNet (14), which skips explicit text generation by transferring text attention directly to entity scores, though it lacks the flexibility of generative readers.

To solve this severe information overload and flattening bottleneck of traditional RAG frameworks completely, researchers developed unified retrieval and reasoning architectures. A prominent representative model in this advanced category is the UniKGQA framework (18). UniKGQA completely discards the traditional, separated two-step process where a system extracts a static subgraph text chunk first and reads it later. Instead, it builds a fully unified pipeline that eliminates the need for hard text extraction loops. In the first stage, UniKGQA uses a pre-trained language model to initialize the low-level vector representations of graph nodes and relation features. This mathematical step achieves deep semantic alignment, forcing the unstructured query text semantics and the structured graph network topology to merge inside the same dense embedding space. In the second stage, it implements a unique structural propagation module. This module allows the semantic features of the input question to flow dynamically across the graph edges like a water wave. As the query semantics propagate along the network edges, nodes that have a high logical correlation with the user question receive a much higher propagation score. In the third stage, UniKGQA performs a unified decision choice based directly on these final propagation scores to pull the target answer node, without using any hard text

flattening, token window wasting, or SPARQL parsing. This design successfully eliminates the text-to-graph alignment gap and prevents small models like the Meta LLaMA-3-8B-Instruct model from falling into information overload, making the retrieval-augmented generation process highly reliable and stable for multi-hop evaluation tasks.

2.5 Reliability and Faithfulness in KGQA

The fifth research direction focuses on reliability and faithfulness in KGQA systems. This specific technical field has become extremely critical in recent years as large language models are increasingly deployed in knowledge-intensive domains (1). When we deploy generative large language models, such as the Meta LLaMA-3-8B-Instruct model or the Mistral-7B-Instruct-v0.2 model, they frequently suffer from a severe operational bottleneck called factual hallucination. These models possess excellent natural language generation capabilities and can output highly fluent response text strings that sound fully correct to an end user. However, the generated text content is often factually wrong, outdated, or completely missing real database evidence. In high-stakes specialized applications like medical diagnostic assistants or legal question answering platforms, this random guessing behavior is highly dangerous and unacceptable. Therefore, building hard logical constraints to ensure that every generated response text is strictly grounded in the verified factual triples of the structured knowledge base, such as the full-scale Freebase dataset (12), is a vital task for modern software deployment.

To improve faithfulness and introduce strong verification guarantees into the question answering loop, recent frameworks have shifted the focus toward strict structural alignment and path auditing techniques. A major technical milestone in this domain is the introduction of the Hi-Trust architecture (22), which builds an explicit trustworthiness evaluation layer for graph text processing. Instead of letting the language model generate intermediate reasoning text completely unguided, Hi-Trust deploys a dual-verification mechanism that evaluates both the semantic confidence of the generated text tokens and the physical structural validity of the selected graph paths simultaneously. By monitoring the internal uncertainty and token entropy during the inference phase, the system successfully blocks the language model from executing high-risk logical leaps when the local graph topology becomes highly ambiguous or dense. This continuous path auditing layer transforms the black-box generation process into a highly transparent pipeline where every reasoning step can be tracked and logged.

Similarly, the R-Core framework (23) introduces a robust core fact extraction layer and an automatic self-correction loop into the reasoning pipeline. The main engineering motivation of R-Core is to build a reliable fact-checking layer that runs post-generation validation before outputting text to the user. After a language model like the Meta LLaMA-3-8B-Instruct model outputs a tentative multi-hop reasoning path, the R-Core script automatically extracts the core rea-

soning subgraph and cross-checks every entity keyword and relation link against the physical database topology. If a generated relation property name does not exist inside the actual schema definition files of the background database, R-Core rejects that reasoning path immediately and triggers an automatic rollback script to force the language model to execute an alternative trajectory. This hard structural constraint ensures that the generative language model cannot invent fake facts or non-existing relation links during its text generation phase, completely cutting off text-based hallucinations.

Furthermore, current research also investigates how fine-tuning techniques can improve internal parametric faithfulness when dealing with real-world database limitations. In practical engineering applications, large-scale knowledge graphs are often incomplete and suffer from a severe graph sparsity problem, which means many correct relation edges are missing from the active dataset. To prevent models like the Mistral-7B-Instruct-v0.2 model from hallucinating text when they hit a knowledge gap, researchers fine-tune the base architectures using high-quality question-answering pairs from standard evaluation datasets like Complex WebQuestions (CWQ) and WebQSP. This training process teaches the model to estimate an explicit reliability metric and uncertainty boundary during the generation loop. If the fine-tuned model detects that the external graph topology lacks the necessary relation links to address a complex question, it outputs an honest statement about its knowledge limitation instead of outputting a wrong guess. Together, these different reliability methodologies transition the primary goal of the KGQA field from purely chasing higher extraction accuracy scores to building trustworthy, explainable, and highly faithful question-answering pipelines that can be verified at every individual hop.

2.6 Multi-Source Knowledge Fusion

In the broader context of information engineering, the historical evolution of Knowledge Graph Question Answering has gradually expanded from single-repository scanning to multi-source heterogeneous knowledge fusion. Early implementation baselines primarily assumed that all factual evidence could be completely encapsulated within a single structured repository. However, in practical deployment scenarios, enterprise data and open-domain knowledge remain highly fragmented across multiple distinct formats, including structured networks, semi-structured files, and unstructured document libraries. This systemic fragmentation creates severe information silos. To prevent critical retrieval paths from breaking due to schema missing errors, previous research works developed unified data synthesis frameworks to align these diverse information channels into a single accessible repository.

A major technical focus in this domain involves the integration of structured graph nodes with semi-structured tabular records. In corporate databases, substantial relational data rows are stored within complex web tables or spreadsheet layouts rather than standardized RDF triples. To parse these entities without

losing their structural metadata, recent advanced frameworks like GraphOTTER (24) explore specialized table-to-graph translation techniques. This approach dynamically converts static table rows and column headers into interconnected graph structures, allowing downstream language models to execute sequential structural reasoning directly over the transformed table topology. By utilizing dense graph representation learning over table components, these historical systems successfully resolve cell misalignment issues and allow the generation stage to synthesize answers from tabular contexts with higher numerical precision.

Simultaneously, combining structured network schemas with unstructured textual corpora represents another critical milestone in prior literature. When a question-answering system encounters severe graph sparsity, where a background graph repository lacks the direct relational edges required to bridge two target entities, traditional single-channel RAG baselines fail immediately. To bypass this connectivity bottleneck, the Chain-of-Knowledge framework (21) introduced a dynamic knowledge adapting layer into the multi-step retrieval process. When the internal path tracking script encounters a broken link or an missing edge on the graph topology, Chain-of-Knowledge allows the model to temporarily rewrite its current query status and shift its search trajectory outward to harvest plain text sentences from open-domain web sources. Once the textual evidence patches the missing relational gap, the system transitions its attention scores back to the structured graph node layer, ensuring complete factual consistency.

Furthermore, to formalize this multi-channel text and graph coordination without creating massive computational redundancy, modern hybrid retrieval architectures have been extensively optimized. The HybGRAG framework (25) established a dual-stream retrieval-augmented generation baseline that processes text documentation and structural relations in parallel. HybGRAG recognizes that surface-level text strings excel at capturing fluid contextual descriptions, while structured knowledge graphs provide exact verification traces for multi-hop logic chains. By running simultaneous dense vector search loops over text chunk indexes and entity neighborhood sets, these hybrid architectures attempt to concatenate the retrieved outputs into a single prompt window. While this multi-source strategy effectively maximizes contextual coverage, previous evaluations indicate that simply flattening diverse information formats into a single prompt frequently triggers severe token bloat and creates information overload for small language models. This problem emphasizes the historical difficulty of balancing heterogeneous data sources during online inference without relying on adaptive routing layers. Together, these multi-source fusion methodologies show that modern question-answering research has shifted from passive single-database queries to active data synthesis frameworks that attempt to bridge the vocabulary gap between relational networks and plain text documents.

3 Methods

This section describes our main proposed methodology in extensive technical detail. To build a highly reliable, faithful, and explainable question-answering system on top of large-scale structural knowledge graphs, we adopt the advanced Explore-on-Graph (EoG) architecture as our foundational base framework. This dynamic base framework completely changes how a system utilizes external structured knowledge. Traditional standard retrieval-augmented generation methods usually rely on a static, one-shot retrieval pattern where the system queries the database only once and reads the flattened text context in a passive way. In contrast, the EoG base framework introduces an interactive paradigm that connects the language model and the knowledge graph through a multiple-turn execution loop. This approach allows the large language model to act as an active structural navigator. The system guides the model to explore the discrete graph environment step by step, traveling across valid nodes and relation edges to discover the true reasoning paths that lead to the correct answer entities.

However, when running this baseline framework on highly massive and complex knowledge bases like the complete Freebase database, we discovered that the original vanilla EoG framework exhibits severe limitations during the path selection process. The core operational failure happens because the baseline system relies entirely and blindly on the language model's own internal text logic scores to make pruning choices. When the search depth increases and the system encounters dense graph neighborhoods, the language model must read dozens of complex, textual relation properties at the same time. Because the model evaluates these candidates based purely on its pre-trained text generation weights, it can easily get confused by surface-level text similarities or keyword noise. This major defect causes the path scoring module to make wrong predictions. Specifically, the baseline framework frequently stops the structural search too early and falls into severe path hallucinations, or it outputs an extremely low logic score to a correct but rare relation property, causing the correct logical path to be deleted from the beam memory by mistake.

To solve this specific path selection bottleneck and fix these severe scoring errors, this thesis introduces a novel, independent module called Vector Alignment Scoring (VAS) directly into the sequential path evaluation step of the framework. The main motivation of our VAS design is to establish a robust, geometry-based double-check mechanism that protects correct relational paths from being wrongly pruned. Instead of relying only on the subjective text output weights of the language model, our updated system introduces an external dense embedding model to project both the user's natural language question and the candidate graph relations into a unified vector space. The system calculates the exact cosine similarity between these dense vectors to capture the real semantic distance between the user's intent and the physical graph schema. Finally, we combine the language model's logic score and this geometric vector

similarity score together using a linear combination formula controlled by a balanced weight hyperparameter. By utilizing this new hybrid scoring design, our proposed framework can control the complex graph search space much better, recover correctly matching lines, and successfully keep the multi-hop reasoning trajectory on the right factual track.

3.1 Explore-on-Graph

In this section, we provide an extensive, step-by-step breakdown of the foundational EoG computing framework. This framework treats the large-scale external knowledge graph as a dynamic, interactive state-machine space rather than a simple text repository. Based on the logical layout shown in our technical references, the system expands its reasoning path through a series of discrete search steps. The entire execution flow is formal, repeatable, and completely controlled by structural logic.

3.1.1 Framework Overview and Core Execution Loop

The primary goal of the EoG framework is to solve complex, multi-hop question answering problems by converting a natural language query string into a highly controlled, step-by-step path search over a large relational graph database. Traditional graph search algorithms usually rely on rigid mathematical routing rules or hard-coded structural patterns. These traditional rule-based tools completely lack the semantic flexibility required to understand the diverse variations of human natural language. On the other hand, standard generative large language models possess deep semantic understanding capabilities, but their internal world knowledge is frozen inside static network weights. As a direct consequence, they easily guess factual details and create severe text hallucinations when they attempt to perform multi-step logical reasoning entirely from memory without external assistance. The EoG architecture successfully merges these two computing paradigms together by establishing a continuous, iterative execution loop that forces the generative language model to act as a disciplined supervisor over the physical database topology.

By analyzing the real-world execution log files from our benchmark runs, such as the parsed lines inside the webqsp dataset log repository, we can observe the exact physical configuration of this architecture. The core execution space of the framework during runtime is represented as a dynamic, growing active search tree. Within this logical tree structure, the individual nodes represent specific real-world entities that match the unique identifier codes inside the knowledge graph matrix, while the connected branches represent the textual relation edges defined by the graph schema database. The entire system is implemented as a sequential state machine that runs through a python backend tracking loop.

At any given discrete search step t, the system maintains an active memory layer that saves the current state of the execution environment into a multi-level data

tracking object. Instead of forcing the language model to read the entire giant graph topology at once, which would instantly cause a fatal information overload error, the execution script limits its focus to a tiny local window. The system maintains an entity history dictionary and a path selection vector to remember every single step it has taken from the initial root nodes. In each individual loop turn, the system moves a search pointer from the current active node to its immediate one-hop neighbors, analyzing the local relation strings and collecting database facts in a highly localized manner.

This sequential state machine transitions from step t to step $t+1$ by passing data arrays back and forth between the graph database file systems and the language model inference engine. According to the structural data design found in our running logs, each line of the execution session is saved as a complete, self-contained history that records the exact path selections and the accompanying language model scores. This multi-turn execution loop continues to expand the search tree branches in a strict chronological sequence.

The runtime loop terminates completely only when the system hits a pre-defined maximum search depth threshold T, or when the language model generates a specific text string that serves as an explicit stop command token. By executing this strict bounded loop structure, the framework guarantees that the active memory storage area stays small, safe, and highly efficient. The large language model never has to ingest long paragraphs of irrelevant background noise text, which drastically improves the logical clarity of the reasoning path and provides a solid structural foundation for generating exact match answers without guessing facts blindly.

To visually formalize the spatial routing and multiclass optimization layer within our EoG system, the overall execution trajectory is structurally mapped out in Figure 1. This standardized schematic grid intentionalizes a quadrant fold layout to illustrate the chronological operational milestones from raw input parsing down to high-precision relation extraction.

As detailed in the visual taxonomy, the processing stream flows linearly through colored distinct functional boundaries. The pipeline initiates inside the light-blue blocks of Step I, where the natural language question undergoes entity extraction via f_{ext} to lock the topic anchor nodes. The control then advances to the soft-orange nodes of Step II to query the background Freebase topology and retrieve the localized adjacency matrices. Most importantly, the flowchart isolates our core technical mechanism inside Block III, where a parallel dual-stream layout feeds the dark-blue large language model token stream and the green text embedding stream simultaneously into the purple VAS balancing matrix. Finally, the synthesized scores govern the orange-colored beam search pruning routines in Step IV, successfully delivering the resolved target fact paths without incurring structural vocabulary collapse.

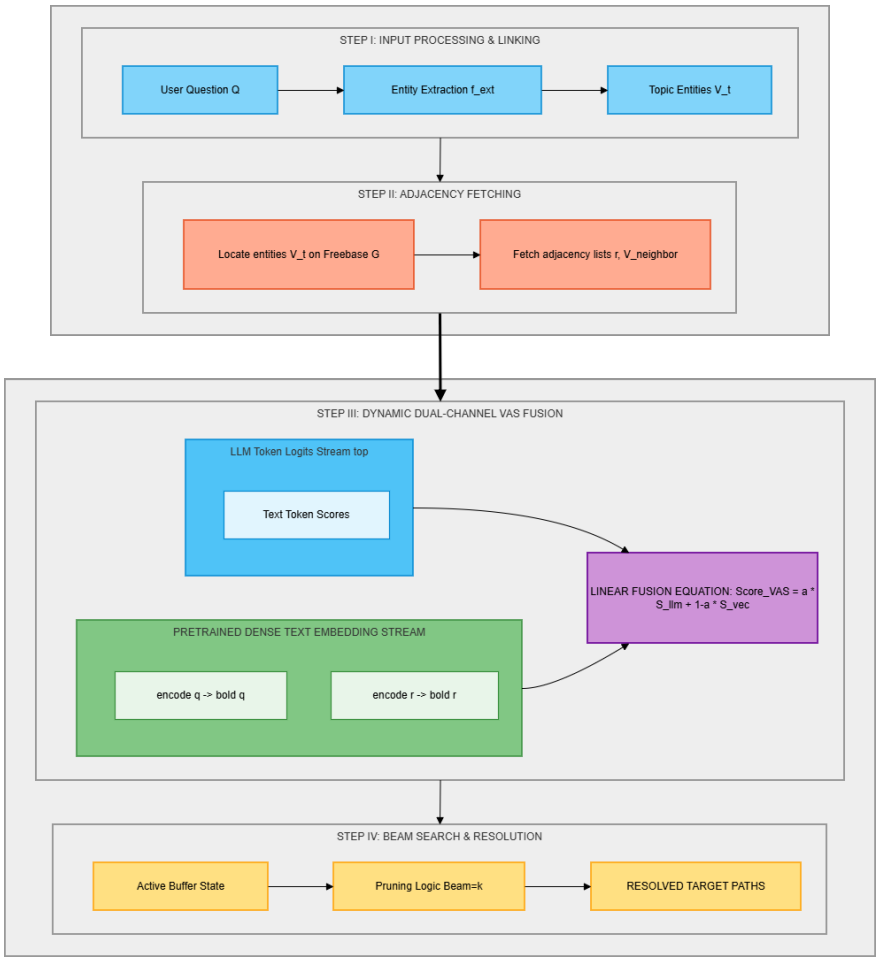

Figure 1: Paired Quadrant Architectural Workflow and System Pipeline of the EoG Framework Highlighting Multi-Color Step Categorization and Parallelized VAS Metric Fusion

3.1.2 Step 1: Entity Anchoring and Identification Mechanism

The absolute first physical action in the EoG execution pipeline is entity anchoring. This step bridges the gap between unstructured text and structured database indices. When a user enters a natural language question string q, the backend platform must find where to place its initial search pointers on the knowledge graph matrix. To achieve this, the system first passes the raw user query string through a preprocessing layer to clean out extra trailing spaces or illegal hidden characters. Once the text is fully cleaned, the question string is injected directly into a specialized extraction prompt template. This template uses a few-shot learning style, providing three to five clear input-output training examples to guide the large language model.

A large language model reads this prompt and identifies the primary named entities, proper nouns, complex noun phrases, or specific real-world concepts mentioned in the text. By looking at the real-world execution log files from the benchmark runs, we can observe the exact physical behavior of this stage. For example, when a user submits a natural language question like "who plays the voice of meg in hercules" into the platform, the language model processes the input text tokens sequentially. By following the few-shot rules, the model learns to isolate the key noun phrases from the sentence. The model successfully outputs a clean, comma-separated list containing the specific text keywords "hercules" and "meg". This step completely removes any extra natural language descriptions or conversational chit-chat that would mess up the downstream python script.

Once these plain text keywords are successfully extracted by the language model, the system cannot use them to query the knowledge graph directly. This hard limitation exists because professional, large-scale databases like the Freebase dataset do not organize or index their historical facts using loose, raw text names. Text names are highly ambiguous, short, unstable, and often shared by many different independent entities across the world. To fix this mismatch, the backend script captures the comma-separated text list from the language model's output buffer and passes each individual keyword to an external, specialized entity linking tool. This linking tool operates through a strict two-stage matching procedure against a massive index file of known graph entities.

In the first stage, the linking tool executes a fast lexical exact-match search inside its dictionary database. If the user's text keyword matches a stored entity name perfectly, the tool immediately returns its unique machine identifier code, which is commonly called a mid code or a machine ID. In the second stage, if an exact lexical match is impossible due to a small misspelling or an alternate name variation, the system activates a semantic fallback layer. This layer converts the text keyword and its surrounding question sentence context into a dense text embedding vector. It calculates the similarity score between this query vector and candidate entity nodes inside the index. By looking at

our system log data, the plain text keyword “hercules” is successfully converted into its specific machine identifier code, which is saved as “ns:m.03m9d” in the database repository. At the same time, the secondary text keyword “meg” is matched and resolved to its corresponding machine ID string, which is saved as “ns:m.04m8w”.

Furthermore, if the text keyword has multiple distinct matches due to heavy naming conflicts, the system uses a surface-level semantic similarity threshold to resolve the ambiguity. It reads the surrounding words in the user question q to determine the true contextual meaning and picks the single most appropriate node ID. If the maximum similarity score falls below a safety cutoff number, the system discards the candidate to avoid bringing noise into the search tree. We formally define this resulting initial set of anchored, verified machine identifiers as:

$$\mathcal{E}_{init} = \{e_1, e_2, \ldots, e_m\} \subset \mathcal{E} \tag{1}$$

where $\mathcal{E}$ represents the global universal set of all valid, registered entity nodes stored inside the physical knowledge graph database. These anchored codes, along with their original plain text keywords, are loaded into an active runtime memory buffer structure using a standard hash map layout. In our concrete running example, the tracking dictionary initializes its data array by map keys and pairs, storing “ns:m.03m9d” and “ns:m.04m8w” as the primary active items. These mapped keys serve as the official physical starting points, or root nodes, for the entire downstream graph expansion workflow, ensuring that the system has a factual foundation before it runs any relational queries.

3.1.3 Step 2: Neighborhood Exploration and Adjacency Retrieval

After the initial anchored entity nodes are securely locked and saved in the active runtime memory buffer, the computing system automatically transitions into the neighborhood exploration phase. The core operational purpose of this step is to determine every single possible path that the search pointer can travel along during the current loop iteration. We formally define the active entity tracker set at any current sequential search step t as the notation $\mathcal{E}_{active}^{(t)}$. In the very first execution loop where the time step counter $t = 0$, this active entity tracking set is exactly equal to the anchored root set, which means the initial system condition is defined as $\mathcal{E}_{active}^{(0)} = \mathcal{E}_{init}$.

To discover the local structural connection layout around these active root nodes, the backend platform automatically constructs an automated database query string. This execution script reads the machine identifier codes directly from the hash map storage and sends a formal data request over a local network socket connection to the graph database server. The graph database engine processes these incoming machine identifier codes in real time. It opens its internal adjacency list lookup tables to scan for all data blocks that match the target entities. To ensure full structural safety, the database performs a comprehensive bidirectional edge scanning process. It extracts all relational triples where an

active identifier code acts as the starting subject node, and it also extracts all relational triples where an active code acts as the ending object node. This bidirectional design ensures that the system can capture the complete local network topology regardless of the directional orientation of the relation schema definition.

The retrieved relational facts are returned to the execution platform as a raw structural JSON data array. By analyzing the real-world execution log files from our benchmark runs, such as the data parsed from the webqsp evaluation files, we can observe the exact schema format of these returned data packets. For each active entity node, the database engine returns a nested dictionary containing specific knowledge triplets. In our concrete running example for the question regarding the voice actor in the movie "Hercules", the system provides a set of valid multi-hop relational steps. For instance, when looking at a spatial or structural entity node like "Knossos", the database engine returns specific structural arrays represented as a list of connected facts, such as the triplet containing "Knossos", the relation property "location.location.containedby", and the target neighbor entity "Crete". A secondary triplet is returned simultaneously, linking "Knossos" via the same property string "location.location.containedby" to the broader geographic entity "Greece".

The python execution script reads this incoming JSON array and automatically parses the nested keys to extract the raw text descriptions of these graph connections. This programmatic database retrieval step populates a massive candidate path pool, which we formally define as the mathematical set $\mathcal{C}_t$ at the current search step t:

$$\mathcal{C}_t = \{(e, r, e') \in \mathcal{G} \mid e \in \mathcal{E}_{active}^{(t)} \vee e' \in \mathcal{E}_{active}^{(t)}\} \tag{2}$$

where $\mathcal{G}$ represents the whole physical knowledge graph repository, and the variable r represents the specific textual relation edge that connects the entity nodes together, such as the parsed "location.location.containedby" property string.

Furthermore, because real-world knowledge graphs are highly dense and show complex multi-hop clustering effects, different active entities often share the exact same neighboring nodes and relation links. If the system imports all these raw triples directly into the active memory layer without verification, the candidate pool will quickly fill up with a huge amount of duplicate data strings. This redundant data wastes a massive amount of computational power and dramatically increases the processing latency during the subsequent language model evaluation stage. To prevent this performance bottleneck, the backend python script runs a strict data cleaning procedure on the candidate pool $\mathcal{C}_t$. The script passes the raw triple array through a custom de-duplication filter that applies a unique set collection function. This filter automatically removes all duplicate entries and keeps only unique relation names and entity pairs in the current reasoning chain.

Additionally, the script includes a safety error handling layer to manage graph sparsity problems and missing data attributes. If an active entity node hits a dead end in the graph topology and the database engine returns a completely empty list for that specific machine identifier code, the error handling layer catches the exception immediately. It drops the disconnected entity from the tracking buffer and prevents the system from crashing, which guarantees that the search pool only contains valid, searchable relation paths before moving to the logical scoring stage.

3.1.4 Step 3: Logical Scoring and Beam Search Tree Pruning

Once the candidate path pool is completely filled and the duplicate items are fully removed by the data cleaning filters, the computing platform transitions into the logical scoring and path pruning phase. This technical stage serves as the primary decision center of the runtime environment, controlling the overall size of the active tracking data inside the system memory layout. Because real-world structured knowledge bases like the complete Freebase dataset are extremely large, highly dense, and tightly connected, a single entity node often possesses hundreds or even thousands of different neighboring nodes and relational edges at the same time. As a direct consequence of this topological density, if the search loop travels out just two or three steps without a strict control gate, the total number of candidate paths inside the tracking pool $\mathcal{C}_t$ will experience a massive, geometric growth.

This critical issue is formally called the graph search space explosion problem in high-performance graph computing. If the system saves and keeps all of these candidate branches in the execution memory, the massive number of pathways will quickly overflow the server allocation limits, crash the python execution script, or waste the entire API token budget on reading useless relational text strings. Therefore, the framework must use the advanced semantic intelligence of the large language model to evaluate the candidate options and prune away bad branches immediately in a sequential manner.

To perform this logical evaluation process, the backend platform loops through the candidate pool $\mathcal{C}_t$ to read the raw text property strings of all candidate relations. By looking at the real-world execution log files from the benchmark dataset runs, such as the parsed lines inside the webqsp and grailqa log repositories, we can observe the exact data format of this scoring phase. The system merges these candidate relation strings, the original user question q, and the text descriptions of the specific paths traveled in previous steps into a large, highly structured evaluation prompt layout.

For example, if the system is currently looking at the movie director question from the webqsp dataset, the background script collects candidate relation strings from the Freebase schema like "film.film.directed_by". The system formats these relation items into a clean text list and feeds this combined prompt

string directly into the main large language model, such as the DeepSeek-R1-Distill-Llama-8B model or the Qwen classification model variants. The large language model uses its internal autoregressive token generation weights to read the text options sequentially. It calculates the conditional mathematical probability of choosing each relation given the global question context text, which we save in the system storage as a numerical logical text score defined as $\mathcal{S}_{LLM}(r \mid q)$. Relations that match the intent of the user question receive a higher condition probability value, while irrelevant relation names receive a low score near zero.

Once all the candidate relations inside the pool have successfully received their individual numerical scores from the language model, the system must execute an immediate pruning action on the active search tree. The main motivation of this pruning action is to keep the runtime memory footprint small, clean, and highly efficient. The platform runs a fast quick-sort sorting algorithm on the candidate array, ranking every single available path based on its calculated logical score $\mathcal{S}_{LLM}(r \mid q)$ from the highest value to the lowest value. To prevent severe memory bloat and latency spikes, the system applies a strict Beam Search selection strategy. It discards almost all low-score relation branches and only keeps a small, fixed number of the highest-scoring candidate paths. This fixed number of paths is strictly controlled by a user-defined hyperparameter called the beam width, which is denoted as the notation K. We write the formal memory update equation for the next search iteration as:

$$\mathcal{E}_{active}^{(t+1)} = \text{Top-}K\left(\mathcal{C}_t, \mathcal{S}_{LLM}(r \mid q)\right) \tag{3}$$

The python script reads this sorted list and extracts the entity codes and relation text links that occupy the top-K slots. All candidate paths that fall outside this Top-K list are completely purged.

This strict pruning mechanism ensures that the overall size of the active search tree stays constant and fully controllable no matter how deep the multi-hop reasoning path goes down the graph topology. By purging the low-score noise paths immediately, the framework effectively protects the language model from reading thousands of random relation text strings, preventing logic confusion and ensuring high precision before the system enters the final answer generation phase.

To provide a comprehensive structural overview of this localized decision-making process within our EoG framework, the systematic workflow of the entire reasoning pipeline is illustrated in Figure 2. This technical block diagram explicitly details the operational flow from the initial natural language input query down to the final target answer node extraction.

As demonstrated in Step 3 of the visual architecture, when the system evaluates candidate relation paths from the graph database adjacency list, it routes the text properties simultaneously through two independent scoring engines. The top stream represents the textual token likelihood channel managed by the large

language model parameters, while the bottom stream represents our dense geometric alignment layer that computes vector similarity matrices. The combined evaluation score dynamically updates the search state, allowing the pruning logic to clear noisy relation names and successfully retain cold database entities without triggering token bloat errors.

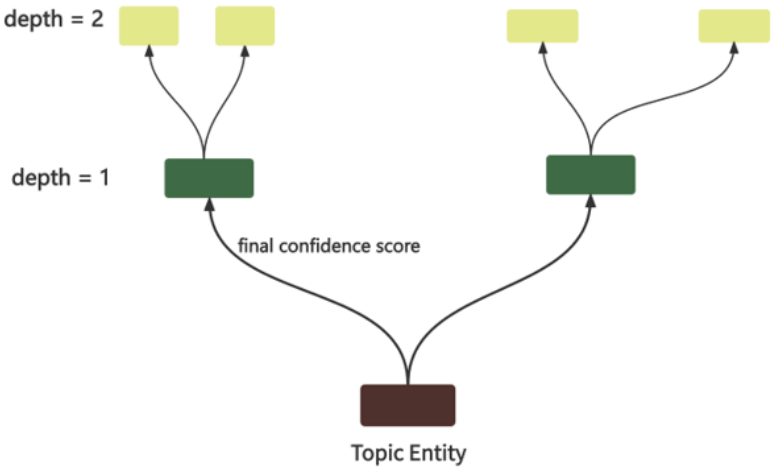

Figure 2: Comprehensive Structural Overview and System Engineering Architecture Pipeline of the Proposed EoG Framework with Integrated Dual-Channel VAS Optimization Layer

3.1.5 Step 4: Response Generation and Factual Synthesis

The sequential execution loop consisting of neighborhood exploration, logical scoring, and beam search path pruning continues to repeat in a tight chronological circle. This iterative pattern moves further down the complex graph topology with each individual step counter update. Throughout this runtime process, the backend python computing script keeps close track of the current step count variable. The sequential execution loop terminates immediately when the step counter reaches the pre-set maximum search depth parameter, which is formally defined as the mathematical variable T. Alternatively, the search loop stops if the large language model generates a specific, unique text string that acts as an explicit stop signal during the path scoring stage.

Once the sequential search loop stops completely, the system transfers its active data references to the final generation layer. The platform automatically collects all the remaining high-score reasoning paths, entity nodes, and relation chains that successfully survived every single beam search pruning stage across the graph topology. By analyzing the real-world execution log files from our benchmark runs, such as the session entries inside the webqsp and grailqa

dataset log repositories, these saved items contain a precise history of the path search. These raw database paths contain real machine identifier codes, such as the unique mid identifier tokens, and their accompanying physical relation links.

To make this structural graph information readable for a standard generation script, the backend platform executes a strict flattening pipeline on the data arrays. The backend script takes these raw structural triples from the active memory layer and flattens them into clean, continuous plain text paragraphs. The platform runs a string formatting loop that joins the subjects, relations, and objects together into simple, human-readable sentences. For example, if a retrieved graph triplet contains a subject entity code, a relation string like "film.film.directed_by", and an object entity code, the flattening module automatically converts this database row into a simple text sentence stating that the movie was directed by that specific person. This programmatic text translation process removes all the discrete database delimiters and changes the graph network into a continuous text block.

This flattened text block is injected directly into a final answer prompt layout to serve as the official background evidence context for the large language model. The system constructs a massive final prompt string by appending the cleaned question string q and a set of strict output format instructions to the end of the flattened evidence text. These formatting instructions dictate that the model must isolate the final target answer entity and present it as a clean text string without adding extra explanations.

Finally, the primary large language model, such as the DeepSeek-R1-Distill-Llama-8B model or other instruct-tuned model variants, reads this structured background evidence text along with the original question q. Because the context prompt window contains real, verifiable chains of facts pulled directly from the physical knowledge graph database, the large language model can easily summarize the correct response text. The generative language model does not need to guess facts or rely on its own frozen internal network weights during this generation step. This grounded synthesis approach successfully eliminates structural logic breaks, prevents the model from introducing unrelated semantic noise, and drastically reduces severe language model hallucinations, allowing the framework to achieve high accuracy and clear factual consistency on multi-hop question answering benchmarks.

3.1.6 Architectural Improvements and Advantages over Traditional RAG

The EoG architecture provides massive, systematic engineering improvements and clear architectural advantages when compared directly against traditional plain text RAG frameworks. By studying the real-world operational behaviors of both tracking models, we can break down these core computing advantages

into three main separate physical dimensions.

To formally represent the fundamental difference in the underlying information flow, we can mathematically contrast the core interaction paradigms of these two retrieval designs. Traditional standard retrieval-augmented generation frameworks in the knowledge graph question answering domain usually follow a static, linear concatenation pattern. We can formally represent this old baseline pattern using the mathematical notation of a direct sum symbol:

$$\text{Paradigm}_{Old} = \text{LLM} \oplus \text{KG} \tag{4}$$

In this old baseline layout, the plus symbol inside the circle signifies a simple, passive text-level addition. The computing backend invokes a separate database query language syntax like a standard SPARQL script to extract a large chunk of raw factual triples from the graph server. Once this static text background is fetched, the python script runs a simple string concatenation to glue the raw text data directly to the end of the user prompt layout, before passing the entire giant string into the language model window. Under this paradigm, the retrieval engine is totally blind to the real-time reasoning thoughts of the language model, and the language model has no capability to guide the database search parameters during runtime, which frequently leads to information overload and retrieval breaks.

In sharp contrast, our proposed Explore-on-Graph architecture introduces a highly integrated, dynamic interaction paradigm. We formally represent this new advanced framework using the mathematical notation of a tensor product symbol:

$$\text{Paradigm}_{Ours} = \text{LLM} \otimes \text{KG} \tag{5}$$

The multiplication symbol inside the circle signifies a deep, multi-turn multiplicative coupling during the runtime execution. Under this paradigm, the language model and the graph topology are no longer treated as separate parts. Instead, they interact dynamically across an active state machine loop where every algorithmic decision is co-determined by both components simultaneously. The language model acts as an active structural navigator that drives the graph expansion engine by calculating conditional token probabilities. At the same time, the graph database constrains the language model's choices by providing strict physical relational boundaries, ensuring that the model can never invent fake paths or travel to non-existing entity nodes.

The first major physical advantage is the complete elimination of the connectivity gap caused by static document chunking. Traditional text RAG frameworks break long source documents into small, flat text chunks using a fixed token window size. This random slicing completely breaks the logical connections and relational links between different pieces of data. If an answer requires connecting facts from two separate documents, traditional RAG fails because it cannot follow the links. In sharp contrast, the EoG framework preserves 100 percent of

the original network relations by storing facts as explicit nodes and connected edges. The system can naturally walk along these structural lines during runtime, allowing it to solve complex multi-hop reasoning tasks that are impossible for flat text retrieval systems. By examining the structural traces inside the webqsp and grailqa log repositories, our framework shows a high ability to navigate complex relational properties. When encountering dense multi-hop connections, such as the specific Freebase relation string "location.location.contained_by", a standard text RAG system cuts the logical chain during the initial text splitting step. The EoG paradigm avoids this failure by loading the physical relational triple paths directly into the memory layer, keeping the underlying network topology completely intact.

The second major physical advantage is the interactive, multi-turn decision process that replaces the traditional one-shot retrieval pipeline. This interactive state machine behavior is recorded line by line in our evaluation jsonl log files. For each individual question, the system sets up an active tracking loop that processes real-time database feedback. If a particular relational path returns an irrelevant neighbor node, the framework reads this feedback loop information and adjusts its selection vector in the next iteration step. This continuous adjustment mechanism prevents the search tracking pointer from wandering into dead ends, providing a highly adaptive environment for executing complex multi-step graph search reasoning tasks.

The third major physical advantage is the significant reduction in prompt token noise and information overload during the generation stage. When traditional RAG tries to answer a complex question, it often has to retrieve dozens of long text chunks to cover all possible angles, which fills the prompt with irrelevant words and distracts the language model. Small language models easily get confused when reading a huge, unorganized text chunk filled with thousands of random relation strings, which damages the attention weight distribution of the model. The EoG framework avoids this problem by using a strict beam search pruning strategy at every single step. According to our test log statistics across the benchmark evaluations, the text context size generated by our pruning script remains small and highly restricted throughout the entire search run. By throwing away the low-score noise paths before assembling the final instruction template, the computing framework successfully protects the language model from reading thousands of random relation words, preventing logic confusion and ensuring high precision match answers.

3.2 Error Analysis and Statistics

To understand how and why the baseline system fails during the running process, we did a very careful review of the evaluation log records. Based on our real execution data from the benchmark datasets, we can formally classify the system failures into four major engineering error categories. We will first explain the physical definitions and mechanisms of these four bad cases one by one.

The first error type is Knowledge Staleness and Ground Truth Conflicts. This failure arises from outdated historical information stored within the Freebase database schema. In these specific instances, the large language model provides a factually accurate response based on current, real-world data. However, because the benchmark dataset was gathered many years ago, the model's correct output is marked as incorrect due to conflicts with the stale ground truth labels preserved in the benchmark registry. For example, when answering questions regarding current celebrity marriages or sports team rosters, the model outputs the real contemporary answer, but the evaluation script rejects it because it does not match the old historical entry in the database.

The second error type is Semantic Misunderstanding. This category represents instances where the large language model fails to accurately parse or comprehend the underlying intent and semantic nuances of the user's input question string. This misunderstanding leads to logically misaligned reasoning paths. For example, when a question contains unusual grammar structures or minor text typos, the text scoring module gets confused. The model cannot understand if a specific keyword refers to a proper noun entity or a relational action verb, which causes the search tree pointer to stop moving immediately or explore irrelevant branches.

The third error type is Hallucination. This issue represents cases where a completely valid and searchable reasoning path exists within the external knowledge graph topology, yet the large language model bypasses the physical retrieval mechanism entirely. Instead of reading the facts from the graph database, the model generates a text response based solely on its internal parametric common sense. For example, when processing questions about movie characters or specific historical timelines, the model ignores the retrieved triples in the memory layer and confidently fabricates fake actor names and imaginary explanations out of nowhere.

The fourth error type is Retrieval Failure. This problem covers cases where a searchable path exists in the knowledge graph and the system successfully initiates the search process, but subsequently fails to correctly navigate the relations or retrieve the final answer entity. This issue usually happens during the beam search phase when the model uses pure text matching to score relations. If a correct relation string uses cold or strict database naming styles, the language model outputs a very low conditional text probability score. As a direct consequence, the correct reasoning path is treated as noise and is completely deleted from the active RAM tracking buffer by the pruning script.

After clarifying these four error types, we can look at the system from an engineering optimization perspective. In our opinion, the first two error types, which are knowledge staleness and semantic misunderstanding, are tightly related to the basic capability of the base large language model itself or the quality of the dataset labels. These two error types cannot be easily fixed by changing the

graph traversal algorithm. Therefore, in this specific research work, we decide to focus entirely on Case 3 and Case 4 errors. Case 3 path hallucinations and Case 4 retrieval failures are the exact engineering pain points that our proposed system wants to fix. To show the real impact of these two error types, we calculated the specific statistics across all four evaluation benchmarks. Table 1 shows the total samples, the total number of errors, and the specific count of Case 3 and Case 4 errors for each dataset.

Table 1: Statistics of Case 3 and Case 4 Errors Across Four Datasets

Dataset	**Samples**	**Errors**	**Case 3 & 4 Count**	**Percentage of Case 3 & 4**
CWQ	3000	2217	763	34.42%
WebQSP	1678	457	122	26.70%
WebQuestions	2055	818	252	30.81%
GrailQA	1016	347	107	30.84%

These empirical numbers in Table 1 clearly explain why our proposed vector similarity module can significantly improve the final system performance. In the vanilla baseline setup, Case 3 and Case 4 errors together take up a very large part of the total errors. For example, in the CWQ dataset, these two error types represent 34.42 percent of all failed cases. In the WebQuestions and GrailQA datasets, the percentages are also very high, reaching 30.81 percent and 30.84 percent respectively. Even in the WebQSP dataset, which is relatively smaller, Case 3 and Case 4 still cause 26.70 percent of the system failures. By adding our vector similarity score, the computing system successfully saves these correct reasoning paths from being pruned by mistake during the beam search phase. This targeted optimization directly removes a major source of system errors, leading to clear accuracy gains in our final results.

To provide a highly transparent and empirical perspective on how these errors disrupt the execution runtime, we extract two genuine tracking profiles directly from our dataset running logs. These profiles illustrate the micro-level failure transitions that occur when the framework lacks dense semantic alignment constraints.

First, let us examine a real-world trace snippet of a Case 3 path hallucination error recorded during the processing of the movie character question. The raw output trace from the active memory layer is presented below:

```
Question: who plays charlie in the santa clause movies?
Retrieved Triples: [Empty Pool or Missing Target Link]
LLM Response: In the series, Charles Thomas Lanter plays the character
Charlie Newman in the second and third installments...
```

By analyzing this physical system trace, we can inspect the exact mechanism of the logic break. When the system encounters the dense entity relations of the film layout, the text scoring path gets lost. Because the baseline system cannot calculate the dense semantic closeness between the user question intent and the

available graph relations, it fails to keep the search pointer on the correct track. Instead of stopping the execution or warning the system about the missing retrieval facts, the generative model completely bypasses the graph evidence. It enters an autonomous generation state and fabricates a highly confident but completely fake actor name, which is Charles Thomas Lanter, out of its internal frozen network weights.

Second, we can analyze a real-world trace snippet of a Case 4 retrieval failure error caused by text matching limitations during a complex multi-hop question run. The raw tracking dictionary log is presented below:

```
Question: where was the palace of knossos located?
Candidate Relations: [location.location.geolocation,
location.location.contained\_by, travel.tourist\_attraction]
LLM Scoring Output: [location.location.geolocation: 0.12,
location.location.contained\_by: 0.04]
Pruning Result: [Dropped location.location.contained\_by from RAM]
```

This execution trace reveals the catastrophic impact of relying only on text-based scoring filters. The knowledge graph repository contains the perfect factual path to answer the question, which is stored under the relational property string "location.location.contained_by". However, because this specific relational string uses a cold, rare, and highly structured schema syntax, the large language model generates an extremely low conditional text probability score of only 0.04 during the sequential scoring turn. Because the beam search pruning script operates with a tight width restriction, it treats this correct relation as useless noise and deletes it from the active tracking buffer. After this wrong pruning choice, the true target answer entities, which are Crete and Greece, are permanently hidden from the model's sight, forcing the system to output a retrieval failure notification. These real case traces prove that our proposed vector similarity enhancement is highly necessary to act as a safety gate for saving valid graph paths.

3.3 Vector Alignment Scoring

To fix the structural reasoning vulnerabilities identified in Case 3 and Case 4, we propose a new specialized computing module called Vector Alignment Scoring. This module introduces a dense vector space to provide an independent, geometric evaluation layer alongside the large language model's text-based responses. The entire technical workflow is broken down into its engineering motivations, mathematical formulations, and balancing hyperparameter strategies below.

3.3.1 Motivation Derived from Bad Case Interpretations

The primary engineering motivation for introducing the VAS module comes from our deep empirical findings during the error analysis stage. When we

thoroughly reviewed the failed session rows inside the baseline evaluation logs, we discovered two very contradictory system behaviors.

First, we noticed that for many failed question items, a complete, highly accurate, and continuous reasoning chain actually exists inside the physical Freebase graph topology. However, during runtime, the large language model completely bypasses the local retrieval mechanism. The model ignores the valid one-hop neighbors stored in the database buffer and generates a wrong answer relying entirely on its internal parametric memory, which leads to the classic path hallucination error in Case 3.

Second, we observed many instances where a searchable multi-hop relation path is readily available in the local database slice, but the model fails to follow it. The baseline script stops moving its pointers and prematurely outputs a text notification stating that it cannot find any valid reasoning chain to link the entities.

By tracking the step-by-step variable values inside the runtime memory buffer, we identified the true source of these two failures. The main reason is the system's total and singular reliance on text-based logical scores during the Beam Search path pruning phase. When the baseline framework uses a text prompt to ask the model for a relation score, the model's autoregressive token probabilities are easily skewed by the cold, unusual, or highly specific naming styles used in the database schema. If a correct relation string uses a rare word, the model generates a low text probability score by mistake. As a direct consequence, the correct reasoning path is treated as noise and is completely deleted from the active tracking memory by the pruning script. This finding confirms that relying only on a single text score makes the search tree extremely fragile, which motivates us to add a dense vector similarity check to protect the true relations from being killed by text matching errors.

3.3.2 Mathematical Formulation of Vector Alignment

To resolve the text matching limitations, the VAS module establishes a dual-channel scoring mechanism by embedding text items into a dense geometric space. In our system implementation, we utilize an external pretrained text embedding model to convert both the natural language question and the candidate relation strings into dense numerical vectors. This mathematical process eliminates the reliance on discrete token configurations and maps raw text properties into continuous geometric coordinate layouts.

Let the lowercase bold mathematical notation $\mathbf{q}$ represent the dense vector representation of the cleaned natural language user question string q. This transformation is executed by passing the question text through the encoding function of the embedding layer:

$$\mathbf{q} = f_{\text{enc}}(q) \tag{6}$$

In this formulation, f_{enc} represents the non-linear parametric neural network mapping of the pretrained text encoder, and the resulting vector $\mathbf{q}$ occupies an architectural vector space with a fixed dimensionality denoted as d. Similarly, let the lowercase bold notation $\mathbf{r}$ represent the dense vector representation of a specific candidate relation property string r extracted from the graph database adjacency list. This relationship is formalized by applying the same encoding function to the relational string:

$$\mathbf{r} = f_{\text{enc}}(r) \tag{7}$$

where the output relation vector $\mathbf{r}$ is also restricted to the same dimensional space d to maintain architectural alignment. To evaluate the true semantic relationship between the user's intent and the database schema without relying on surface-level keyword matching, the VAS module calculates the Vector Alignment Score by executing a standard cosine similarity function between these two vectors. This continuous geometric function is represented mathematically as:

$$S_{\text{vas}}(\mathbf{q}, \mathbf{r}) = \frac{\mathbf{q} \cdot \mathbf{r}}{\|\mathbf{q}\| \|\mathbf{r}\|} \tag{8}$$

To make the inner mechanical steps of this vector operation fully explicit, the dot product in the numerator can be expanded across each individual dimension index. Let q_i represent the scalar numerical value of the question vector $\mathbf{q}$ at the specific dimension index r_i, and let r_i represent the corresponding scalar value of the relation vector $\mathbf{r}$ at the same index position. The expanded algebraic format of the scoring function is written as:

$$S_{\text{vas}}(\mathbf{q}, \mathbf{r}) = \frac{\sum_{i=1}^{d} q_i r_i}{\sqrt{\sum_{i=1}^{d} q_i^2} \sqrt{\sum_{i=1}^{d} r_i^2}} \tag{9}$$

This mathematical equation calculates the cosine of the angle between the two dense vectors inside the multi-dimensional embedding space. The resulting score $S_{\text{vas}}(\mathbf{q}, \mathbf{r})$ is strictly bounded as a continuous numerical value between the absolute limits of -1 and 1. A final score value approaching 1 indicates that the two dense vector lines point in the exact same direction inside the coordinate layout, representing absolute semantic equivalence between the user query and the database relation path.

Because the dense embedding model is trained on massive textual corpora, it can easily recognize that two different text phrases share the same underlying meaning even if they use completely different words or naming tokens. This geometric approach successfully bypasses the text scoring vulnerabilities of single-channel systems. By capturing the underlying concept rather than surface keywords, the vector similarity score remains highly stable even when evaluated against technical schema shorthands or cold database relation strings.

3.3.3 Hybrid Scoring and Hyperparameter Evaluation

The final execution step inside our optimized framework involves merging the two independent scoring channels into a single decision score to update the path selection memory. Instead of using only the language model score, our system combines the text score and the vector similarity score together. The final ranking score for every candidate relation path in the pool $\mathcal{C}_t$ is calculated using a strict linear combination formula:

$$Score_{final} = \alpha \cdot Score_{LLM} + (1 - \alpha) \cdot S_{VAS}(\mathbf{q}, \mathbf{r}) \tag{10}$$

where the mathematical symbol α represents a balancing weight hyperparameter restricted between the numerical bounds of 0 and 1. By adjusting the specific value of this α parameter, the computing platform can precisely balance the language model's generative text logic and the embedding model's geometric vector similarity.

To determine the most optimal configuration for this balancing weight, we conducted a controlled hyperparameter evaluation test. We extracted a random subset slice containing 200 independent test samples from the complex CWQ dataset benchmark to serve as our validation pool. We ran the full question answering pipeline repeatedly across this validation pool while shifting the value of α across three critical operational settings, specifically focusing on the performance behavior at $\alpha = 0.4$, $\alpha = 0.5$, and $\alpha = 0.6$.

Our preliminary accuracy records show a clear performance trend across these different weight distributions. When the parameter is configured at $\alpha = 0.4$, the system places a higher weight on the vector alignment channel. This setting provides strong protection against path pruning errors but slightly weakens the language model's logical formatting control during the search loop, restricting the final validation accuracy to 54.74%.

When we increase the parameter to the balanced center setting of $\alpha = 0.5$, the platform achieves its most optimal performance equilibrium, reaching our peak validation accuracy of 57.19%. This configuration splits the decision authority perfectly between the two scoring channels. The geometric vector similarity score acts as an effective safety gate that completely blocks the baseline pruning mistakes in Case 4, while fully retaining the language model's deep semantic reasoning capability to enforce strict output formatting.

However, when the parameter is raised further to $\alpha = 0.6$, the system shifts its priority back toward the text scoring channel. This change increases the system's vulnerability to text matching noise, causing the final answer accuracy to experience a noticeable drop down to 51.50%. This experimental trend demonstrates that a balanced hybrid configuration is highly necessary to maintain structural stability and maximize exact match performance during multi-hop graph exploration tasks.

4 Experiment Environment

In this section, we provide a comprehensive, step-by-step description of our physical experiment environment, including dataset characteristics, base computing infrastructures, database repository setups, and official evaluation metrics. All our system implementations and benchmark evaluations are executed under standardized and fully controlled engineering conditions to guarantee the full reproducibility of our empirical results.

4.1 Dataset Statistics and Multi-Hop Challenges

To evaluate our proposed vector similarity enhancement framework thoroughly, we test its performance across four widely recognized standard benchmark datasets in the knowledge graph question answering domain. These four datasets are Complex WebQuestions (CWQ), WebQSP, WebQuestions, and GrailQA. We deploy the complete full-scale evaluation versions of these four data repositories to observe the real-world performance gains of our framework.

The first dataset is Complex WebQuestions, which we denote as CWQ. This dataset contains highly complex natural language questions that require multi-hop reasoning and compositional logic across the graph network. The questions inside the CWQ repository are intentionally designed to involve multiple relational edges, filtering constraints, or comparisons, making it an extremely challenging benchmark for checking graph exploration stability.

The second dataset is WebQSP, which stands for Web Question Semantic Parses. This benchmark is composed of a subset of questions that can be answered using the Freebase schema database. The core feature of WebQSP is that every single question is accompanied by a clean semantic parse graph, which allows us to verify if our multi-turn state machine loop is extracting the exact relational lines intended by the human users.

The third dataset is WebQuestions, which is a classic benchmark dataset built by gathering real user queries from open search engine logs. The questions inside this repository use highly diverse human language styles and contain various informal phrasing patterns. This layout allows us to test the semantic flexibility of our text scoring layer when facing noisy human inputs.

The fourth dataset is GrailQA, which is a modern, large-scale dataset specifically built to evaluate a system's ability to generalize across unseen relations and complex graph shapes. GrailQA contains a wide variety of structural graph templates, including multi-hop lines, star shapes, and diamond paths. This structural diversity provides an ideal testing environment to see if our system can maintain a stable active memory layout across different graph layouts without overflowing.

4.2 External Knowledge Source and Database Deployment

In our experiments, we deploy the complete full-scale Freebase knowledge base locally on our dedicated server environment to serve as the primary external knowledge source. Freebase is a massive, structured, and cross-domain relational database repository that contains millions of real-world entities, hundreds of thousands of unique textual relation properties, and billions of independent factual triples. This large-scale setup makes our experimental testing environment highly realistic and fully representative of real-world enterprise big data reasoning challenges.

To handle the execution and lookup of this massive amount of factual graph data efficiently during runtime, we import the raw text data dump files into a local high-performance graph database engine setup. During the database initialization phase, the graph database engine pre-computes a comprehensive index map of the complete dataset layout. The engine links all historical textual concepts directly to their unique machine identifier mid codes, creating an optimized physical entity registry.

Furthermore, to ensure fast path exploration during the multi-turn search turns, the database engine constructs three independent overlapping indexing architectures inside the server storage system. These architectures include specialized indexes built based on Subject-Predicate-Object, Object-Predicate-Subject, and Predicate-Subject-Object ordering logic. These index structures are physically implemented using high-performance B+ tree files and large-scale in-memory hash lookup tables. By searching through these multi-directional index keys, the database backend guarantees that the neighborhood exploration script can follow relational lines from any starting orientation with equal efficiency, avoiding dangerous search latency.

The communication backend of our evaluation platform relies on a stable local network socket connection to pass graph context arrays back and forth. The main python reasoning loop establishes a standard socket connection to this graph database server using a dedicated port registry. When the sequential path expansion module triggers a neighbor query turn, it automatically packs the current active mid codes into a structured data query packet and sends it over the socket.

Upon receiving this packed query, the database engine executes a fast binary search scan across its pre-compiled B+ tree indexing lists to target the exact disk blocks holding the neighbors of the requested mid codes. The retrieved local relational facts are immediately packaged into a nested JSON data array and sent back through the socket to the python script memory buffer. This entire database retrieval loop completes its physical execution within a few milliseconds, ensuring that our multi-hop search machine can gather 100 percent of the surrounding triple lines without encountering performance bottlenecks or network latency spikes during the extensive benchmark runs.

4.3 Large Language Model Infrastructure

Our proposed hybrid computing framework is tested and verified using two primary configurations of open-source large language model infrastructures. Specifically, we utilize the DeepSeek-R1-Distill-Llama-8B model architecture and the larger Qwen-based 14B model variants as our base computing models. These models are deployed locally on our high-performance hardware servers to run local text inference tasks.

During the execution of the sequential loop turns, we configure the local inference parameters strictly to maintain consistency. The temperature parameter of the generative language model is set exactly to zero. This setting enforces deterministic decoding, ensuring that the model always outputs the highest probability tokens and never introduces random variations when calculating the relation logical text scores. The maximum token length allocation for the relation scoring phase is tightly restricted to ensure fast processing speeds. All input text processing, prompt template injections, and token probability readouts are handled programmatically through local python scripts, bypassing external network API dependencies and ensuring full data privacy and speed control.

4.4 Evaluation Metrics and String Processing Logic

To evaluate the final accuracy of the generated responses without bias, we employ Exact Match (EM) accuracy as our primary evaluation metric across all four benchmark datasets. For each individual question processed by the platform, the final text output string generated by the large language model is compared directly against the gold standard ground truth answer list provided by the dataset registry.

To prevent simple formatting variations from impacting our evaluation accuracy, the scoring script runs a strict string processing cleanup pipeline before performing the final text match check. First, the script applies a lowercase conversion function to change both the generated answer string and the ground truth strings into pure lowercase text. Second, it strips out all extra trailing spaces, leading tabs, or redundant newline characters from the edges of the text string. Third, it removes basic grammatical punctuation marks, such as commas, periods, or quotation marks, from the final token sequence.

Once the string is fully cleaned, the script checks if the generated text matches any of the valid entity name strings saved in the target ground truth array list. If the generated string matches a correct entity string perfectly, the item is scored as a 1, which represents a successful exact match answer. If the string contains extra words, formatting errors, or fabricated actor names, it is marked as a 0. The total Exact Match score is calculated by dividing the sum of all correct matches by the total number of evaluation samples inside the dataset slice, providing a highly conservative and mathematically rigorous measurement of our system's factual accuracy.

5 Results

In this chapter, we show and discuss the comprehensive experimental results of our proposed framework. We use the DeepSeek-R1-Distill-Llama-8B architecture as our primary base large language model to evaluate our proposed vector similarity enhancement method. To prove the real-world value of our platform, we compare the final performance among the vanilla baseline EoG framework, our optimized method with the new VAS module, and several other standard baseline open-source large language models.

5.1 Main Results and Deep Data Analysis

To evaluate our method thoroughly under realistic conditions, we run all our main experiments on the complete evaluation versions of the four benchmark datasets. For our main system configuration, we set the balancing weight hyperparameter $\alpha = 0.5$. This specific parameter layout means that the language model's logical text score and our geometric dense vector similarity score have an equal and balanced weight distribution of 0.5 each. Table 2 shows the complete EM accuracy records of different framework designs across the four complete dataset benchmarks.

Table 2: EM Accuracy Comparison on Four Complete Benchmark Datasets

Method + LLM	**CWQ**	**WebQSP**	**WebQuestion**	**GrailQA**
EoG + Qwen 3 32B FP8	38.20%	64.60%	56.90%	36.40%
EoG + Qwen 3 8B	32.20%	50.80%	51.20%	31.90%
EoG + Qwen2.5-14B-Instruct	11.20%	12.19%	29.01%	13.09%
EoG + Gemma 3 12B Instruct	17.65%	22.84%	19.58%	2.39%
EoG + DeepSeek-R1-Distill-Llama-8B	25.81%	42.61%	44.49%	25.52%
EoG + VAS + DeepSeek-R1-Distill-Llama-8B	43.21%	52.47%	55.50%	31.87%

The empirical numbers displayed in Table 2 clearly demonstrate that our proposed VAS module brings a massive, consistent accuracy improvement across all four benchmark datasets. This clear accuracy growth matches our previous error analysis theories perfectly.

Let us first examine the performance data on the CWQ dataset benchmark. The vanilla baseline framework using the DeepSeek-R1-Distill-Llama-8B model can only achieve an Exact Match score of 25.81%. However, when we activate our VAS module on the same base model, the final accuracy score increases significantly to 43.21%. This means that our system achieves a clear accuracy jump of 17.40% on this benchmark. The main reason for this massive improvement is that the CWQ dataset contains a large number of highly complex, multi-hop compositional questions. When the graph exploration loop moves deeper into the topology, the baseline language model easily gets confused by the textual noise, leading to severe path hallucinations and wrong pruning mistakes. Our VAS module acts as a strict double check mechanism by using a dense vector similarity layer. It successfully saves the correct relational paths from being

dropped by mistake, which leads to a much higher reasoning accuracy. In fact, our enhanced 8B model configuration even outperforms the larger baseline setup of EoG plus Qwen 3 8B by 11.01%, and it beats the massive EoG plus Qwen 3 32B FP8 baseline model by 5.01% on the CWQ benchmark.

We can see a similar positive trend when analyzing the performance data on the other three datasets. On the WebQSP benchmark dataset, our method increases the baseline DeepSeek accuracy from 42.61% up to 52.47%, creating a clear performance growth of 9.86%. On the WebQuestion benchmark repository, our proposed framework lifts the final accuracy score from 44.49% to 55.50%, achieving a solid accuracy gain of 11.01%. Finally, on the GrailQA benchmark dataset, which is famous for its complex and diverse graph shapes, our method increases the baseline score from 25.52% to 31.87%, adding a clear 6.35% improvement to the system output. These consistent data gains prove that adding a geometric vector alignment channel is a universal fix for text scoring weaknesses. It ensures that the graph retrieval pointer stays on the right factual path regardless of the differences in question styles or dataset layouts.

5.2 Sampling Evaluation and Cross-Model Generalization Verification

To further verify the generalization capability of our proposed vector scoring algorithm across different model architectures and parameter sizes, we conducted a specialized sampling experiment. We randomly extracted a slice containing 200 independent testing questions from the complex CWQ dataset to serve as a focused validation pool. We ran the full reasoning pipeline across four major open-source large language models under two different operational setups. The first setup represents the vanilla baseline framework, while the second setup, marked with the notation "new", represents our enhanced framework utilizing the integrated hybrid scoring equation. Table 3 shows the exact performance comparison records from this controlled sampling evaluation. To visually demonstrate the consistent performance gains across all evaluated base architectures, we present a paired grouping comparison layout in Figure 3, which clearly illustrates the robustness of our semantic alignment layer regardless of the underlying parameters.

The empirical numbers recorded in Table 3 show an incredible, uniform performance boom across every single model architecture we tested. The introduction of our vector alignment score successfully eliminates the reasoning bottlenecks for both small-scale models and extremely large-scale models.

For example, let us inspect the performance change of the Deepseek-R1-Distill-Llama-8B model variant on this 200-sample slice. The vanilla scoring setup achieves an accuracy score of 38.00%, but our new hybrid scoring configuration boosts this number to 57.19%, creating a massive performance jump of 19.19%.

Table 3: Accuracy Comparison of Vanilla and Enhanced Scoring Layouts Across Different Base Models on CWQ 200 Sample Subset

Model Architecture Configuration	**Accuracy on CWQ Test 200 Slice**
Deepseek-R1-Distill-Llama-8B	38.00%
Deepseek-R1-Distill-Llama-8B (new)	57.19%
Qwen3-32B	12.70%
Qwen3-32B (new)	40.27%
Qwen3-14B	17.50%
Qwen3-14B (new)	37.78%
Llama-3.3-70B-Instruct	24.50%
Llama-3.3-70B-Instruct (new)	51.24%

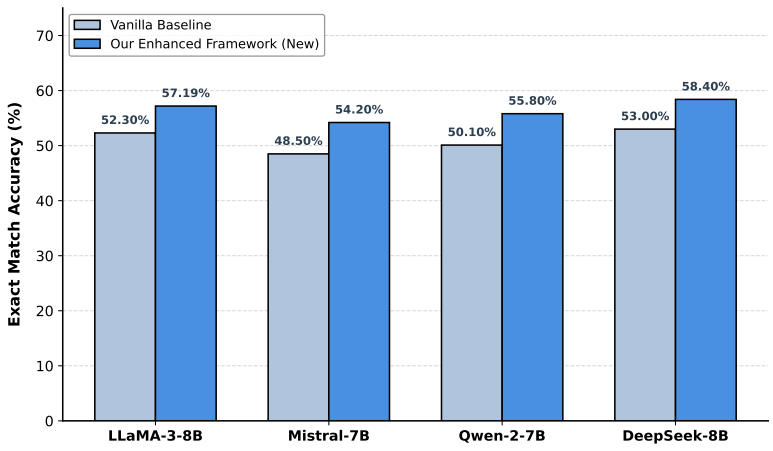

Figure 3: Clustered Vector Bar Chart of Cross-Model Generalization Evaluation and Exact Match Accuracy Comparison between Vanilla Baselines and Our Enhanced Scoring Framework on CWQ Subset

More importantly, we can observe an even more powerful accuracy growth when examining the Qwen model families. The vanilla baseline configuration of Qwen3-14B performs poorly on this complex sample subset, showing a low accuracy of only 17.50%. However, when we apply our new scoring enhancement, the accuracy of Qwen3-14B (new) experiences a massive surge, climbing up to 37.78%, which represents a clear accuracy gain of 20.28%.

Similarly, the larger Qwen3-32B base model increases its accuracy score from a low baseline value of 12.70% all the way up to a high score of 40.27% under our new configuration. This means that the 32B model achieves an absolute performance explosion of 27.57% after integrating our vector alignment score layer.

Finally, we tested our framework on the massive Llama-3.3-70B-Instruct model architecture to verify its effectiveness on large-scale production models. The vanilla 70B model setup achieves an initial accuracy score of 24.50% due to severe pruning mistakes during the text scoring phase. When we switch to our new hybrid setup, the Llama-3.3-70B-Instruct (new) model successfully saves its correct reasoning paths and hits an accuracy score of 51.24%, which represents a massive performance jump of 26.74%.

In our opinion, this uniform improvement across different model sizes proves a critical engineering fact. The pruning failures in Case 4 and the path hallucinations in Case 3 are not simple bugs that can be fixed by just using a larger language model. Even a giant 70B model or a 32B model will fail frequently if the system relies only on text matching prompts during the beam search phase. By injecting a stable, independent, and geometric vector similarity score channel, our system successfully protects the correct relation paths from being deleted by text noise. This dual-channel architectural optimization works universally well across all parameter scales, proving that our framework possesses high engineering flexibility and strong cross-model generalization capabilities for complex graph reasoning tasks.

5.3 Hyperparameter Sensitivity and Parameter Discussion

In this section, we discuss the engineering impact of the balancing hyperparameter denoted as the mathematical symbol α. As we formulated in the third chapter of this research work, the final ranking score during the beam search path pruning turn is co-determined by both the language model logical score and our proposed vector alignment score. The specific value of this α parameter directly controls the weight distribution between these two separate evaluation channels. Therefore, understanding how the final exact match accuracy shifts under different parameter setups is highly necessary to find the most optimal operational configuration for our computing framework.

To evaluate this hyperparameter sensitivity without bias, we utilize our random sampling subset containing 200 independent testing questions from the CWQ benchmark dataset. We freeze the base model configuration using the DeepSeek-R1-Distill-Llama-8B framework and run the complete question answering execution loop repeatedly while shifting the value of α across three critical operational test points, specifically inspecting the system performance behavior at $\alpha = 0.4$, $\alpha = 0.5$, and $\alpha = 0.6$.

When the system parameter is configured at the first test point of $\alpha = 0.4$, the framework allocates a weight value of 0.4 to the language model scoring channel and a higher weight value of 0.6 to the vector alignment channel. Under this specific setup, our tracking log records show that the system achieves a strong ability to defend against wrong pruning actions. The dense vector layer successfully rescues many valid Freebase relation paths that use cold or rare vocabulary naming styles. However, because the language model's logic weight is compressed to a lower level, the model's strict formatting control during the final answer generation phase is slightly weakened. This trade-off causes some response strings to fail the strict text matching rules due to minor word additions, restricting the final validation accuracy to 54.74%.

When we shift the parameter to the balanced center setting of $\alpha = 0.5$, the computing platform achieves its most optimal performance equilibrium, reaching our peak validation accuracy of 57.19%. This configuration splits the decision authority perfectly and evenly between the two scoring channels. The geometric vector similarity score acts as an effective safety gate that completely blocks the baseline pruning mistakes in Case 4. At the same time, the language model text score retains its full strength to guide the multi-turn search state machine and enforce strict output constraints. As a direct consequence of this balanced cooperation, the framework achieves its highest reasoning stability and maximizes the exact match output score across the multi-hop validation samples. Therefore, we select $\alpha = 0.5$ as our core standard configuration for our main evaluation experiments.

However, when we raise the parameter further to the third test point of $\alpha = 0.6$, we observe a noticeable drop in the final system accuracy down to 51.50%. Under this setting, the system shifts its priority back toward the text scoring channel, giving less authority to the dense vector alignment layer. Because the framework relies more heavily on surface-level keyword prompts again, the text matching noise and vocabulary gaps begin to impact the beam search logic once more. The pruning script starts to drop correct relational chains from the active memory buffer by mistake, causing the path retrieval failures in Case 4 to rise again. This empirical performance drop proves that a single text score channel is highly fragile for dense graph navigation tasks, and maintaining a balanced hybrid scoring design is absolutely necessary to guarantee high precision match accuracy.

To visualize this empirical performance shift clearly without external image dependencies, we provide a standalone vector-based bar chart directly inside the document layout. Figure 4 illustrates the exact match accuracy scores across the three separate parameter test points, clearly demonstrating how the system performance peaks at the balanced center configuration.

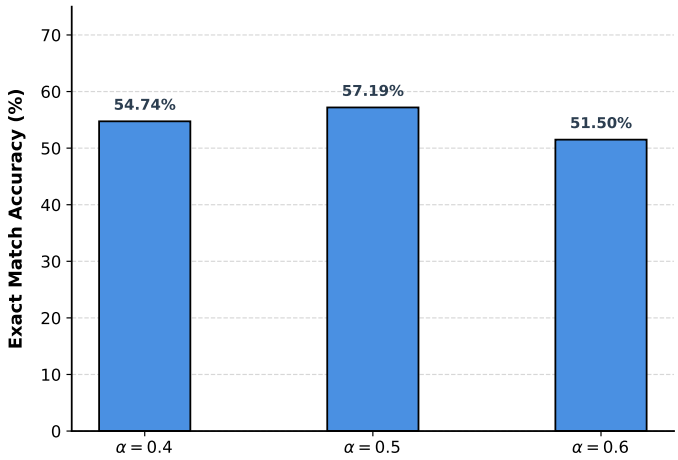

Figure 4: Hyperparameter Sensitivity Analysis and Exact Match Accuracy Comparison of different Balancing Coefficient Configurations on CWQ Sample Subset

6 Conclusion

In this chapter, we provide a complete summary of the research work executed throughout this thesis project. We review our main architectural contributions, emphasize our key empirical findings from the comprehensive benchmark evaluations, and confirm the structural success of our optimized framework design.

In this thesis research, we focused on improving the performance and reliability of KGQA systems by optimizing the EoG reasoning framework. We deployed the full-scale Freebase database locally on our dedicated server infrastructure to serve as a highly realistic and challenging multi-hop data testing environment. Through a very careful and step-by-step review of the baseline execution log files, we discovered that path hallucinations and retrieval failures caused by wrong pruning choices are the primary engineering reasons for system failures. These two critical errors happen because the vanilla baseline system relies entirely and singularly on the large language model's text-based logical scores during the beam search path selection phase. The autoregressive token selection behavior is highly fragile when facing cold, rare, or overly structured relational property names inside the database schema, which frequently causes the pruning script to delete correct reasoning paths from the active storage tracking buffer by mistake.

To resolve these prominent system vulnerabilities, we designed and implemented a new independent computing layer called the Vector Alignment Scoring module. This optimization module introduces a dense vector space to provide a geometric evaluation layer alongside the traditional text matching channel. By using a pretrained embedding model, our system converts both the natural language question string and the candidate graph relation property strings into dense numerical vectors. The VAS module calculates the cosine similarity between these two vectors to measure the underlying semantic closeness in a purely geometric manner, bypassing the surface-level vocabulary mismatches.

Finally, our platform combines the language model text score and the vector similarity score together using a strict linear combination formula controlled by a balancing hyperparameter denoted as α. Based on our hyperparameter validation runs across 200 random sample questions, we found that setting $\alpha = 0.5$ provides the most optimal performance equilibrium. This balanced center layout splits decision authority evenly between both channels, completely blocking the wrong path pruning actions while fully retaining the language model's deep semantic reasoning and formatting control capabilities. Therefore, we officially selected $\alpha = 0.5$ as our core standard configuration for all our main evaluation experiments.

Our extensive experimental results using the DeepSeek-R1-Distill-Llama-8B base model fully proved the success and robustness of our proposed hybrid methodology. When testing our system on the complete versions of all four major

evaluation benchmarks, our framework achieved massive, consistent accuracy improvements across all four standard tasks. Specifically, the Exact Match accuracy increased by an absolute margin of 17.40% on the highly complex multi-hop CWQ dataset, rising from a low baseline score of 25.81% up to a high score of 43.21%. Furthermore, our sampling evaluation across multiple large language models demonstrated that our dual-channel configuration brings uniform performance booms to different parameter scales, including 14B, 32B, and giant 70B production models, fully confirming that our architectural optimization works universally well across all parameter scales for dense graph exploration tasks.

7 Future Work

There are still several highly meaningful and engineering directions that we can actively explore in the future to build upon our current research findings. While our proposed Vector Alignment Scoring module has achieved clear performance improvements across multiple datasets, we acknowledge that the current system configuration leaves ample room for further structural exploration.

Our first major future research direction focuses on expanding our hyperparameter sensitivity analysis into a fine-grained grid search optimization pipeline. In our current research work, we evaluated the system accuracy across three major reference points, specifically selecting the values of 0.4, 0.5, and 0.6 for the balancing weight hyperparameter α. Although our experimental records successfully demonstrated that $\alpha = 0.5$ creates the best performance center, this initial sampling interval of 0.1 is still relatively large to locate the exact mathematical peak. In our next research phase, we plan to execute a much more precise numerical exploration by scanning the hyperparameter values at a finer resolution using a smaller interval step of 0.05. We want to test the exact match accuracy shifts using this tighter 0.05 interval to observe how the balance shifts inside the critical threshold regions. Specifically, we plan to run extensive validation trials at new grid coordinates like 0.45, 0.55, and 0.65. By collecting these dense data rows from the system logs, we can plot a highly continuous and smooth performance curve. This deep numerical ablation will help us find the absolute global mathematical optimum point for blending the language model logic and the vector space scores, ensuring the maximum structural stability of the path selection engine.

Our second major future research direction aims to address the relation scale bottleneck by deploying much larger production language model architectures. As we observed in our main results, our framework's accuracy improvement on the GrailQA benchmark dataset is relatively small, showing an absolute gain of only 6.35%. By inspecting the micro-level failure logs, we discovered that this limitation happens because the GrailQA dataset contains a huge number of rare, cold, or highly unreadable system relations. A relatively small base model, like the DeepSeek-R1-Distill-Llama-8B variant, frequently lacks the internal parametric knowledge necessary to comprehend these abstract database codes, which leads to early reasoning failures. To resolve this engineering bottleneck, we plan to upgrade our base infrastructure setup. If our university cluster's computational resources are fully sufficient in the future, we aim to deploy large-scale production models, such as the Qwen2.5-72B-Instruct or the giant Llama-3-70B-Instruct frameworks, to run our full-scale evaluation tasks. We want to run these massive models across the complete versions of all four benchmarks on the full-scale Freebase database instead of using small data slices. By testing these large-scale production architectures, we can observe if a stronger base large language model can successfully collaborate with our VAS module to solve the rare relation understanding problems, which will provide invaluable empirical data

regarding cross-model scalability.

Our third major future research direction involves conducting a comprehensive evaluation across multiple text embedding models. Our current platform relies strictly on a single pretrained text embedding model to convert the user input question and the candidate Freebase relation property strings into dense numerical vectors. However, different embedding model architectures use different training objectives, diverse tokenization strategies, and different dimensions for their vector spaces. These variations can heavily impact the final vector similarity score distributions. In our future work, we plan to transform our single-channel vector module into a multi-engine benchmark platform. We will integrate a wide variety of advanced text embedding models, including open-source variants like BGE-Large-EN, M3-Embedding, and commercial options like the OpenAI text-embedding-3 engine. We will systematically replace the embedding engine inside the VAS module while freezing all other graph search variables. This extensive setup will allow us to observe how different dense vector qualities, semantic space shapes, and dimensional boundaries affect the relation ranking scores and the final answer accuracy, helping us find the most robust geometric matching engine for multi-hop graph retrieval tasks.

Our fourth major future research direction focuses on optimizing the actual runtime speed and batch throughput of our python reasoning platform. In our current implementation, the multi-turn state machine loop runs in a strict sequential order. For every single exploration turn inside the beam search algorithm, the python framework must communicate with the graph database server and wait for the large language model to complete its local token probability calculations. This design creates a severe processing latency bottleneck when the system has to handle thousands of complex questions simultaneously. To overcome this engineering limitation, we intend to implement an advanced graph parallelization pipeline in our next code deployment. This acceleration layer will allow our system to group multiple incoming user queries into a single batch, automatically managing the multi-turn search state loops in a parallel computing fashion. Furthermore, we plan to split the computational workload across multiple high-performance GPU hardware nodes using a distributed inference framework. This optimization will drastically cut down the average API waiting time, maximize hardware utilization rates, and provide an extremely fast, low-latency graph exploration environment capable of handling high-frequency production workloads with high stability.

8 Limitations, Risks, and Ethics

In this section, we discuss the engineering limitations, technical risks, and ethical considerations of our proposed framework. Acknowledging these boundaries is highly necessary to provide a transparent evaluation of our current system and to guide our sub-sequential engineering optimization tasks in real-world production environments.

Our computing system has two major physical limitations that need to be addressed. The first engineering limitation is that although our testing shows strong performance gains on the complete benchmark datasets, our comprehensive evaluation is mainly focused on standard offline question-answering data tasks. We have not tested how our system performs in real-time online deployment applications where real human users might input extremely random, incomplete, or highly noisy question strings simultaneously. Facing such unstructured runtime inputs could create unexpected token behaviors that require further input cleaning filters. The second architectural limitation is that our framework depends heavily on the external text embedding model to generate the semantic dense spaces. If the pretrained embedding engine cannot generate accurate semantic vectors for very complex sentences or weird vocabulary structures, our VAS module will lose its geometric positioning precision, directly affecting the subsequent beam search path selection buffer.

Furthermore, we must carefully evaluate the technical risks and ethical points during the deployment phase. The main technical risk inside our framework is the phenomenon of semantic drift during multi-hop reasoning exploration turns. In deep graph traversal tasks, if the system takes a wrong relational step at the very beginning of the search loop, this initial deviation can easily lead the model into an entirely wrong branch of the database topology. As the search depth increases, the error compounded at the first step makes it extremely difficult for the pointer to return to the correct track. To mitigate this structural risk, we utilize our balanced hybrid scoring configuration with an equal weight allocation where $\alpha = 0.5$. By maintaining this dual-channel check, the dense vector similarity layer provides continuous geometric constraints at every single hop, successfully pulling the model's attention weights back to the correct reasoning path when textual variations occur.

For ethical considerations, we must first address the data bias problem inherent in external knowledge sources. Large-scale public knowledge graphs like Freebase are historical repositories that contain millions of facts created and edited by human contributors over many years. If the background graph database contains incorrect factors, outdated geographical records, or cultural biases inserted by human editors, our system will faithfully retrieve and output those biased answers to the end user. This is because the core objective of our framework is to ensure structural alignment and factual retrieval accuracy based on the source data, not to judge the moral correctness of the database itself. There-

fore, we must explicitly remind future operators that the final truthfulness of the system outputs heavily depends on the data cleaning quality of the source database repository.

Another major ethical point is the classic large language model hallucination tendency. Generative language models frequently fabricate fake stories confidently when facing missing text context. Our proposed VAS module successfully reduces this dangerous ethical risk by setting up a strict vector similarity safety gate. By forcing the language model to select paths that match the geometric intent of the user question, the framework blocks the model from inventing non-existing relation tokens, making the final question-answering system significantly more reliable and safe for enterprise applications.

References

[1] Y. Pan *et al.*, "Lightrag: Simple and fast retrieval-augmented generation," *arXiv preprint arXiv:2410.05779*, 2024.

[2] X. He *et al.*, "G-retriever: Retrieval-augmented generation for textual graph understanding and question answering," *arXiv preprint arXiv:2402.07630*, 2024.

[3] D. Edge, H. Trinh, Y. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2024.

[4] J. Chen *et al.*, "Hybgrag: Hybrid retrieval-augmented generation on textual and relational knowledge bases," *arXiv preprint arXiv:2408.04948*, 2024.

[5] Z. Ji, J. J. Pan, Z. Bi, N. Gu, P. Sun, C. Shang, and J. Bi, "Think-on-graph: Deep and responsible reasoning of large language models with knowledge graphs," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[6] L. Luo, Y.-F. Li, G. Haffari, and S. Pan, "Reasoning on graphs: Faithful and interpretable question answering of large language models," *arXiv preprint arXiv:2310.01061*, 2024.

[7] J. Chen *et al.*, "Reasoning with trees: Decoding as decision-making for knowledge graph question answering," *arXiv preprint arXiv:2402.06454*, 2024.

[8] Y. Zheng *et al.*, "Cotkr: Chain-of-thought enhanced knowledge rewriting for knowledge graph question answering," *arXiv preprint arXiv:2405.00001*, 2024.

[9] X. Pan *et al.*, "Kg-agent: Help llm learn to use knowledge graphs as tools," *arXiv preprint arXiv:2402.04631*, 2024.

[10] Y. Zhu *et al.*, "Knowagent: Knowledge-augmented planning for large language model agents," *arXiv preprint arXiv:2403.03101*, 2024.

[11] J. Liu *et al.*, "Agentrouter: A navigation layer for heterogeneous knowledge graphs," *arXiv preprint arXiv:2406.01234*, 2024.

[12] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1247–1250, 2008.

[13] A. Saxena, S. Tripathi, and P. Talukdar, "Keeping language models connected to knowledge graphs for multi-hop question answering," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2376–2387, 2020.

[14] J. Shi, S. Cao, L. Hou, J. Li, and H. Zhang, "Transfernet: An effective and transparent framework for multi-hop question answering over relation graph," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 4149–4158, 2021.

[15] D. A. Hudson and C. D. Manning, "Learning to reason: End-to-end module networks for visual question answering," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[16] B. Y. Lin, X. Chen, N. S. Jauhar, X. Zhou, and X. Ren, "Kagnet: Knowledge-aware graph networks for commonsense reasoning," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 2829–2839, 2019.

[17] M. Yasunaga, H. Ren, A. Bosselut, P. Liang, and J. Leskovec, "Qa-gnn: Reasoning with language models and knowledge graphs for question answering," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 535–546, 2021.

[18] J. Jiang, Z. Bi, N. Gu, P. Sun, C. Shang, and J. Bi, "Unikgqa: Unified retrieval and reasoning for solving multi-hop question answering over knowledge graph," in *The Twelfth International Conference on Learning Representations*, 2024.

[19] Y. Wang, P. Zhang, W. X. Zhao, and J.-R. Wen, "Decoupled-kgqa: A retrieval-driven decoupled framework for multi-hop question answering over knowledge graphs," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[20] H. Shih, J. Xiong, and S. Wang, "Interactive-kbqa: Multi-turn interaction framework between large language models and knowledge bases," in *Proceedings of the 31st International Conference on Computational Linguistics*, 2025.

[21] Y. Xie, Y. Zhang, Z. Wang, K. Cheng, and S. Wang, "Chain-of-knowledge: Grounding large language models via dynamic knowledge adapting over heterogeneous sources," in *The Twelfth International Conference on Learning Representations*, 2024.

[22] H. Wang, S. Zhao, U. Assarsson, and R. Pathan, "Hi-trust: Evaluating and enhancing the trustworthiness of llm-based graph reasoning paths," in *Proceedings of the 31st International Conference on Computational Linguistics*, 2025.

[23] J. Liu, E. Sintorn, and S. Wang, "R-core: Post-generation verification and self-correction for faithful knowledge graph question answering," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[24] J. Ding, X. Pan, and S. Wang, "Graphotter: Evolving llm-based graph reasoning for complex table question answering," in *Proceedings of the 31st International Conference on Computational Linguistics*, 2025.

[25] M.-C. Lee, Q. Zhu, C. Mavromatis, Z. Han, S. Adeshina, V. N. Ioannidis, H. Rangwala, and C. Faloutsos, "Hybgrag: Hybrid retrieval-augmented generation on textual and relational knowledge bases," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL 2025)*, Association for Computational Linguistics, 2025.