Continue AI-Powered Chatbot for Quick Access to Jenkins Resources

Name: Anshuman Singh
Email: anshu.2005.12@gmail.com
Github handle: IND-Anshuman (Anshuman Singh)
Resume: Anshuman's Resume

Project Abstract.
With this project I will try to upgrade the Jenkins chatbot by introducing a graphRAG pipeline that combines with the initial RAG pipeline to finally convert it into a powerful chatbot that considers contextual understanding and interconnected relationships before answering the users query to boost the quality and accuracy of output for the users.

Project Description:

The Problem

Through my research I found that the current Jenkins AI chatbot uses a standard RAG pipeline: documentation is chunked, embedded, and retrieved using vector similarity. This works reasonably well for straightforward informational queries. If someone asks "What is Jenkins Pipeline?", the system can usually pull relevant documentation and summarize it.

However, after studying the existing pipeline, a few limitations become visible.

First, the system treats most content as independent chunks of text. It does not states relationships between plugins, versions, errors, and core components. For example, if a user asks about a plugin failure after an upgrade, the answer may require understanding dependencies between plugins and version compatibility. Second, complex or multi-step questions are difficult to answer. Queries such as:

- "Which plugins depend on workflow-api?"
- "Why is my pipeline failing after upgrading plugin X?"
  often require reasoning across multiple entities. A flat RAG setup does not actually support this kind of multi-hop reasoning.

Right now, these relationships are only implicitly present in text. The chatbot does not have a structural understanding of them. As a result, answers can sometimes be incomplete, overly generic, or occasionally misleading when multiple components interact.

Motive of the Proposal

The goal of this proposal is to extend the existing chatbot from a purely text-based retriever into a system that can reason over structured relationships in the Jenkins ecosystem.

Instead of treating documentation as isolated chunks, I propose constructing a knowledge graph representing plugins, dependencies, errors, versions, and their relationships. This would allow the chatbot to:

- Prioritize structurally important relationships (e.g., depends_on vs. simple textual references).
- Restrict retrieval to relevant domains when possible to reduce time taken in generating output.
- Combine semantic search with graph-based traversal for better precision and accuracy.

I am thinking of reducing hallucinations and improving the quality of answers for complex troubleshooting and compatibility based questions with these changes.

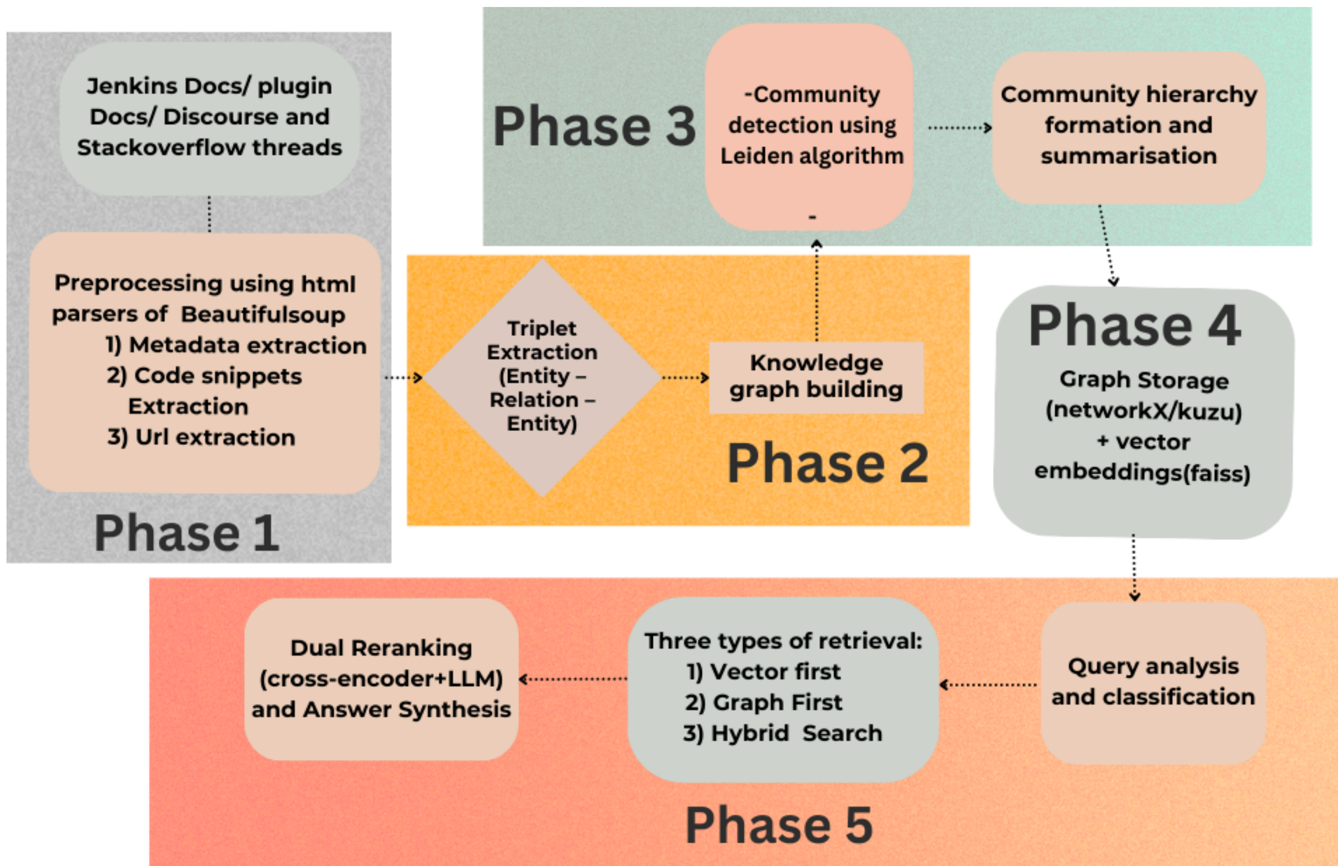
Impact on Jenkins

These enhancements that I mentioned would help upgrading the basic jenkins chatbot to a highly contextual and hierarchy considering retrieval agent. It improve user experience in the following ways:

- The user will be able to get highly accurate answers with least amount of hallucinations.
- Troubleshooting will become easier due to metadata formation which contains code snippets and error resolving mentions.
- Earlier when the users get dependency or version mismatch, they have to do a lot of research to fix it. But with the strong relationship understanding of the knowledge graph, these errors can be fixed easily.
- This chatbot can be considered as a foundation for future AI driven enhancements like adding OCR pipeline, advanced automations, intelligent recommendations and other impactful integrations.

Technical Architecture:

I plan to upgrade the jenkin's simple RAG chatbot into a powerful graphRAG one in 6 phases:



Phase 1: Enhancing the data extraction process to form metadata

After reviewing the current data extraction pipeline in chatbot-core/data, I noticed that while the extraction itself works reasonably well, most of the content is ultimately stored as flat chunks of text. There is little structural metadata extraction done. For a GraphRAG-based system, this becomes a limitation. If we want to build a meaningful knowledge graph later, we need richer context at the preprocessing stage. So the first step would be to enhance the preprocessing layer before any graph construction begins.

Extracting Metadata and Heading Hierarchy

Before stripping HTML content into plain text, I plan to introduce a parsing step that extracts:

- URL
- Page title
- Section hierarchy (H1, H2, H3, etc.)
- Parent-child relationships between sections

It will preserve the logical structure of documentation pages instead of flattening them immediately.
A simplified representation could look like:

```
{
 "url": "...",
 "title": "...",
 "sections": [
  {
   "heading": "Pipeline Syntax",
   "level": 2,
   "content": "...",
   "parent": "Using Jenkins"
  }
 ]
}
```

This is just a simple representation of the format. It can be much more complex.

Code Snippet and Link Extraction

In addition to headings, I will extract:

- Code blocks (with labels where possible)

- Internal documentation links
- Possibly external references if they are relevant

Code snippets are especially important in Jenkins documentation because many plugin behaviours are explained through example codes. Internal links can also serve as weak signals for relationships between pages.

```python
from bs4 import BeautifulSoup

def extract_jenkins_metadata(html_content, url):
    """
    Extracts hierarchical structure and code blocks from Jenkins documentation.
    """
    soup = BeautifulSoup(html_content, 'html.parser')
    page_data = {
        "url": url,
        "title": soup.title.string if soup.title else "Unknown",
        "sections": [],
        "code_snippets": []
    }

    current_parent = None

    # Capture headings to establish hierarchy
    for tag in soup.find_all(['h1', 'h2', 'h3', 'pre']):
        if tag.name in ['h1', 'h2']:
            current_parent = tag.get_text(strip=True)

        elif tag.name == 'h3':
            page_data["sections"].append({
                "heading": tag.get_text(strip=True),
                "parent": current_parent,
                "content": tag.find_next_sibling('p').get_text(strip=True) if tag.find_next_sibling('p') else ""
            })

        elif tag.name == 'pre':  # Extract code snippets
            code_content = tag.get_text(strip=True)
            page_data["code_snippets"].append({
                "associated_heading": current_parent,
                "code": code_content
            })

    return page_data
```

Specification: This is not the complete plan for the implementation of this phase since I have not yet totally understood how much and what type of data the docs_crawler.py file is extracting. Also I have to finalise which parser I will use for the metadata extraction, most probably it will be an html parser of Beautifulsoup. There might be more enhancements that might be implemented in order to complete this phase successfully.

Phase 2: Knowledge Graph Construction

Once structured metadata is available, the next step is to construct the actual knowledge graph.
The motive here is to extract entities (plugins, errors, versions, core features, etc.) and define explicit relationships between them. This will be done in multiple controlled stages to avoid building a noisy or fragmented graph.

1. Entity and Relationship Extraction

Using the structured JSON produced in Phase 1, I will process content in smaller chunks and use an LLM (local or API-based) to extract triplets of the form:
(Entity A) —[Relation]→ (Entity B)

A predefined relationship schema will be used to constrain extraction. Initial relation types may include:

- Depends_on
- Provides
- Solved_by
- Compatible_with
- Extends
- References
- Configured_by

Risk: One important concern here is hallucination.I have experienced in my similar projects that LLMs hallucinate a lot in this step because their context window for complex queries is limited and if prompts are too open-ended, the LLM may invent entities or relationships. So prompts should be strict and restrictive to the relationship schema to reduce this risk. And a batching process should be introduced so that everything gets proper attention from the LLM without incurring any contextual limit.
The hierarchical structure extracted in Phase 1 should also help the model better understand context boundaries (e.g., whether something is part of a specific plugin section or general documentation).

2. Entity Normalization and Deduplication
Raw extraction often produces duplicate or slightly inconsistent entities. For example:
- "Git Plugin"
- "git-plugin"
- "Git plugin"

If not handled properly, this can fragment the graph and reduce traversal accuracy.
To address this, I will implement a multi-stage normalization pipeline:
1. Surface normalization (lowercasing, punctuation removal)
2. Alias resolution
3. High-threshold fuzzy matching (>90%)
4. Version separation (treating plugin versions distinctly when needed)
5. Confidence-based filtering

3. Assigning Relationship Weights
Not all relationships carry equal importance. For example:
- depends_on usually represents a strong architectural relationship.
- references might simply indicate a textual mention.

To reflect this difference, I plan to assign weights to relations.

Relation	Weight
depends_on	4
solved_by	4
provides	3
compatible_with	3
references	1

4. Confidence Filtering
Finally, the confidence scores generated during triplet extraction will be used to remove weak or unreliable edges.

Specification: I have not yet decided which type of llm will be used for the triplet extraction. Initially I will use the local Mistral model but if there will be low quality results from the local llm I would probably try to switch to an api-key based llm in the extended coding period to maximise the richness of the knowledge graph. This switch, according to me, will not be very expensive as the graph construction will not be much frequent.

Phase 3: Community Detection, Hierarchical Structuring and Community Summarisation
After building a normalized and weighted knowledge graph in Phase 2, the graph will likely be large and densely connected. However, the Jenkins ecosystem itself is not flat. In practice, it is modular as it has quite dense interconnections that are also priority based. This phase introduces structural organization into the graph through community detection.

1. Community Detection using Weighted Multi-Resolution Clustering
Since relationships were assigned weights in Phase 2, we can now use those weights during clustering.
Not all edges should influence clustering equally. For example:
- depends_on usually reflects a strong architectural link.
- solved_by connects errors to specific plugins.
- references may only indicate a loose textual mention.

If clustering ignores these differences, weak references might distort group formation. So I plan to use weighted clustering to ensure that stronger relationships have more influence when forming communities.

Algorithm Choice
I propose using the Leiden algorithm to capture both broad domains and finer sub-groups. I am also thinking of experimenting with multi-resolution clustering(. For example:

- Lower resolution → larger, domain-level clusters.
- Higher resolution → smaller, feature-level clusters.

Exact resolution values (e.g., 0.8 or 1.2) will likely require tuning after observing real graph behaviour.

2. Community Object Creation
Once communities are stabilized, they will be represented explicitly in the system rather than remaining implicit clustering outputs.
Each community will store:

- A community ID
- Hierarchy level (e.g., L1 or L2)
- Parent community reference (if applicable)
- Node count
- Basic cohesion metrics
- Dominant entity and relation types

This allows the retrieval system to treat communities as structured units, not just arbitrary sets of nodes.

3. Community Hierarchy Formation
Using the results from multi-resolution clustering:

a) Fine-grained communities (L2) will be mapped to broader communities (L1).
b) Parent-child relationships will be stored explicitly.

This creates a two-level hierarchy such as:
Level 1: CI/CD Core

- Pipeline Plugins
- SCM Integrations
- Notification Plugins

The exact labels will depend on what emerges from clustering, rather than being manually defined in advance.
This hierarchy allows the system to reason not only about individual plugins, but also about the broader domain they belong to.

4. Community Summarisation
Clustering groups nodes structurally, but the system still needs a semantic description of what each community represents.

A. Bottom-Up Summarisation Strategy
Instead of summarizing the entire graph at once, I will use a bottom-up approach.

Step 1: Summarise Fine-Level Communities (L2)
For each fine-level community, the local LLM will then generate a short structured summary describing about the cluster it represents, the features and plugins involved, typical problems or use-cases. This gives each cluster a semantic identity.
Step 2: Summarise Coarse-Level Communities (L1)
Rather than reprocessing all nodes again, L1 summaries will be generated from the summaries of their L2 sub-communities. Here we do not want to produce perfect documentation, but generate enough context for routing and explanation purposes.

B. Structured Summary Storage
Each community summary will store:

1. A short domain description
2. Key entities
3. Common user intents
4. Typical issues
5. Structural characteristics

The above image is a neo4j representation of a small part of a knowledge graph for a large legal document made by my earlier graphRAG project Nsure AI. It can represent how after this phase, the knowledge graph should no longer behave as a single undifferentiated structure. Instead, it will have stable clusters, multi-level hierarchy and semantic summaries. This makes it easier for later retrieval phases to restrict search space and perform more coherent multi-hop reasoning.

Specification: The leiden algorithm is not the final algorithm to be used in this phase, there are many other algorithms that might be a good choice for this project like Infomap, Hierarchical Agglomerative Clustering etc. The leiden algorithm is just a baseline algorithm and I might test other algorithms too if time permits.

Phase 4: Graph Storage Architecture (NetworkX and Kuzu Integration)
After constructing the knowledge graph and detecting communities, the next practical question is how to store and query it. This phase focuses on choosing a storage approach that is realistic for the scale of Jenkins data while remaining manageable during development.

Option 1: NetworkX (In-Memory Graph)
NetworkX is well suited for graph construction and experimentation within a Python-based pipeline. This part of the pipeline can remain offline or batch-based.
Limitations
However, NetworkX is not optimized for high performance queries, quick large scale traversal and other heavy tasks. Since it stores everything in memory, it may not scale well for runtime query handling as the graph grows.

Option 2: Kuzu (Embedded Graph Database)
For runtime usage, Kuzu, an embedded graph database, is also a good option according to me since it provides efficient on-disk storage, indexed traversal, cypher-like query support and is light-weight to deploy.
Kuzu provides:

Proposed Hybrid Approach
Rather than choosing one tool exclusively, I am thinking of separating offline graph construction from runtime querying by using both the tools(networkX and kuzu). Though it will be more complex but it will be worth the effort for future improvements
Step 1: Graph Construction (Offline)
Using NetworkX :
1. Build the graph
2. Normalize entities
3. Assign weights
4. Run community detection
5. Create hierarchical structure

This stage focuses on correctness and experimentation.
Step 2: Graph Export
After the graph structure stabilizes:
- Convert nodes and edges into a structured export format
- Insert them into the Kuzu database
- Create indices on:

1. Entity ID
2. Relation type
3. Community membership

Compatibility Between NetworkX and Kuzu
Since these systems are different, synchronization needs to be handled carefully.
Key points to ensure:
1. Node identifiers must remain consistent.
2. Relation types must map exactly to the database schema.
3. Community labels must be stored explicitly.
4. The export process must be deterministic.

To reduce the risk of mismatches:
- Canonical entity IDs will be generated during normalization.
- Export scripts will validate schema consistency before insertion.
- Basic integrity checks will be performed after loading into Kuzu.

Some of these details may require refinement once integration testing begins.

Vector Storage (FAISS Integration)
It is important to clarify that FAISS will not store the graph itself.
FAISS will be used only for embedding-based retrieval of documentation chunks, community summaries and node descriptions.
A mapping table will maintain:
vector_id → entity_id
This allows us to bridge semantic retrieval (FAISS) with structural traversal (Kuzu).
For example:
1. A query retrieves semantically similar documentation chunks.
2. Those chunks map to graph entities.
3. The graph is then traversed from those entities for multi-hop reasoning.

This keeps semantic similarity search and structural reasoning clearly separated, but connected when needed.

Specification: Though I have proposed the hybrid structure involving both NetworkX and Kuzu but it will be an overkill for the current Jenkins data. Even NetworkX can work fine for the current amount of data extracted by docs_crawler.py. I ran the complete document extraction process and found that the number of urls it has gone through is less than 3000, so overall I can assume that the number of entities extracted by the triplet extraction process would not reach more than 50k and the NetworkX library can support more than 100k entities easily. The use of Kuzu will only arise when the entity count reaches near 200k when NetworkX starts malfunctioning. So which library to use will be decided after consulting the mentors. But the hybrid form suggested can ensure long-term performance of the chatbot without any proper maintenance.

Phase 5: Adaptive Retrieval
According to me not every user question requires the same level of reasoning. Some queries are straightforward definitions. Others involve dependencies, version conflicts, or error resolution across multiple plugins.
So instead of applying a single fixed retrieval strategy to every query, so I have thought about using an adaptive strategy.

1. Query Understanding and Strategy Selection
The first step is lightweight query analysis.
Before retrieving anything, the system attempts to identify what type of question it is. For example:
- “What is Jenkins Pipeline?” → Informational
- “Which plugin depends on workflow-api?” → Structural
- “Why am I getting MissingContextVariableException?” → Troubleshooting
- “How does Git plugin integrate with pipeline in Jenkins 2.361?” → Multi-step

To do this, the system can check:
- Are specific plugin names mentioned?
- Are error messages present?
- Does the query imply dependency or compatibility reasoning?
- Is a specific version referenced?

Based on these signals, one of three strategies is selected:

a) Vector-First Retrieval (Informational Queries)
For simpler questions, semantic similarity is often sufficient.
In these cases:
- The system performs embedding-based search over documentation.
- It retrieves the most relevant sections.

- If needed, it may consult the graph lightly for additional structural context.

The main difference from the earlier RAG approach is that retrieval can now be restricted to relevant communities instead of searching the entire corpus blindly.
This keeps latency low while still benefiting from structural grounding.

b) Graph-First Retrieval (Structural Queries)
When a query clearly asks about relationships — for example:

- Dependencies
- Compatibility
- Extensions
- Error-to-plugin mapping

—it makes more sense to consult the graph directly.
Instead of retrieving text chunks and hoping they mention the answer, the system can query the graph for depends_on relationships involving that entity.
This reduces ambiguity and improves precision, especially for dependency tracing.

c) Hybrid graph Expansion (Complex Queries)
Some queries require reasoning across multiple steps. For example:
"Why is my pipeline failing after upgrading plugin X?"
Answering this may require identifying the relevant error, its relation to a plugin, checking plugin dependencies and version compatibilities etc. To handle this, the system will perform controlled multi-hop traversal.
Key constraints:

- Start from entities detected in the query.
- Traverse up to a limited depth (e.g., 2 hops).
- Prioritize stronger relationship types (depends_on, solved_by, etc.).
- Enforce a node or edge budget to prevent over expansion.

These constraints are important because unrestricted graph traversal can quickly become expensive and noisy.

5. Community-Constrained Expansion
Since communities were introduced earlier, they can now help restrict traversal.
If a question clearly relates to pipeline plugins, expansion should remain within that domain unless a strong relationship requires stepping outside it. The exact constraints may require tuning after observing real query behaviour.

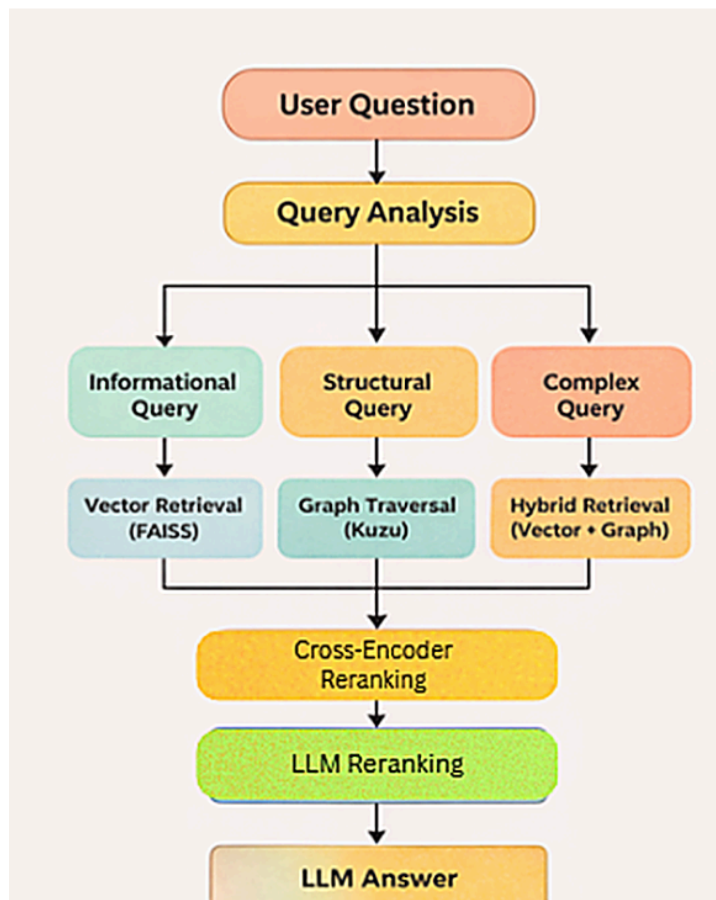
6. Dual Re-Ranking for Precision
After retrieval (vector-based, graph-based, or hybrid), the system will likely have multiple candidate snippets or graph paths.
Rather than sending all of them directly to the LLM, a filtering step is introduced.
Stage 1: Lightweight Re-Ranking
A smaller model or heuristic scoring system ranks candidates based on:

- Semantic relevance
- Relation weight importance
- Community alignment

This stage removes clearly weak candidates.
Stage 2: LLM-Based Re-Ranking
A local LLM then evaluates the top remaining candidates to check:

- Direct relevance to the query
- Completeness
- Technical consistency

This two-step process helps balance efficiency and answer quality.
After this phase, the chatbot still relies on RAG, but retrieval is no longer purely similarity-based. It can switch strategies according to queries, traverse structural relationships etc.



Specification: There will be multiple filtering steps, batching and guardrails that will be implemented in this phase that I have not included in the explanation. These steps will help in situations like json code breaking, context limiting problems and many others. Also the llm reranking in the dual reranking process can increase latency a little, but it is very useful according to my experience as it helped me increase the accuracy and robustness of my own graphRAG project above 90%.

Phase 6: Evaluation and Validation

The goal of this phase is straightforward: compare the upgraded system against the existing RAG-based chatbot and verify whether it performs better in areas where structured reasoning is expected to help.

1. Creating a Small Benchmark Set

To evaluate the system fairly, I will prepare a curated set of Jenkins-related questions grouped into categories.

Informational Queries

Examples:

- "What is Jenkins Pipeline?"
- "What does the Git plugin do?"

Dependency Queries

Examples:

- "Which plugins depend on workflow-api?"
- "Does plugin X require plugin Y?"

Troubleshooting Queries

Examples:

- "Why am I getting MissingContextVariableException?"
- "How do I fix a pipeline stage not executing?"

Multi-Hop Queries

Examples:

- "How does Git plugin integrate with pipeline?"
- "Why is my pipeline failing after upgrading plugin X?"

The benchmark set does not need to be large, but it should be diverse enough to cover both simple and structurally complex scenarios.

2. What Will Be Measured

The comparison between the simple RAG system and the upgraded GraphRAG system will focus on practical evaluation

criteria.

1. Answer Accuracy
2. Relevance of Retrieved Context
3. Multi-Step Reasoning Quality
4. Hallucination Reduction
5. Latency

Exact numerical thresholds may be defined after initial experimentation.

Why I am fit for this project:

I am currently a second-year B.Tech IIIT Jabalpur student with a strong interest in Generative AI, knowledge systems, and structured retrieval architectures. Over the past year, I have focused on building practical RAG systems rather than simple prompt-based applications.

Most notably, I built and deployed **Nsure AI**, a full GraphRAG-based chatbot that constructs a knowledge graph for every uploaded document and then queries that knowledge graph.

Repository: https://github.com/IND-Anshuman/Nsure_graph_AI

The system includes:

- LLM-based triplet extraction
- Entity normalization and deduplication
- Graph construction using NetworkX
- Leiden+Louvian based hybrid community detection
- Hybrid retrieval (vector + graph expansion)
- Dual-stage reranking (cross-encoder + LLM)
- Cloud deployment optimized for low memory usage

This project required solving real-world challenges such as API rate limits, large-context management, graph scaling, and latency optimization — all of which are directly relevant to upgrading the Jenkins chatbot to a structured GraphRAG system.

Through this experience, I have gained:

1. Insights on how graph-based retrieval systems works
2. Experience designing relationship schemas and normalization pipelines
3. Hands-on implementation of clustering and community detection
4. Integration of vector search with graph traversal
5. Deployment and performance optimization in constrained cloud environments

Since the proposed Jenkins project involves building a structured knowledge graph, implementing multi-hop reasoning, community detection, and adaptive retrieval, my prior experience directly aligns with its technical requirements. Overall, my experience with designing, implementing, and deploying a production-grade GraphRAG system gives me both the conceptual understanding and practical skills required to successfully complete this project.

Project Deliverables

1. Enhanced Data Preprocessing

In this phase, the preprocessing pipeline will be updated to extract structured metadata from Jenkins documentation. Instead of flat text chunks, the system will preserve titles, URLs, heading hierarchies, code snippets, and internal links, producing structured JSON suitable for graph construction. This will provide cleaner, context-rich data for later graph building.

2. Knowledge Graph Construction

This phase focuses on building the knowledge graph from the extracted data. It includes defining entity and relationship schemas, implementing LLM-based triplet extraction, and normalizing entities to remove duplicates. Relation weighting and confidence filtering will also be added to keep the graph clean and reliable.

3. Community Detection and Hierarchical Structuring

The constructed graph will be organized into meaningful clusters using weighted Leiden clustering. Small unstable clusters will be merged, and a two-level hierarchy will be created to represent broader domains and finer feature groups. Each community will also include a short summary to support later retrieval.

4. Storage Architecture Integration

The graph will be exported into a Kùzu database for runtime traversal, while NetworkX or igraph will be used for offline construction and experimentation. FAISS will be integrated separately for embedding-based retrieval, with a mapping layer connecting embeddings to graph entities. This will provide an efficient and scalable storage layer.

5. Adaptive Retrieval and Multi-Hop Reasoning
An adaptive retrieval system will be implemented to switch between vector, graph, or hybrid retrieval depending on the query type. Controlled multi-hop traversal, community-restricted expansion, and reranking will help improve precision. This should allow the chatbot to handle dependency and troubleshooting queries more reliably.

6. Evaluation and Benchmarking
A benchmark dataset of Jenkins-related queries will be created to compare the original RAG chatbot with the upgraded GraphRAG system. The evaluation will measure accuracy, context relevance, reasoning quality, hallucination reduction, and latency. Results will be summarized to clearly show the improvements and trade-offs.

7. Documentation and Final Submission
The final stage will include complete documentation, setup instructions, architecture diagrams, and well-commented code. A demo video and final report will also be prepared. The repository will be structured to ensure reproducibility and easy maintenance for future contributors.

```
chatbot-core/
├── data_pipeline/              # Phase 1: Enhanced Data Preprocessing
│   ├── crawler_enhancements.py # Updates to existing docs_crawler.py
│   ├── html_structural_parser.py # Extracts H1/H2, parent-child relations, code snippets
│   └── json_formatter.py       # Formats output for the graph builder
│
├── graph_engine/               # Phase 2 & 3: Knowledge Graph Construction
│   ├── triplet_extractor.py    # LLM prompts for Entity-Relation-Entity
│   ├── entity_normalizer.py    # Fuzzy matching and deduplication pipeline
│   └── community_builder.py    # Leiden algorithm & community summarization
│
├── storage/                    # Phase 4: Graph Storage Architecture
│   ├── faiss_manager.py        # Vector embeddings for semantic search
│   ├── networkx_builder.py     # In-memory offline graph construction
│   └── kuzu_adapter.py         # (Optional) Export logic for runtime DB
│
├── retrieval/                  # Phase 5: Adaptive Retrieval
│   ├── query_analyzer.py       # Intent classification (Informational, Structural, Complex)
│   ├── graph_traversal.py      # Multi-hop logic within constraints
│   └── dual_reranker.py        # Cross-encoder + LLM filtering
│
└── evaluation/                 # Phase 6: Evaluation and Validation
    ├── jenkins_benchmark.json  # Curated QA set
    └── evaluator.py            # Measures multi-step accuracy and latency
```

Proposed Schedule

Throughout the coding period, I plan to maintain regular commits and incremental progress rather than large, infrequent pushes. Each milestone will aim to produce working and reviewable code before evaluation checkpoints. The early stages will focus more on design clarity and validation, while later stages will emphasize implementation, integration, and testing.

March 16 – March 31 (Application Period)
During this period, I will study the existing Jenkins chatbot repository and understand how the current RAG pipeline works. I will identify possible extension points for introducing graph-based reasoning and outline an initial system architecture. I will also discuss the approach with mentors on mailing lists or Discourse to validate the direction.

April 1 – April 30 (Acceptance Waiting Period)
If selected, I will refine the technical design by defining entity and relationship schemas, graph structure, and normalization strategies. I will also design the evaluation plan and run small experiments such as testing triplet extraction on sample data. The goal is to finalize a clear implementation roadmap before coding begins.

May 1 – May 24 (Community Bonding Period)
During community bonding, I will review the design with mentors and break the project into concrete development tasks.

Repository structure, milestones, and the evaluation dataset will be prepared. This ensures that development can begin smoothly once the coding period starts.

May 25 – July 6 (First Coding Phase – Midterm)
The first coding phase will focus on building the knowledge graph pipeline. I will implement enhanced preprocessing, structured data extraction, triplet generation, and entity normalization. Community detection and hierarchical clustering will also be introduced during this phase to organize the graph structure.

July 6 – August 24 (Final Coding Phase)
In this phase, I will integrate the graph storage layer using Kùzu and connect it with FAISS for embedding retrieval. Adaptive retrieval, controlled multi-hop traversal, and reranking will be implemented. The upgraded system will then be integrated into the Jenkins chatbot and evaluated against the baseline RAG system.

August 24 – November 2 (Extended Period, if applicable)
If the extended period is available, I will focus on optimization and polishing. This includes improving traversal efficiency, refining summaries, running additional evaluation experiments, and strengthening documentation. A final demo and documentation package will also be prepared.

Future Improvements
The proposed GraphRAG architecture is designed to be extensible beyond the GSoC timeline. Possible future enhancements include:

- **Incremental Graph Updates:** Automatically updating the graph when new plugins or documentation changes are released, instead of rebuilding it from scratch.
- **Version-Aware Reasoning:** Adding explicit version tracking to improve compatibility and upgrade-related queries.
- **Explainable Reasoning:** Displaying dependency chains or reasoning paths used to generate answers, improving transparency and user trust.
- **User Feedback Integration:** Incorporating user feedback to refine retrieval quality and improve accuracy over time.
- **Performance Optimization:** Further improving latency and scaling strategies as the Jenkins ecosystem grows.

Continued Involvement
As a second-year B.Tech student, I am still exploring and shaping my long-term career path, so I have decided a very specific direction yet. However, I am strongly interested in Generative AI, knowledge systems, and intelligent assistants. I would like to continue contributing in areas related to AI-powered tooling, structured knowledge systems, and intelligent automation within Jenkins.

Major Challenges Foreseen
Several technical challenges may arise during the implementation of this project:

- **Accurate Triplet Extraction:** Ensuring that the LLM generates precise and schema-compliant relationships without introducing hallucinated entities. From my experience local llms often hallucinate in these types of heavy and high context involving process.
- **Entity Normalization at Scale:** Preventing duplicate or fragmented nodes in a large and diverse dataset (plugins, errors, versions, discussions).
- **Community Detection Stability:** Maintaining meaningful and stable clusters as the graph grows in size and complexity.
- **Latency Management:** Balancing multi-hop reasoning and dual reranking with acceptable response time inside the Jenkins environment.
- **Integration Complexity:** Seamlessly integrating Kuzu-based graph traversal and vector retrieval into the existing Jenkins chatbot plugin without disrupting current functionality.

References

- Jenkins Documentation
  https://www.jenkins.io/doc/
- Jenkins Plugin Development Guide
  https://www.jenkins.io/doc/developer/
- Kuzu Graph Database
  https://kuzudb.github.io/docs/
- FAISS (Facebook AI Similarity Search)
  https://github.com/facebookresearch/faiss
- Lewis et al., 2020 – *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*
- Traag et al., 2019 – *From Louvain to Leiden: guaranteeing well-connected communities*

Language Skill Set

- **Python — Proficient**
  Used for backend development, knowledge graph construction, and implementing GraphRAG pipelines.
  I also have a good foundation in the following GenAI-related libraries and tools:
  - **NetworkX** (graph modeling and traversal)
  - **SpaCy** (NER and text processing)
  - **FAISS** (vector similarity search)
  - **Neo4j** (for graph visualisation and testing): Used it in my Nsure AI project to visualising the graph and testing for irregular graph structure (like hairball graph), so that I can tune the parameters of graph generation. graph_maker.py
  - **Scikit-learn** (clustering and vector operations)
  - **LLM API integrations** (Gemini and similar models for extraction and reasoning)
- **HTML/CSS —** Working Knowledge
  Used for structuring and styling frontend interfaces.