Distributed RAG Pipeline

Final Report

Team Feynman

Ruturaj Ramchandra Shitole (Team Lead)

Boxin Yang

Swarup Totloor

Under the guidance of

Prof. Roozbeh Haghnazar

The George Washington University

1. Introduction

Large Language Models (LLMs) have demonstrated usefulness in dealing with natural language problems, but are inaccurate and limited when deployed in real-world contexts. Retrieval Augmented Generation (RAG) is a recent technique that incorporates the missing information into the LLMs by using stored and indexed contexts. A RAG pipeline performs retrieval, augmentation, and generation in series to provide a more informed response. An important part of a RAG pipeline are the retrieval components – embedders and vector databases. A corpus of structured and unstructured data is processed, embedded, indexed, and stored in vector databases. Whenever a natural language query is made to the system, the RAG pipeline retrieves the relevant context by embedding the query and finding the relevant information associated with it in the vector database. This information is then added as additional context to the query received by the LLM.

With the development of newer and larger models, the RAG pipelines have grown in complexity, and distributing them across multiple nodes has become essential for the scalability, reliability, and efficiency of the underlying system. Some recent developments address these issues, like Milvus [10], a scalable and highly available distributed vector database. However, these solutions address them only in isolation. Our project aims to provide a more comprehensive solution, holistically tackling every component of a distributed RAG pipeline and its challenges, in particular, the fault tolerance and scalability of the pipeline.

The motivation behind this project stems from the growing need for intelligent, scalable, and secure conversational AI systems that can efficiently process and retrieve information from large, distributed document collections. As organizations increasingly rely on digital knowledge bases, there is a demand for solutions that can provide accurate, context-aware responses to user queries while maintaining data privacy and operational reliability.

The primary objectives of this project are to design and implement a distributed Retrieval-Augmented Generation (RAG) system that leverages modern language models and vector databases for efficient document retrieval and response generation, to build modular and scalable microservices for document processing, language model inference, and user interaction, enabling independent development and deployment, to provide a user-friendly frontend interface that allows seamless interaction with the backend AI services, and to facilitate easy deployment and scaling of the system using containerization and orchestration technologies such as Docker and Kubernetes.

These objectives guide the design and implementation choices throughout the project, aiming to deliver a robust and extensible conversational AI platform.

2. Related Work

Transitioning from traditional RAG pipelines to distributed ones introduces a range of technical challenges. Our literature review presents several noteworthy approaches for addressing scalability, fault tolerance, heterogeneity in data processing, and system efficiency. These approaches have helped us understand the current state of RAG pipelines and design our system accordingly.

Scalability is a foundational requirement for distributed RAG pipelines due to the increasing size of document corpora and user query loads. Traditional RAG architectures often encounter performance bottlenecks, particularly during vector retrieval over large knowledge bases. To address this, studies proposed partitioning vector data across multiple nodes and sharding knowledge bases to distribute query load [2][4]. This horizontal scaling technique not only improves retrieval throughput but also facilitates parallel processing of queries. Multi-tiered indexing strategies further enhance scalability. For example, compressed representations of vectors can be maintained in GPU memory, enabling faster access while offloading metadata storage to CPU memory [9]. Additionally, tools like Milvus [10] and pgvector [4] provide distributed vector search capabilities with high throughput, allowing real-time systems to maintain performance under growing demands. These strategies are often deployed on Kubernetes, which automates container orchestration, autoscaling, and load balancing, thereby improving overall resource utilization [3]. Hybrid retrieval strategies have also been explored to enhance scalability and accuracy. Combining dense vector similarity with traditional sparse keyword search yields better recall and precision. Techniques such as RAG Fusion enable multi-source retrieval and prioritize contextually relevant documents before they are passed to the LLM [1][3].

Distributed RAG pipelines are particularly vulnerable to failures due to their reliance on multiple components—retrievers, vector stores, and LLM endpoints. Fault tolerance mechanisms are thus essential. One widely adopted approach involves introducing redundancy at both the system and data levels. Leader election and consensus algorithms, as implemented by systems like Apache ZooKeeper [6], ensure consistency and high availability by coordinating node roles and synchronizing metadata across the cluster. Snapshot isolation techniques provide consistency during read-heavy workloads, preventing conflicts in concurrent queries [10]. Load balancers such as NGINX, in conjunction with Kubernetes, play a critical role in detecting failures and redirecting traffic to healthy nodes. Moreover, replication strategies are used to maintain high availability and prevent data loss during node crashes or network partitions.

To further reduce fault propagation, systems must handle partial failures gracefully. This includes fallback mechanisms at each component level and retry logic for failed retrievals or LLM inferences. RAG pipelines deployed in production often include health checks and circuit breakers that disconnect faulty services before they affect downstream components.

Many RAG systems were initially designed for homogenous text input, limiting their ability to generalize to real-world applications that involve multimodal content such as images, tables, and structured data. As a result, recent works explore the use of multimodal transformers and cross-modal embeddings that unify various data types into a common vector space [5]. These methods improve retrieval across diverse document formats but also introduce latency and demand significant compute resources. In addressing this trade-off, researchers explore early fusion (embedding multiple modalities together) and late fusion (combining results after separate retrievals) strategies. While early fusion offers better semantic coherence, it typically requires retraining large models, whereas late fusion allows more flexibility and system modularity.

Efficiency in distributed RAG pipelines is primarily constrained by compute and memory resources. Given the high cost of deploying LLMs and dense retrieval systems, various optimization techniques have been introduced. Meduri et al. [2] propose knowledge distillation and quantization methods to compress models, reducing memory footprint while preserving accuracy. These techniques are particularly beneficial for edge deployments or GPU-constrained environments. Approximate Nearest Neighbor Search (ANNS) techniques, such as HNSW and PQ (Product Quantization), are commonly used to accelerate vector search operations at scale [4]. Though these methods introduce a slight loss in precision, the trade-off is acceptable for most RAG applications, especially when compensated by re-ranking strategies. Other efficiency improvements involve I/O optimizations like redundancy-aware deduplication, which reduces repeated data transfers, and GPU-CPU collaborative filtering to leverage heterogeneous compute resources for large-scale search tasks [9].

3. Project Design

We set out to build a distributed RAG system that would handle both chat-based queries and document uploading for additional context, all while remaining scalable, observable, and fault-tolerant. At the core of the system is **LangChain**, which acts as the coordinating layer for our pipeline. The client has two primary modes of interaction:

1. **Chat with the LLM**

Users can interact with the LLM via a chat interface frontend. The frontend application routes the query to the langchain component, which then interacts with the LLM and returns a streaming response that is relayed by the frontend application to the user.

2. **Upload Documents for Context**

Users can also upload text documents with a chunking strategy of their choice. These documents are sent to a dedicated document processor service, also powered by LangChain, but isolated from the chat backend. The document processor reads the user-uploaded documents and chunks them using the chunking strategy. Then it interacts with the embedder to fetch the embeddings for each chunk and uploads it to the vector database (Milvus).

We use Qwen, a powerful and light open-source LLM from Ollama, which makes running LLMs more manageable and container-friendly. Ollama provides the runtime environment with a REST interface, while LangChain constructs the prompts and handles the RAG orchestration, making the entire inference process modular.

For the vector database, we chose Milvus due to its performance in high-dimensional vector similarity search and support for scalable and distributed deployments. When documents are embedded, their vector representations are stored in Milvus along with associated metadata. At query time, Milvus is queried using vector search to return top-k relevant chunks, which are then used in the final prompt generation step handled by LangChain.

Every client interaction, whether it's uploading documents or querying the model, is routed through an authentication service. This service validates user identity via API keys. If necessary, we can also apply rate limiting and usage policies and tag data with user-specific identifiers to ensure secure and personalized retrieval.

To track system performance and monitoring, we use **Langfuse** for tracing and analytics. Langfuse allows us to log every request and its associated RAG pipeline run, visualize LLM response times and retrieval quality, and catch any anomalies or slow queries. This is a critical tool in understanding how the system behaves in production and improving both latency and output quality.

Finally, everything is deployed in a Kubernetes cluster, as this gives us fine-grained scalability per component, fault isolation between services, as well as built-in health checks and auto-recovery.

We primarily have two types of services in our LangChain pipeline. Stateful services, which are the LangChain node, Data Processor, and Stateless services in the Langchain Component are the Web server, LLM, and Embedder. Stateful components are managed via Kubernetes Stateful sets. Kubernetes Stateful sets guarantee a sticky identity for pods in terms of network and storage. That means when the pod goes down, it is guaranteed to resume with the same storage and networking identity. Stateful sets manage Replica sets. Each replica set keeps a track of the number of pods for an application running at the moment. When a pod goes down, the state of the system changes and replica sets schedule a new pod to run. The benefit of using stateful sets with replica sets lies in the fact that when the pod is rescheduled, the stateful state guarantees the same identity of the new pod.

Stateless components are managed via Kubernetes Deployments. Kubernetes Deployments manage the replica sets which monitor the system for the number of pod replicas. If a pod goes down, the replica sets schedule a new pod to run again. If any of the other components that these pods depend on goes down, the pods implement an exponential backoff strategy. In this strategy, the pods keep retrying the requests with exponentially increasing timeouts.

3.1 Use-case Diagram

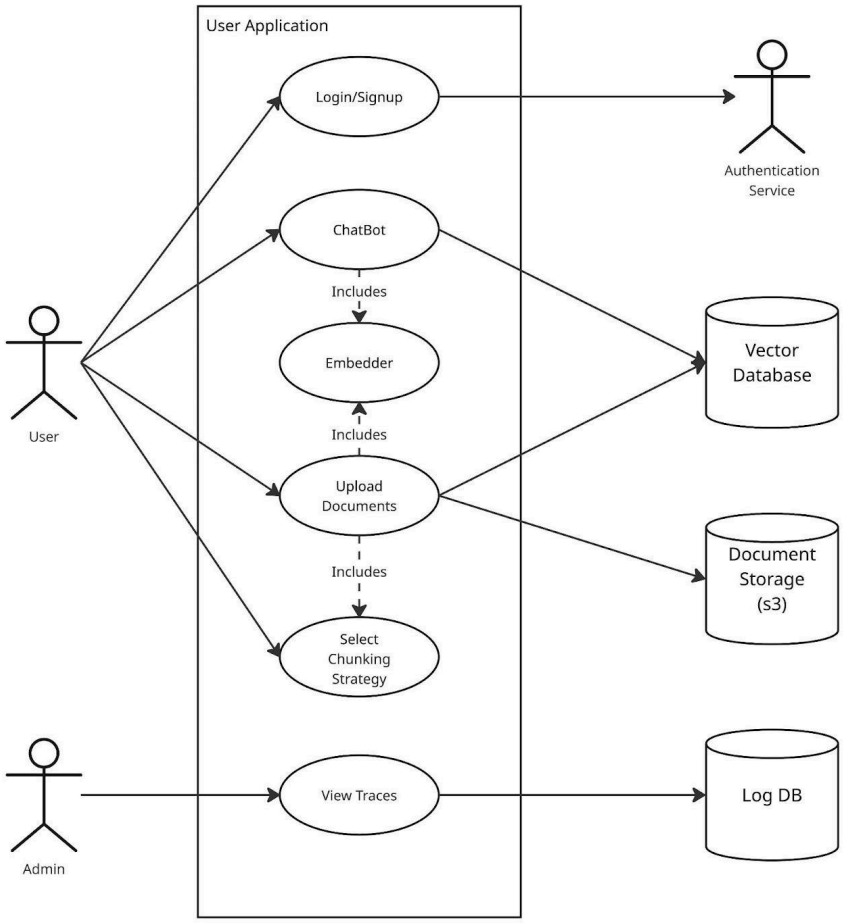

Fig 3.1: Use Case Diagram for RAG Pipeline

The figure above shows the use-case diagram for the RAG pipeline. On the left are the users of the RAG pipeline who interact with the system via the User Application. A user can Login/Signup, Chat, and Upload Document for processing. The User Application interacts with various systems and handls the RAG flow such that the entire pipeline is hidden from the user.

The admin is the maintainer of the pipeline who can view the performance of the entire system.

3.2 Sequence Diagrams

3.2.1 User Query Flow

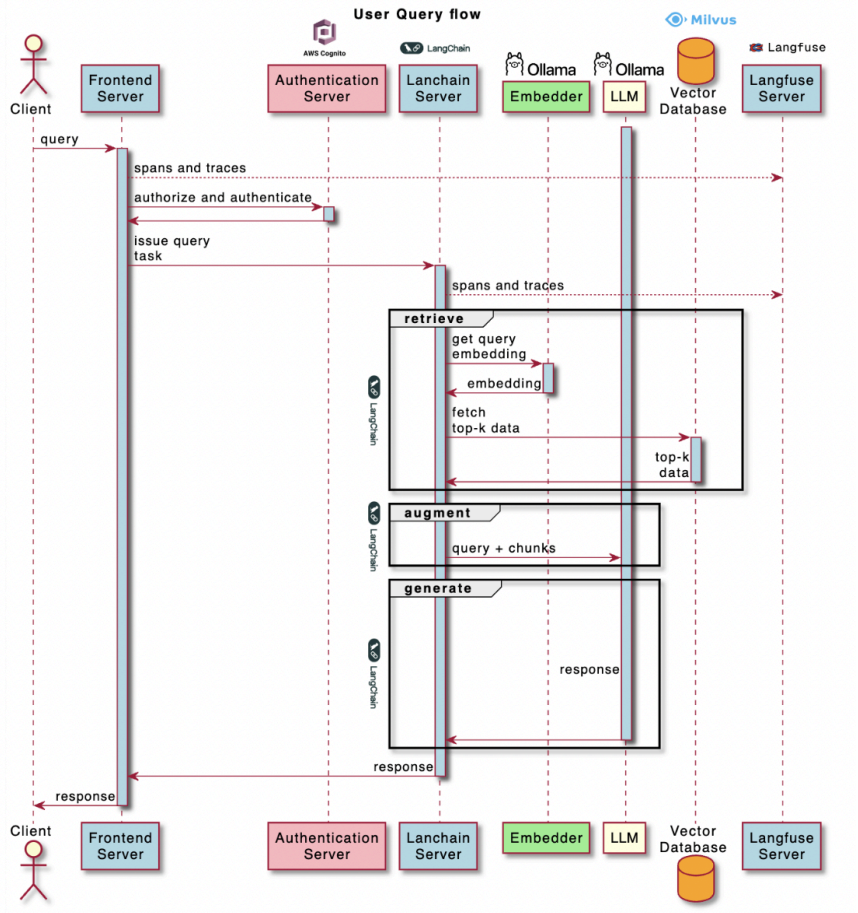

Fig 3.2: A sequence diagram of user query flow

This figure shows the series of interactions performed between the user and services as a query is initiated by the user. Once a user sends a query, the frontend component verifies if the user is authenticated to send a query. Then it sends the query over to the Langchain server and records a trace of the incoming request. In the retrieve phase the langchain server sends the query to the embedder to receive the embedded vector for the query. Then It talks to the vector database to fetch the K-closest vectors for the query vector and the corresponding text. Then in the augment phase, the langchain server adds the fetched text from the vector database as a context to the user prompt. Then, in the generation phase, the LLM responds, and the response is relayed back to the user. Throughout the flow, the traces and spans across various components are sent to the Langfuse server for better observability.

3.2.2 Document Upload Flow

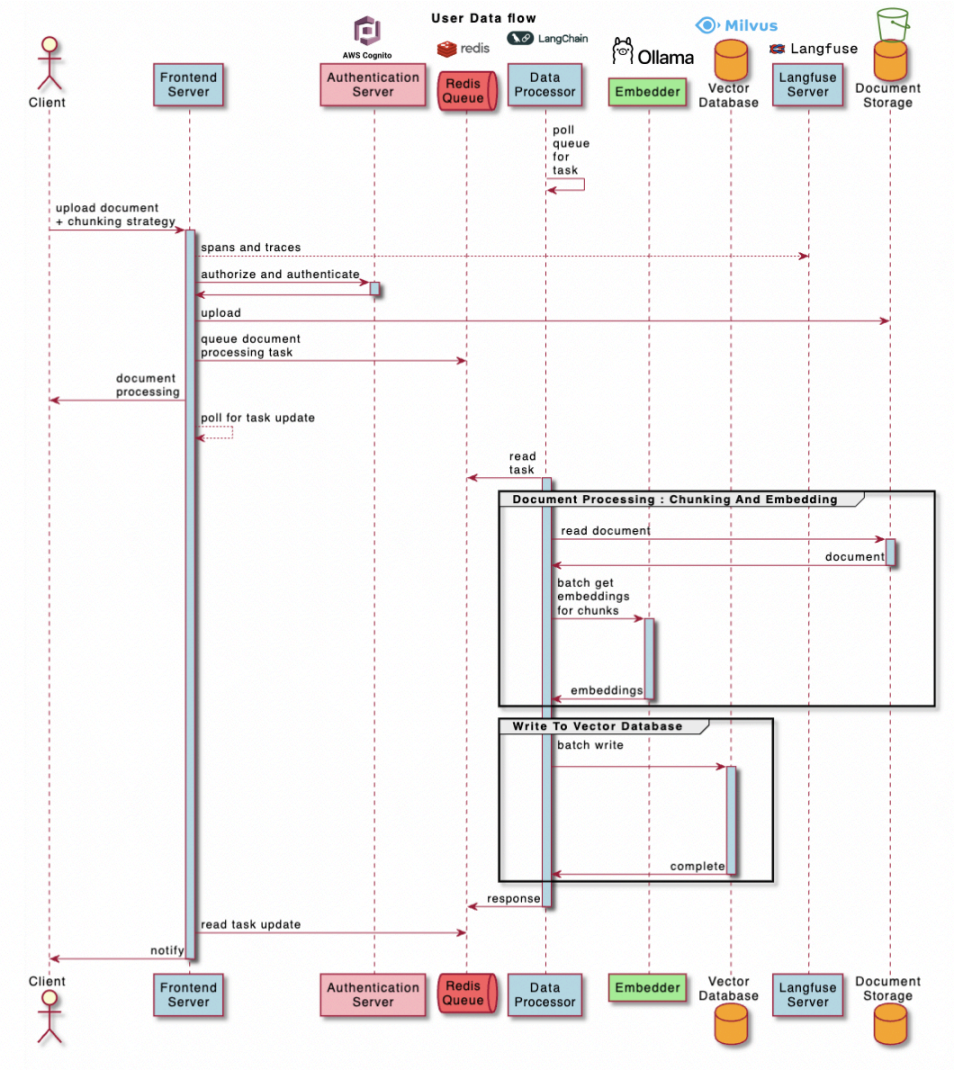

Fig 3.3: A sequence diagram of user document upload flow

This figure illustrates the sequence of interactions between user and services for the document upload flow. When the user uploads a document for processing and select a chunking strategy, the frontend component first verifies the authentication of the user and then uploads the document to a document store like S3. Then the frontend server queues a document processing task into a redis queue. The data processor is continuously polling the redis queue and once it reads the document processing task, it fetches the corresponding document from the document storage and chunks it based on the chunking strategy. For each chunk, it fetches the corresponding vectors from the embedder and uploads them to the vector database (Milvus).

3.2.3 Langfuse Traces Processing

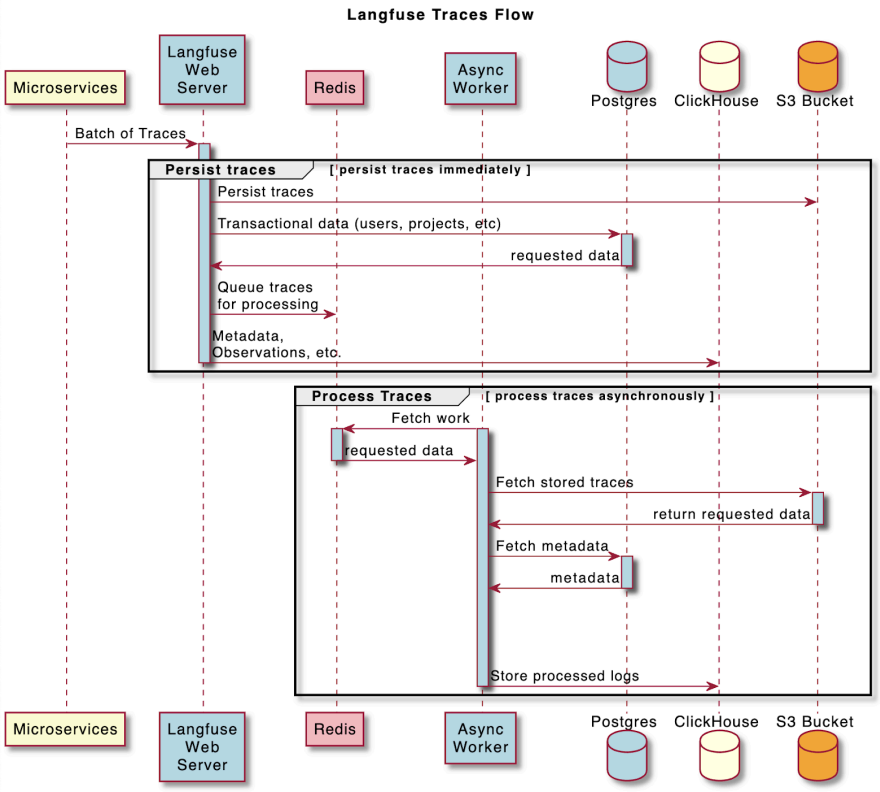

Fig 3.4: Langfuse Traces Processing

This diagram outlines a trace-processing system leveraging microservices for observability. Traces are initially persisted to S3 bucket and transactional data to Postgres, then queued via Redis. An Async Worker processes batches asynchronously: fetching metadata, enriching traces, and storing results in ClickHouse (analytics) and S3 (raw logs). The Language Web Server orchestrates initial trace ingestion, authentication, and client communication. Langfuse manages end-to-end observability, tracking spans/traces for performance monitoring and debugging. The architecture ensures scalable, decoupled processing with optimized storage (relational, analytical, object) for different data types.

3.2.4 Langfuse View Traces

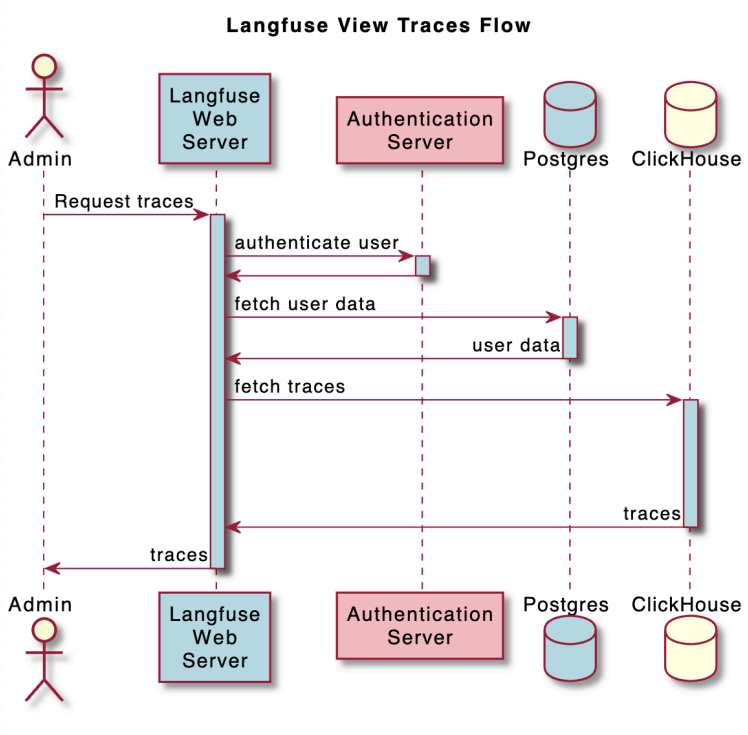

Fig 3.5: Langfuse View Traces

This diagram shows how maintainers of the rag pipeline can monitor the system using Langfuse to view the traces and spans. An admin initiates a request through the Langfuse Web Server, which first authenticates the user via the Authentication Server. User metadata (e.g., permissions, roles) is fetched from Postgres (transactional data), while traces (performance logs, spans) are pulled from ClickHouse (optimized for analytical queries). The integrated flow ensures admins can monitor system behavior, debug issues, and audit interactions while maintaining access control. Observability data is segregated: Postgres handles relational metadata, ClickHouse manages high-volume trace storage, and the Langfuse interface centralizes visibility.

3.2 Component Diagram

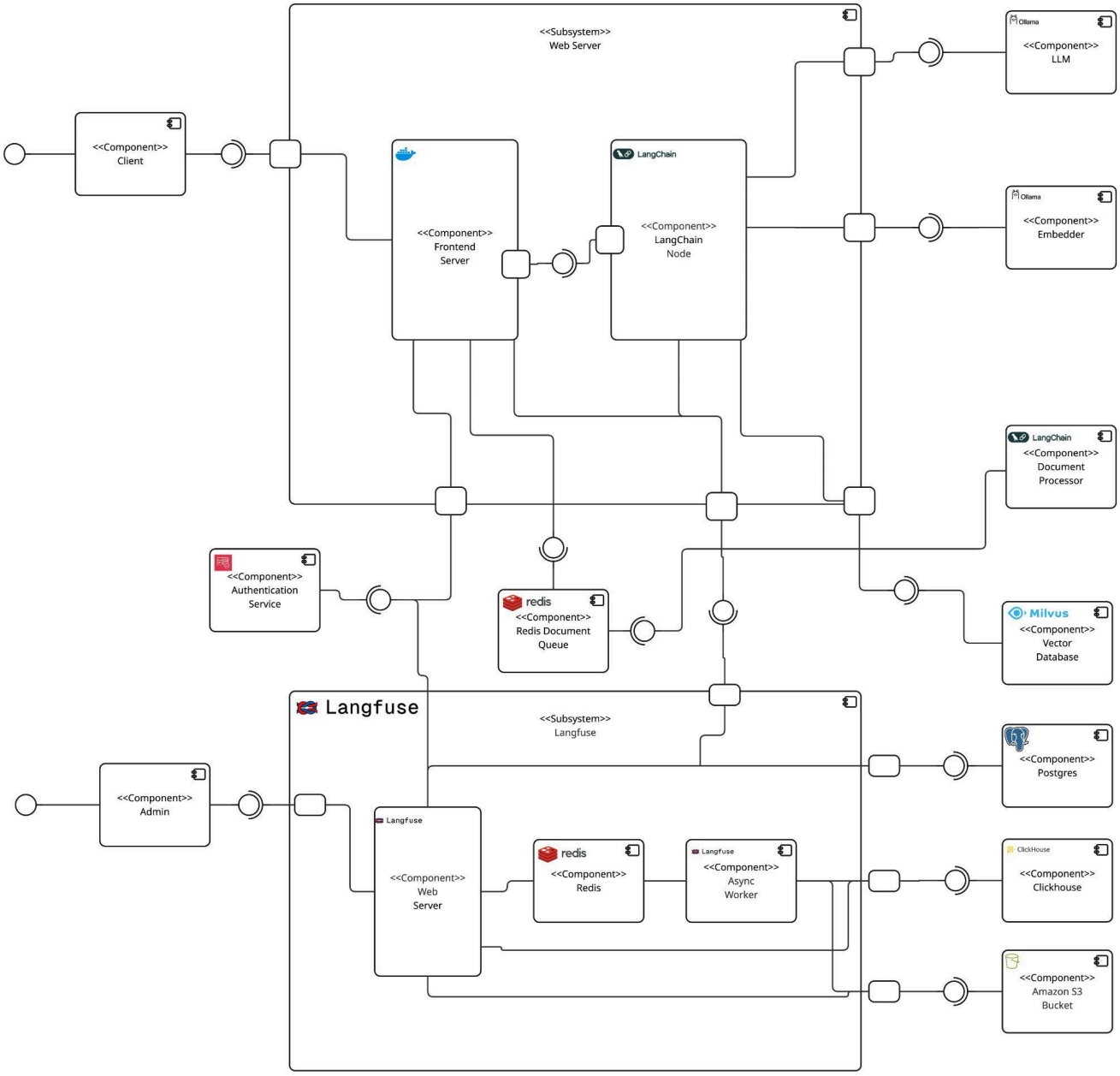

Fig 3.6: The system diagram

This diagram illustrates the component structure of the pipeline. It shows the interface each component provides to the other.

3.3 Risk Analysis

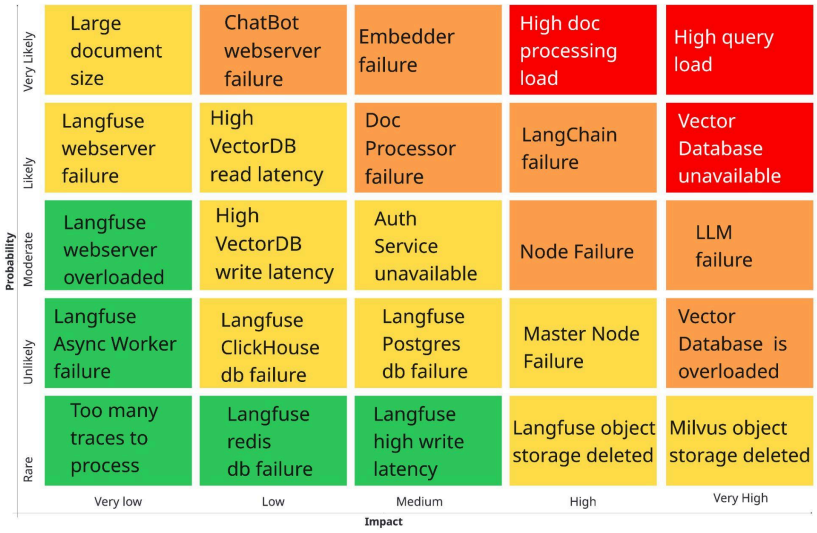

Fig 3.7: Risk analysis

This diagram shows the risk analysis for the system, we focus on the following part of risk analysis in this project.

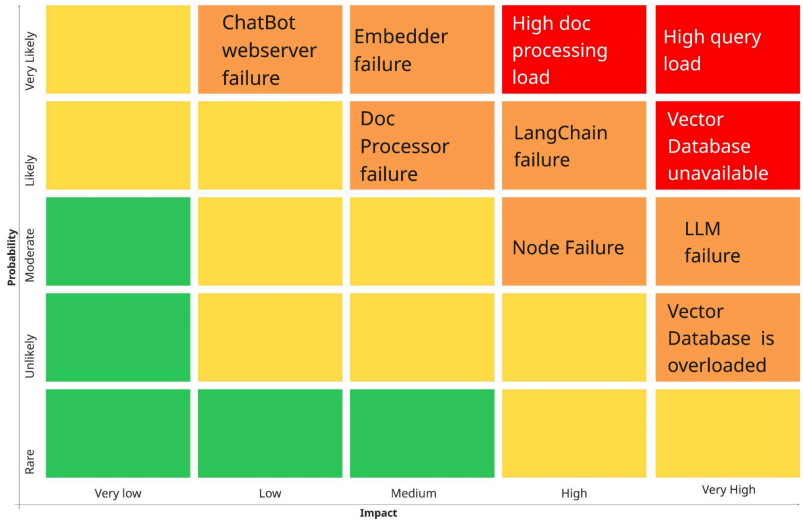

Fig 3.8: Risk analysis, target of this project

3.4 System Diagram

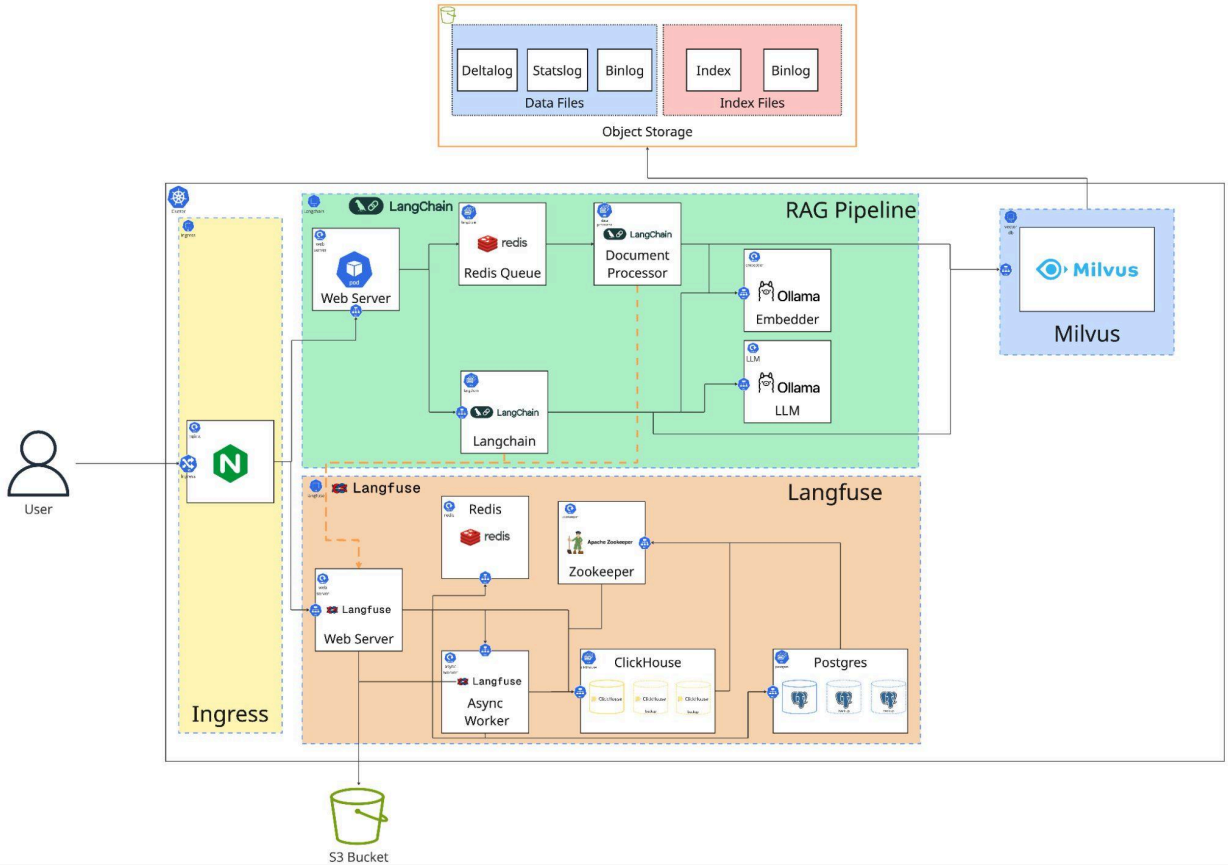

Fig 3.9: The system diagram

The figure above shows the system architecture as deployed on Kubernetes. We have isolated components into different namespaces as described below:

Ingress Namespace

This namespace contains the ingress controller and associated resources. The ingress controller is the entrypoint into the network and handles all incoming traffic. In this system we use the Nginx ingress controller. This helps us handle TLS termination at the ingress-level and removes the need for other components to handle that. The ingress controller maps the incoming request to the system to various services present in the cluster and forwards the request to the appropriate one.

RAG Pipeline Namespace

This namespace contains the components that handle the RAG flow of the pipeline. The components are as described below:

Web Server: This is the frontend server that handles the client requests. We use Next.js to write a server-side loading component and host it using Node.js. If the client submits a query to the web server, it is routed to the Langchain component. If the client submits a document for processing, this component uploads the document to a document storage like S3 and adds a task to the Redis queue.

Langchain: This component handles the RAG flow when a user submits a query.

Retrieval: It takes the user query and embeds it using the embedder, which returns a vector for the query. Then it uses the embedded vector and sends a query to the vector database to fetch the k-closest vectors and texts to the query vector.

Augmentation: The Langchain component filters out the vectors that are beyond a threshold distance and augments the texts from those vectors to the user query with the following format.

```
using the following as the context:
<context-start>
—chunks fetched from vector database—
<context-end>
answer the following:
—user prompt—
```

Generation: The augmented query is then sent to the LLM. The LLM returns a streaming response as the generation is taking place, the response is streamed back to the frontend, providing client with a smoother interface.

Redis Queue: This component holds the queue of tasks added by the web server for processing the documents.

Document Processor: This component continuously polls the redis queue and once a task is available, it dequeues it and saves the task details in the attached volume. It is a stateful component and is deployed with a volume attached to it. This ensures that in case the component goes down, it comes back up with the same volume identity.

The processing task includes:

- Chunking the document by the chunking strategy
- Calling the embedder to get the vectors for all chunks
- Storing the vectors in the vector database

Embedder: This component is responsible for receiving a request for embedding a text and returning the vector. The vector returned by it is of length 384. For this project, we use ollama's container image to host the embedder and use the all-minilm embedder with 33 million parameters.

LLM: This component handles the Generation part of the RAG pipeline. It receives the query and streams the response back to the component requesting.

Milvus Namespace

This namespace handles the components required for running Milvus.

4. Distributed Systems Challenges

When designing such a RAG system, we planned to address the primary concepts of any distributed system. Given their importance, we decided to focus mainly on **scalability** and **fault tolerance** as the challenges to solve. Our system is also **transparent**, since the user and each component in our system are not aware of the distributed nature of the other components. Our system is also **concurrent,** as the NGINX ingress is able to handle a large amount of concurrent incoming traffic. Using Kubernetes as our orchestrator also allows us to potentially integrate other aspects of a distributed system like **heterogeneity** and **security**. Below we focus on the two primary aspects of our system.

4.1 Fault Tolerance

Our services are logically divided into stateful and stateless components to manage recovery and persistence effectively. The LangChain server and the Document Processor are stateful and deployed using Kubernetes StatefulSets. These components store internal state in PersistentVolumes, preserving pipeline progress across pod restarts. When a pod fails, StatefulSets ensure a new pod is spawned with the same network identity and persistent volume, enabling in-place recovery. For external dependencies, such as vector DBs and LLMs, these services implement exponential backoff for retries, gracefully handling temporary outages.

The Web Server, Ollama-hosted LLM, and embedding service are stateless and run via Kubernetes Deployments, ensuring quick pod recreation on failure via ReplicaSets. Early in development, our use of Milvus on local hardware (without SSDs) caused performance degradation. Milvus requires ≥500 IOPS and <10ms latency for reliability. Our system's IOPS was well below the required threshold, leading to "etcd" instability and frequent Kubernetes cluster leader elections. Milvus Lite was tested but showed inconsistent behavior during vector search. Our Resolution? We moved to Zilliz Cloud, a fully managed Milvus service that abstracts away infrastructure issues, providing stability and better fault handling.

4.2 Scalability

To scale effectively with traffic and resource availability, we configured Horizontal Pod Autoscaling (HPA) for both stateless and stateful components based on CPU and memory usage. Kubernetes dynamically increases or decreases the number of pods depending on current load and available resources. Specific resource-aware decisions we made include the Ollama LLM Deployment and our Milvus Vector DB. Since we do not have GPU nodes in our cluster (due to resource constraints), our performance does not scale too well with a high throughput. We run Qwen 0.5B in a CPU-only Ollama container with 10 CPUs and 20Gi memory allocated per instance. While this setup provides acceptable functionality, response times remain relatively high due to lack of GPU acceleration.

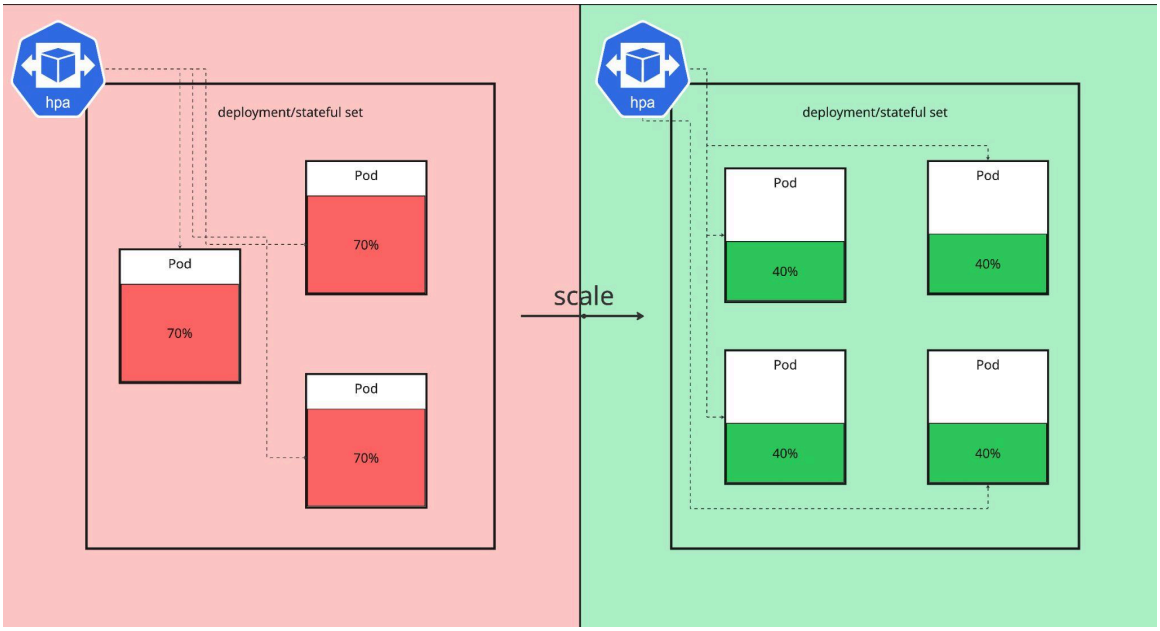

Fig 4.1: Pod scaling in deployment

5. Project Outcomes and Reflections

5.1 Project outcomes

This section describes the execution and user interface for the RAG pipeline. We show how the RAG chatbot can be used for querying and interacting with the LLM and uploading a document to provide a context for querying. We also show how Langfuse is helpful in understanding the scenes behind the RAG pipeline. It shows the input and output for each stage, along with the time it took to execute.

5.1.1 The user interface

The RAG user interface

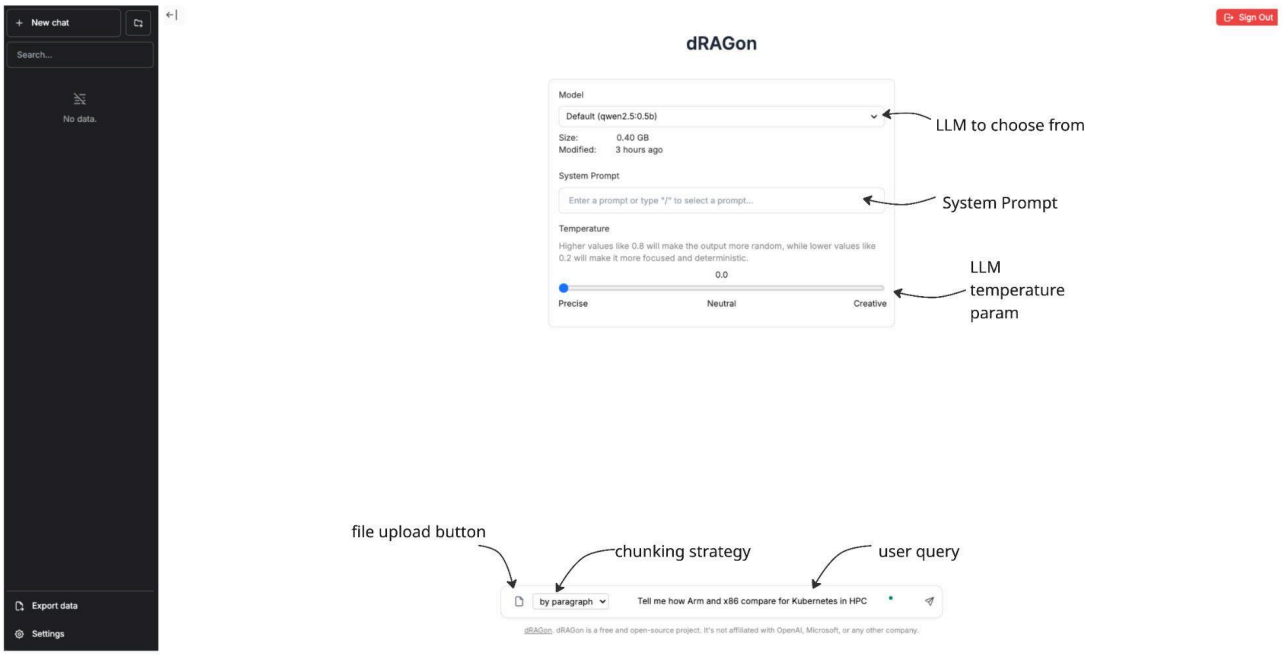

Fig 5.1: The user interface

This figure shows the user interface, where the user is allowed to choose from a variety of options as marked in the figure.

The Langfuse interface

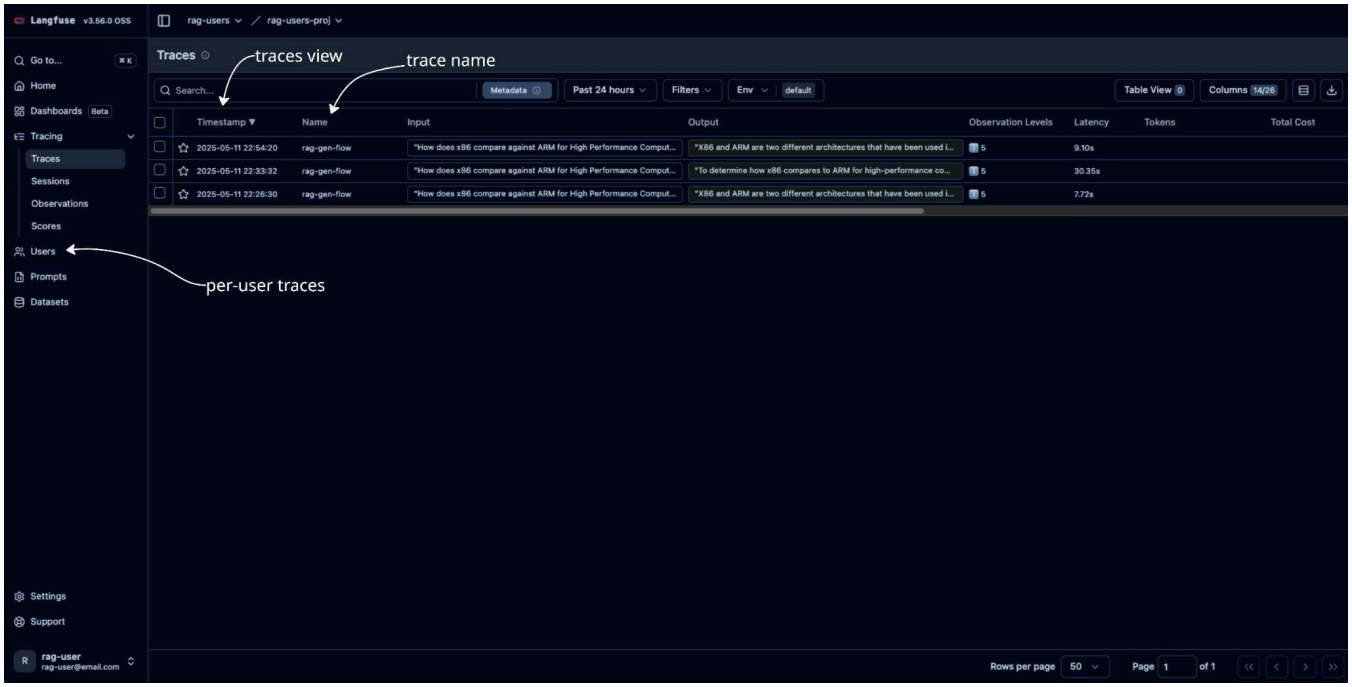

Fig 5.2: The Langfuse interface

This figure shows the Langfuse interface for traces. It shows the options available to the admin for viewing different traces within the system.

The Langfuse trace view

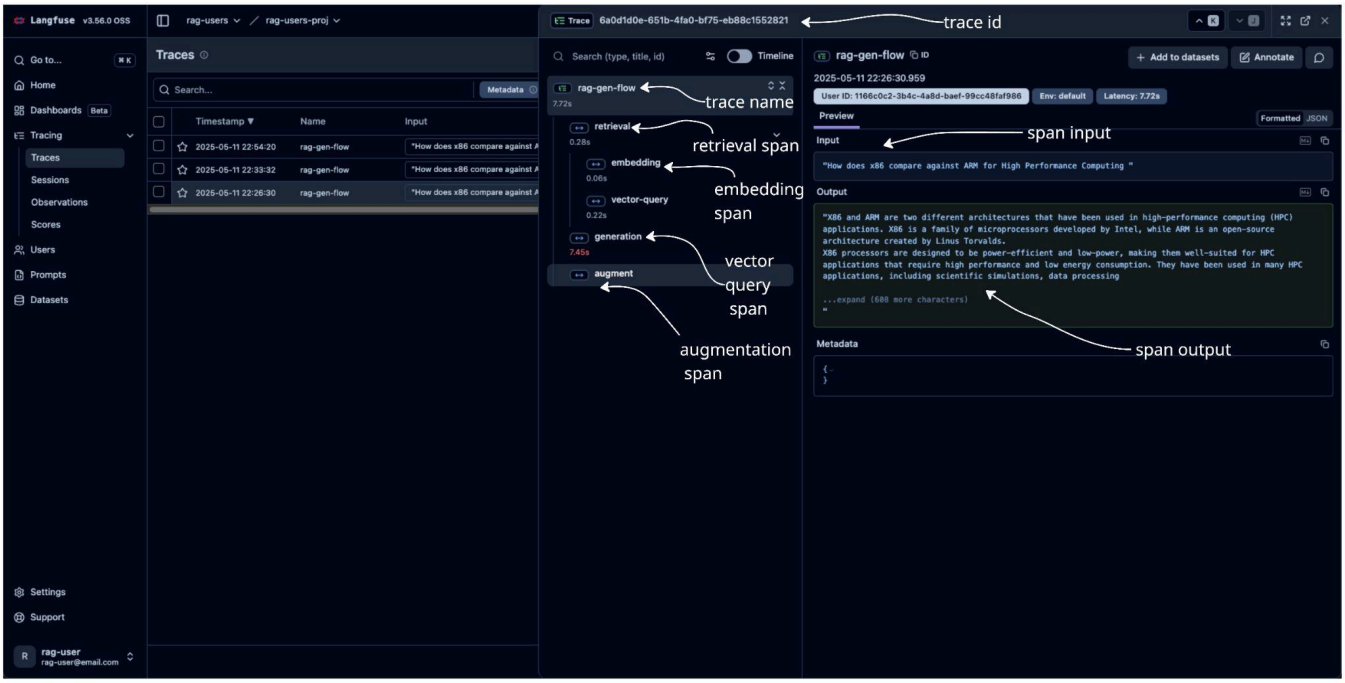

Fig 5.3: The Langfuse trace view

This figure shows the Langfuse interface for traces. It shows how the trace for the RAG pipeline looks like. Each trace is an end-to-end execution within the Langchain component and consists of multiple spans retrieval (embedding and vector-query), augmentation, and generation. Each span has data related to the input and output.

5.1.2 Querying without any context

Querying without context and response

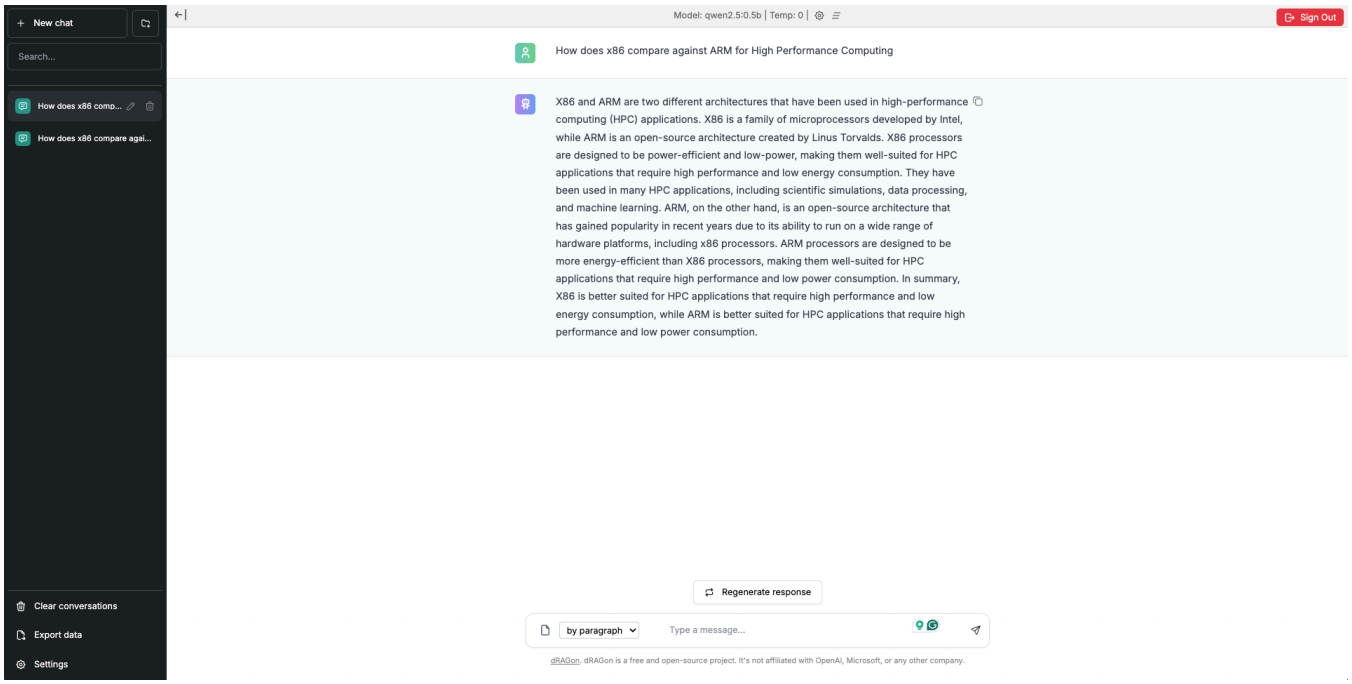

Fig 5.4: Querying without a context

This figure shows the running of a query without the LLM having rich context. The response returned by the LLM is wrong (for example, the claim that *ARM is an open-source architecture created by Linus Torvalds*). This shows how LLMs may hallucinate when queried on specific information.

Langfuse analysis for a query without context

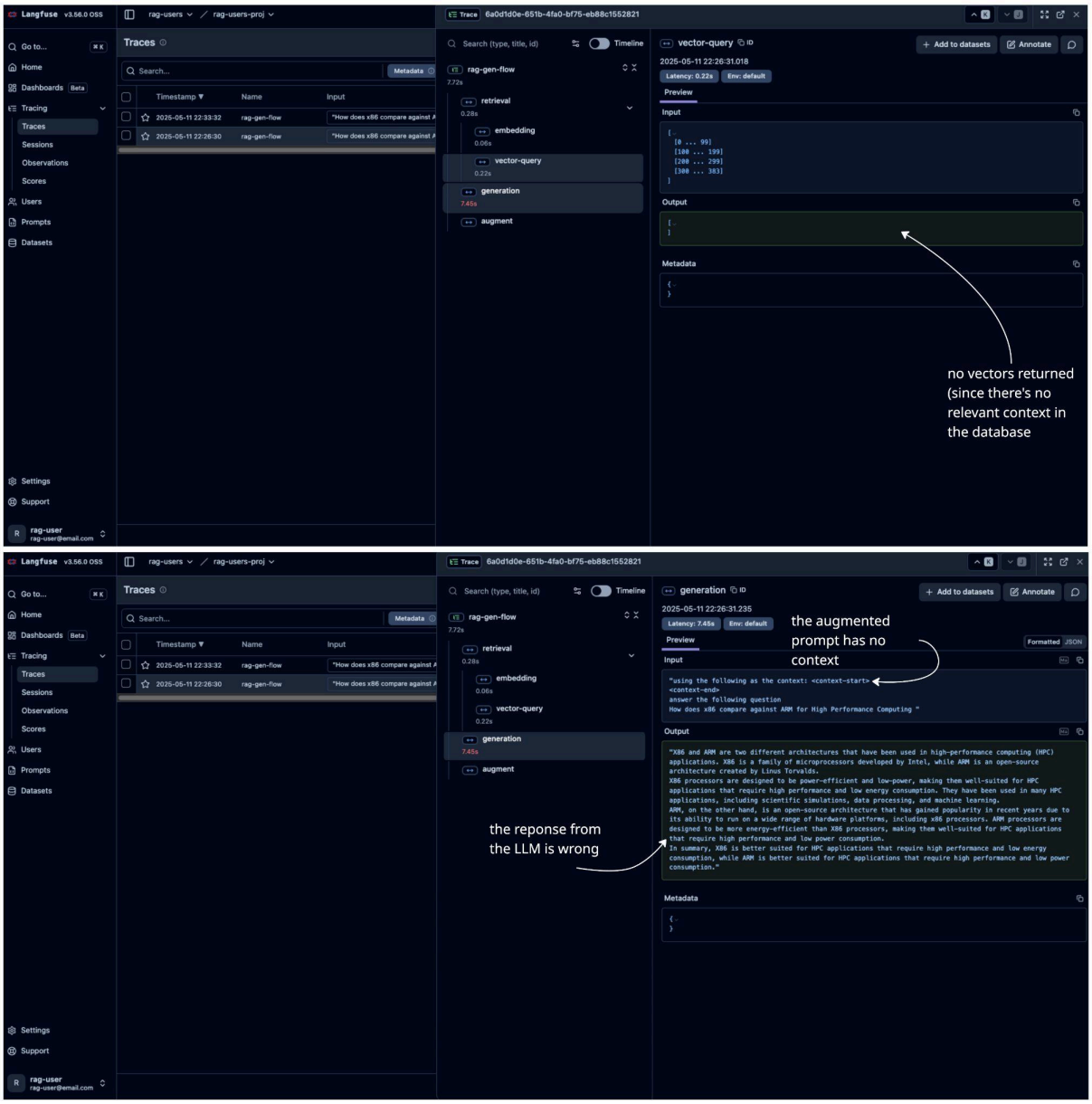

Fig 5.5: Querying without a context

This figure shows the behind-the-scenes when the LLM is queried without any context. The fetch from the vector database returns an empty response. The LLM responds with the wrong response as it does not have a context to work with.

5.1.3 Querying with context

Uploading a document for processing

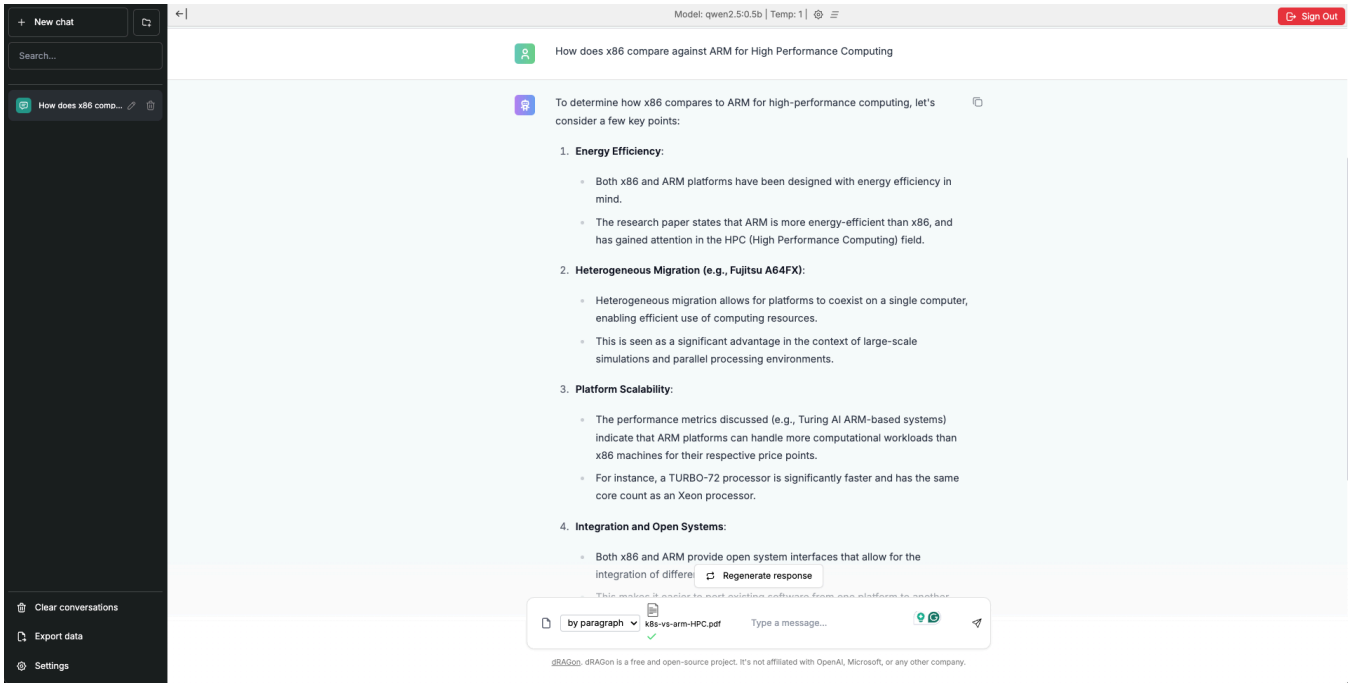

Fig 5.6: Uploading the document

This figure shows the result when a document is uploaded to the UI. The response this time is more informed than before. The chunking strategy used here is to split the document by paragraph and then upload the embedded vectors for each chunk to the vector database.

Document processor logs

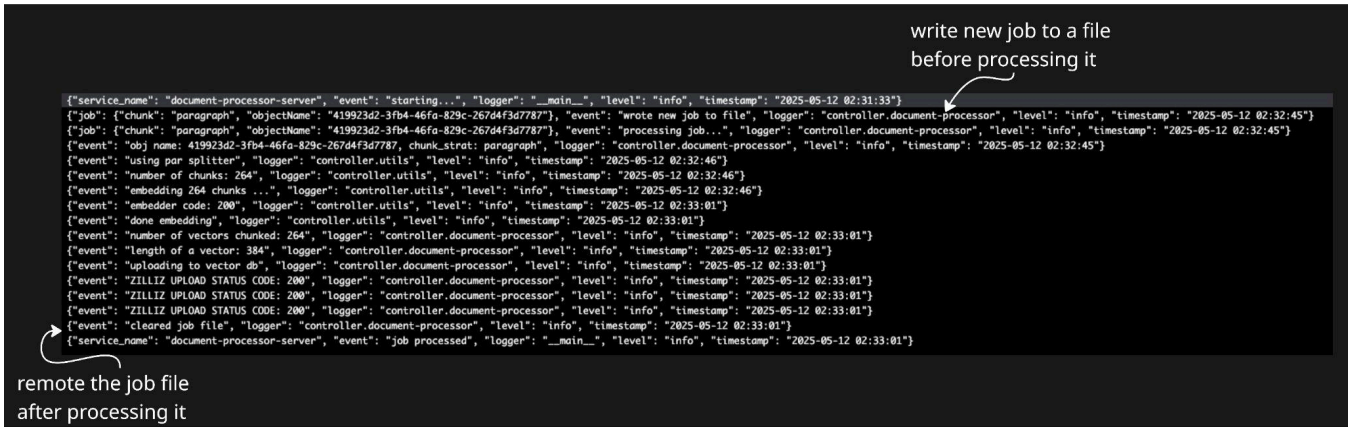

Fig 5.7: Document processor logs

This figure shows the logs from the document processor when a document is uploaded on the web ui. The web ui uploads the document to object storage and schedules it for processing by queuing the document in a Redis queue. The document processor, once it picks up the task, dequeues it and stores it in a file attached to the volume. After storing the file, it chunks the document by the chunking strategy and calls the embedder for embedding the chunks. It calls embedder in batches of 500 chunks per request and uploads them to Milvus 100 chunks at a time.

Langfuse traces

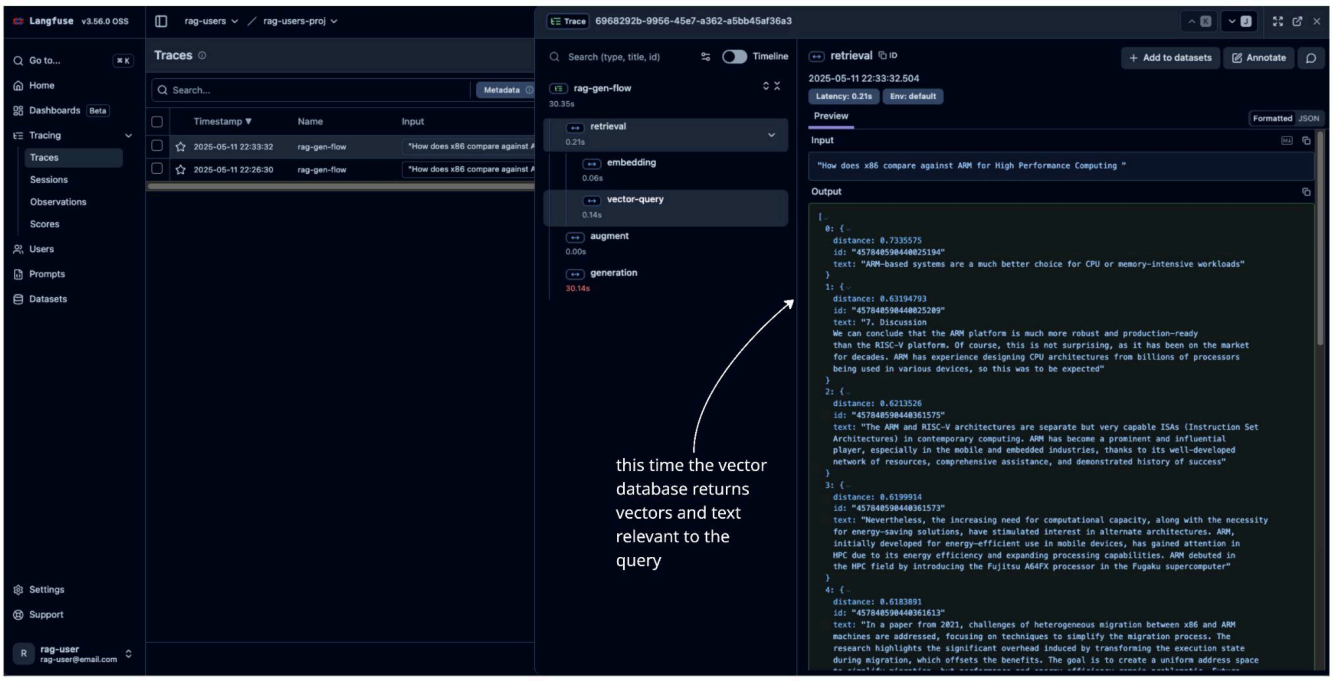

Fig 5.8: Langfuse Trace

This figure shows the langfuse trace after the document was processed and uploaded to the vector database. It shows that during the retrieval process, we now receive more data from Milvus. The vector database returns 10 vectors with their text and the distance from the query vector. We use the distance attribute of each to filter the text chunks across a certain threshold. Filtering this way ensures that not all the vectors from the database are used in augmentation as they may have irrelevant information.

5.2 Handling Fault Tolerance

A distributed RAG pipeline has multiple components that interact with each other. In a typical microservice architecture a component may error out from time to time. This can lead to cascading failures in dependent components. Therefore, it is important to address handling of failures in a distributed system.

Fault tolerance with Kubernetes Deployment/Statefulsets

We utilize the Kubernetes deployments and statefulsets for deploying our applications. These kubernetes resources control the replica sets, which make sure that a given number of replicas are always running for a pod of a specific type.

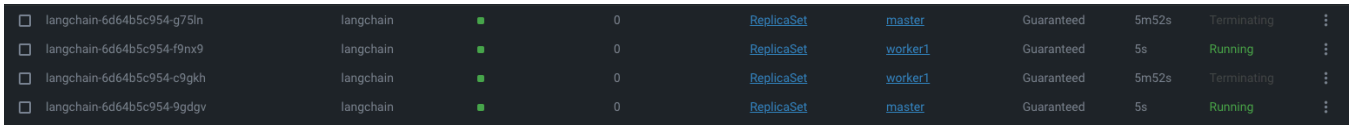

Fig 5.9: Keeping number of pods up

This figure shows that when pods running an application are terminated, Kubernetes instantly spins up more pods to balance the number of running replicas of the pod. The pods with the Terminating status were manually deleted, and the Pods with the Running status were the new pods triggered up by the replication controller.

Retrying requests on Failure

Since a component may go down and be spun up by the Replica Set, the component dependent on it may have initiate a request which could fail while the component is coming back up. We handle this using retry logic in our code. The following function is used to perform an exponential backoff between retries till an expected status code is received.

```python
from functools import partial
from typing import Callable

def retry_with_exp_backoff(request_func: Callable, expected_status_codes: list) -> requests.Response:
    wait_time = 2
    while True:
        response = request_func()
        if response.status_code not in expected_status_codes:
            _logger.info("received unexpected status code, retrying with backoff",
                         status_code=response.status_code, backoff_duration=wait_time)
            time.sleep(wait_time)
            wait_time = wait_time * 2
            if wait_time > 600:
                response.raise_for_status()
        else:
            _logger.info("received expected response")
            return response
```

Fig 5.10: Retry with exponential backoff

This figure shows an example function that accepts a callable request function and a list of expected status codes. It initiates the request and validates the response code, retrying till it reaches one of the expected status code. However, if the wait time goes over 600 seconds, it raises an error for the response it received.

Retrying document processing failures

The document processor is a stateful component. It dequeus the task from the Redis queue and stores the task in a task file within the attached volume. Whenever the document processor goes down, kubernetes Statefulset ensures that it comes back up with the same attached volume. This ensures consistent identity of the

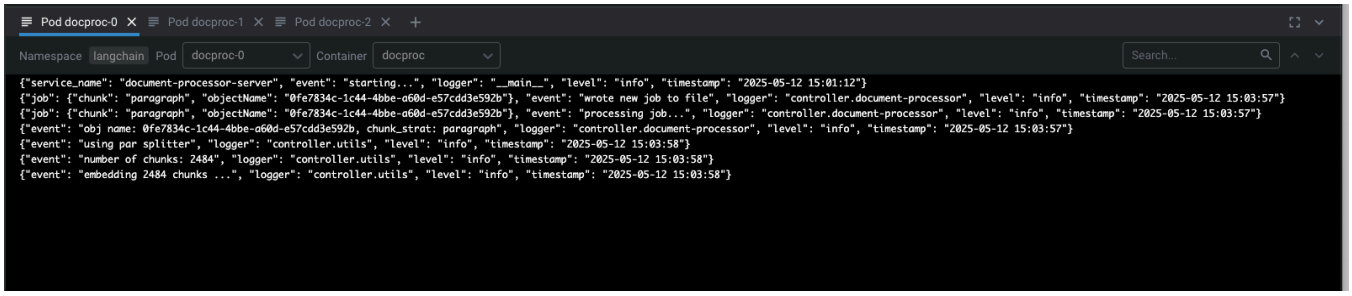

Fig 5.11: Document processor running a processing job

This figure shows logs of the document processor pod, that show that it saves the state before processing the job.

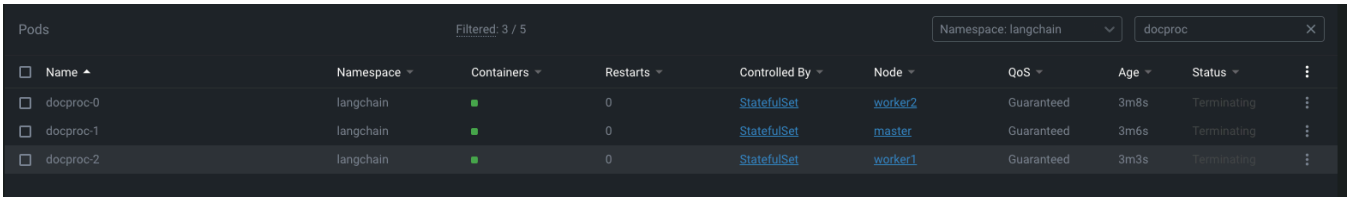

Fig 5.12: Terminating document processor mid task

This figure shows the termination of the document processor pod in the middle of a task. All the replicas are deleted form the cluster.

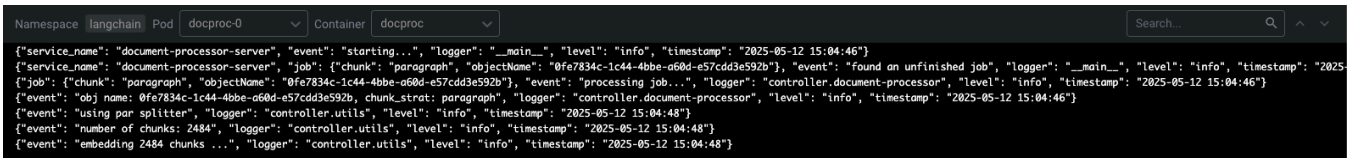

Fig 5.13: Document processor resuming task

This figure that the document processor resumes the task after it has come back up. The log in the second line says *found an unfinished job*, and then it starts processing the job.

5.3 Handling Scalability

Scaling with Horizontal Pod Autoscaler

We use Kubernetes Horizontal Pod Autoscalers (HPA) to scale our application dynamically. Each HPA targets a specific deployment/statefulset. It contains metadata about minimum and maximum number of replicas that the deployment should maintain. When the resource utilization goes beyond the threshold specified in the metrics, it starts to schedule more replicas of the pod.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: langchain
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: langchain
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 60
```

Fig 5.14: Langchain HPA configuration

This figure shows the configuration for the Horizontal pod autoscaler for the Langchain component. It monitors the cpu and memory utilization, and targets the average utilization of 50 and 60 percent for cpu and memory each.

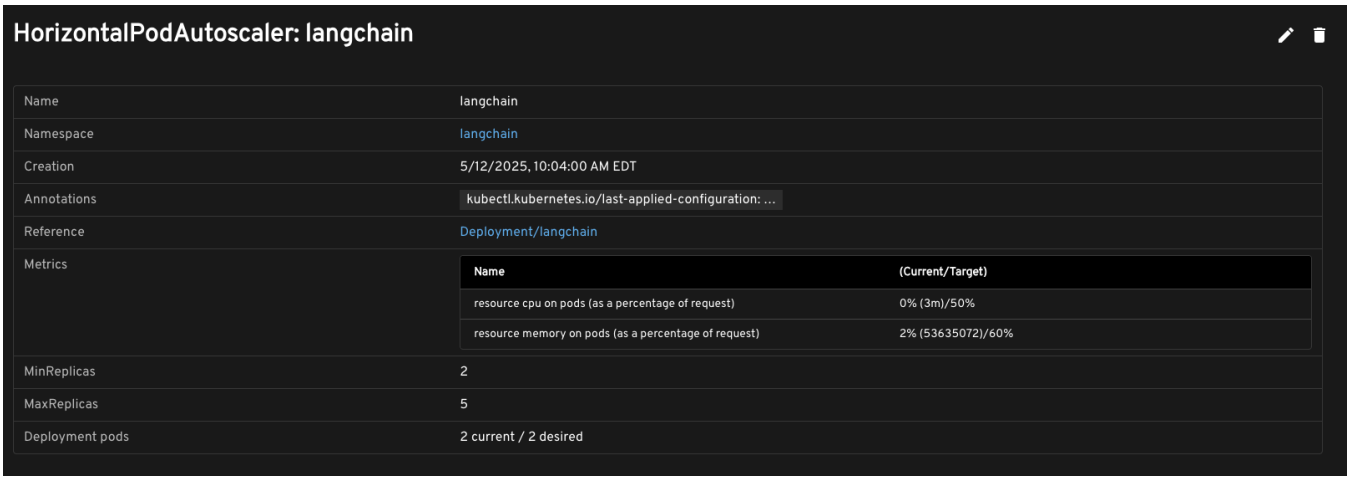

Fig 5.15: Langchain HPA

This figure shows langchain Horizontal Pod Autoscaler. This HPA is continuously monitoring the state of the the langchain pods.

Adding more nodes to the cluster

Fig 5.16: Nodes in the Cluster

As the load grows, more nodes can be added to the Kubernetes cluster, and the number of nodes with the control plane role can be increased for increased fault tolerance. This would ensure that the kubernetes cluster is able to maintain it's state in the event of control-node failures.

5.3 Reflections

Through the development of a distributed RAG pipeline we were able to provide fault tolerance and scalability over individual RAG components. Whenever a component fails, it is restarted within seconds and whenever the incoming load grows, the system scales horizontally by spinning more replicas of the pods. In case of request failures, we implement a retry logic with an exponential backoff. This lets us handl the errors in case the server component takes a while to come back up.

With the use of a RAG pipeline, the responses from the LLM are more contex-aware due to the augmentation of the closest vector chunks from the vector database. Because of the resource constraints, we were only able to run the LLMs on CPU. This limits the size of the LLM we can use. We used Qwen LLM with 0.5 Billion parameters (pretty small for an LLM). Even with this, the reponses generated were more helpful than without any augmentation.

The Ollama server allows interacting with the LLM using a REST interface, and provides a streaming response back to the client as the LLM is generating it. We are able to stream the response back to the user, this makes the entire query operation more seamless and provides a better user experience.

Batching and embedding the vector writes helped us reduce the number of requests flowing through the system, as every vector has 384 floating-point elements, and sending individual requests would have increased the latency for user-related operations.

This system can be made more reliable by implementing better retrieval techniques so that the LLM is more context aware, with fewer number of tokens passing through. The Redis queue can be made more scalable and fault tolerant with a more complex and robust solution like Kafka. Though we are using the host volumes, a network file system would be more helpful in case the node running the job goes down. This would help schedule another node with the same volume identity. Using GPU nodes will make it faster.

Overall, building this project gave us plethora of insights on how complex the real-world systems are and how simple tools can be made more scalable and fault tolerant. It made us understand that the Machine Learning component is only a small part of the system and the infrastructure around it requires more complexity.

Lastly, a huge thanks to Cloudlalb [11] for providing the resources to run our Kubernetes cluster.

6. References

[1] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., ... & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997, 2.

[2] Meduri, K., Nadella, G. S., Gonaygunta, H., Maturi, M. H., & Fatima, F. (2024). Efficient RAG framework for large-scale knowledge bases. in IJNRD 2024, Volume 9, ISSN: 2456-4184

[3] Kabbay, H. S. (2024, July). Streamlining AI Application: MLOps Best Practices and Platform Automation Illustrated through an Advanced RAG based Chatbot. In 2024 2nd International Conference on Sustainable Computing and Smart Systems (ICSCSS) (pp. 1304-1313). IEEE.

[4] Han, Y., Liu, C., & Wang, P. (2023). A comprehensive survey on vector database: Storage and retrieval technique, challenge. arXiv preprint arXiv:2310.11703.

[5] Joshi, P., Gupta, A., Kumar, P., & Sisodia, M. (2024, June). Robust multi modal rag pipeline for documents containing text, table & images. In 2024 3rd International Conference on Applied Artificial Intelligence and Computing (ICAAIC) (pp. 993-999). IEEE.

[6] Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). ZooKeeper: Wait-free coordination for internet-scale systems. In 2010 USENIX Annual Technical Conference (USENIX ATC 10).

[7] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., ... & Dennison, D. (2015). Hidden technical debt in machine learning systems. Advances in neural information processing systems, 28.

[8] Menshawy, A., Nawaz, Z., & Fahmy, M. (2024, April). Navigating Challenges and Technical Debt in Large Language Models Deployment. In Proceedings of the 4th Workshop on Machine Learning and Systems (pp. 192-199).

[9] Tian, B., Liu, H., Tang, Y., Xiao, S., Duan, Z., Liao, X., ... & Zhang, Y. (2024). FusionANNS: An Efficient CPU/GPU Cooperative Processing Architecture for Billion-scale Approximate Nearest Neighbor Search. arXiv preprint arXiv:2409.16576.

[10] Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., ... & Xie, C. (2021, June). Milvus: A purpose-built vector data management system. In Proceedings of the 2021 International Conference on Management of Data (pp. 2614-2627).

[11] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., ... & Mishra, P. (2019). The design and operation of CloudLab. In 2019 USENIX annual technical conference (USENIX ATC 19) (pp. 1-14).