GitLab Handbook Chatbot – Project Overview & Technical Approach

- *https://lnk.ink/prRxR*
- *https://github.com/knavdeep152002/gitlab_chatbot.git*

Objective

We set out to create a production-ready chatbot that can accurately and reliably answer questions about GitLab's Handbook and Direction pages. By leveraging a scalable architecture, robust document ingestion, and a hybrid search approach powered by Retrieval-Augmented Generation (RAG), this chatbot delivers accurate, context-aware responses grounded in GitLab's documentation.

Key Components & Why They Matter

1. Worker-Based Ingestion Pipeline

To handle the ingestion and processing of GitLab's documentation efficiently, we built a pipeline using Celery and RabbitMQ for asynchronous, scalable task management. The pipeline is broken into stages to keep things modular and performant.

Why Workers?

- **Scalability**: Workers can be scaled independently (e.g., one fetcher, multiple processors) to match workload demands.
- **Fault Tolerance**: If one task fails, the pipeline keeps running, with retries to ensure reliability.
- **Async Efficiency**: Heavy tasks like fetching or embedding don't block the main API, keeping things responsive.

Pipeline Stages

- **File Fetching Worker (Single-Worker)**:
    - Regularly checks GitLab's repository for new or updated commits.
    - Identifies changes in the handbook and direction pages via GitLab repositories of respective pages, and observes the MD files.
    - Log files for processing in the database using checkpoints to track progress.

- **File Processing Workers (Multi-Worker)**:
    - Pull raw Markdown content from GitLab.
    - Clean and split content into manageable chunks using LangChain's text splitter.
    - Store chunks as document entries with metadata in the database.
    - Mark each chunk as PROCESSED in the checkpoint system.
- **Embedding Workers (Multi-Worker)**:
    - Generate embeddings for each chunk using a vector model (e.g., OpenAI or Gemini).
    - Store embeddings in PostgreSQL with pgvector for vector search.
    - Create GIN indexes (content_tsv) to enable fast full-text search.

2. Query Flow – Hybrid RAG Chatbot

To provide accurate and relevant answers, we implemented a hybrid Retrieval-Augmented Generation (RAG) approach, combining semantic understanding (via vector search) with precise keyword matching (via GIN full-text search).

How the LangChain Agent Works

- The chatbot uses a LangChain Agent equipped with a custom `hybrid_gitlab_search` tool.
- When a user asks a GitLab-related question, the agent retrieves relevant document chunks using the hybrid search tool and uses them as context to generate a response.
- For unrelated queries, the agent politely declines to answer, reducing unnecessary computation and preventing inaccurate responses.

Why Hybrid Search?

- **Semantic Search (IVFF index)**: Captures the meaning of queries, even if they're phrased differently.
- **Full-Text Search (GIN index)**: Ensures exact matches for specific terms or phrases.
- Together, they deliver highly relevant and accurate context for the LLM to generate answers.

Tech Stack

Here's the toolkit we used to bring this chatbot to life:

Layer	Tools / Libraries	Purpose

Ingestion	Celery, RabbitMQ, GitLab API	Sync and process files periodically
Database	PostgreSQL + pgvector + GIN	Store documents, enable hybrid search
LLM & RAG	LangChain, Gemini Pro (via LangChain)	Contextual reasoning for answers
Backend API	FastAPI	REST endpoints for chat and session management
Frontend	Streamlit	User-friendly chat UI with citations
Deployment	Docker, Docker Compose	Containerised setup for local or scaled use

Why PostgreSQL + pgvector?

We chose PostgreSQL with the pgvector extension for our database needs because it perfectly suits the scale and requirements of this project. Since GitLab's Handbook and Direction pages don't generate millions of vectors, specialised vector databases like Qdrant or Cohere seemed like overkill. PostgreSQL + pgvector offers several advantages for our use case:

- **Hybrid Search Support**: PostgreSQL supports both vector search (via pgvector with IVFF indexes) and full-text search (via GIN indexes) in a single system, simplifying our architecture and enabling efficient hybrid search.
- **Simpler Operations**: Managing updates, deletions, and reinsertions of changed document chunks is straightforward with PostgreSQL's robust querying and indexing capabilities.
- **Cost-Effectiveness**: By sticking with a single, well-supported database, we avoid the complexity and cost of integrating and maintaining multiple specialised systems.
- **Scalability for Our Needs**: While not designed for massive vector datasets, pgvector handles the moderate scale of GitLab's documentation efficiently, with room to grow if needed.

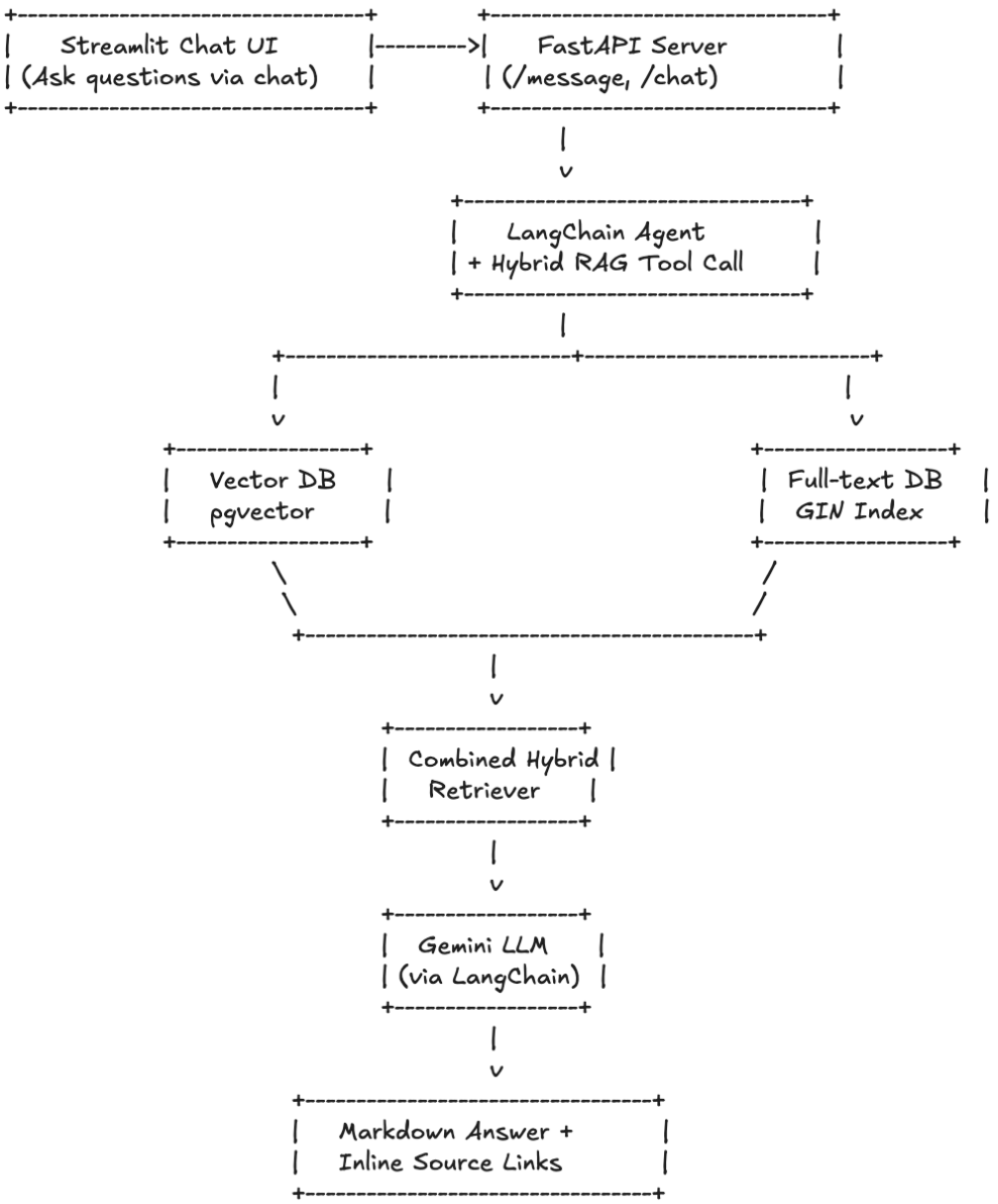
Periodic Sync Strategy

To keep the chatbot's knowledge up-to-date, we implemented a periodic sync process:

- Runs every 3 hours via Celery Beat.
- Uses GitLab's commit history to detect changes, avoiding redundant processing.
- Only processes new or modified files, embedding them as needed.
- Checkpoints ensure no duplicate processing or missed files.

Architecture Diagram

Here's a visual overview of how the components fit together:



Key Design Decisions

We made several deliberate choices to ensure the chatbot is robust and efficient:

Decision	Why
Checkpointing	Prevents duplicate processing and ensures reliable, idempotent ingestion.
Multi-worker ingestion	Speeds up processing and isolates failures (e.g., embedding crashes don’t affect fetching).
Hybrid search	Combines semantic and keyword search for better relevance and precision.
Tool-based RAG agent	Limits responses to GitLab-related queries, reducing hallucinations.
GIN + IVFF indexes	Leverages PostgreSQL’s strengths for fast, accurate hybrid search.
PostgreSQL + pgvector	Supports hybrid search, simplifies operations, and suits our data scale. (cost effective)
Streamlit frontend	Enables quick prototyping with a clean UI and support for citations.

Testing & Validation

We thoroughly tested the system to ensure it works as expected:

- Confirmed that each file is embedded only once, thanks to checkpoint integrity.
- Tested the chatbot with both handbook-specific and unrelated queries to verify accuracy and refusal behaviour.
- Validated that responses include accurate citations and relevant context.

Customization Possibilities

The system is designed to be flexible and extensible:

- Add support for ingesting PDFs or other file types.
- Swap in alternative LLMs like Azure, OpenAI, or Claude.
- Fine-tune the embedding model or modify the retrieval logic for specific use cases.
- Enhance the Streamlit app to support file uploads or multi-user sessions.

Conclusion

This GitLab Handbook Chatbot is a robust, scalable, and user-friendly solution for answering questions grounded in GitLab's documentation. By combining a modular ingestion pipeline, hybrid search, and LLM-powered reasoning, it delivers accurate and relevant responses while being adaptable for future enhancements. The choice of PostgreSQL + pgvector ensures efficient handling of our data scale and hybrid search needs, making this a practical and powerful template for document-grounded chatbot applications.

Note:

The Deployed URL is a bit slow due to a free-tier PostgreSQL connection from Supabase. So the api might respond 10x slower than what they do when set up properly.

Deployed URL:

https://rfnecubcdw9yap7visv2ce.streamlit.app/