Retrieval-Augmented Generation (RAG) Based Chatbot for NIT Jamshedpur

Submitted by

Abhinav Dev C	(2023UGCS099)
Chandrima Hazra	(2023UGCS017)
Hemanvitha Pullela	(2023UGCS001)
Om Raj	(2023UGCS022)
Pritom Roy	(2023UGCS044)

Under the Guidance of

Dr. Koushlendra Kumar Singh

Assistant Professor
Department of Computer Science and Engineering

National Institute of Technology Jamshedpur

Jamshedpur, Jharkhand, India

November 16, 2025

Abstract

This project presents a Retrieval-Augmented Generation (RAG) based chatbot designed to provide accurate and contextual information about the National Institute of Technology Jamshedpur by intelligently crawling and indexing its official website. The system addresses the challenge of fragmented information across web pages and PDF documents by implementing an intelligent assistant that delivers precise, source-grounded answers with relevant citations and streaming responses. The chatbot leverages a comprehensive pipeline: web scraping with Puppeteer, semantic embeddings via Cohere, vector storage in Pinecone, and natural language generation using Google Gemini.

The architecture implements a robust scraping mechanism with sitemap awareness, categorized page discovery, dynamic JSON/XHR parsing, and PDF link extraction. Content is chunked using LangChain's text splitter and embedded using Cohere's v3 model (1024-dimensional vectors), which are stored in Pinecone vector database with cosine similarity indexing. A sophisticated change detection mechanism using MongoDB maintains a ledger of content hashes per URL, enabling incremental updates and safe deletion of stale vectors without redundant processing. The system employs Google Gemini (gemini-2.5-flash) for response generation with Server-Sent Events (SSE) streaming, providing real-time user feedback.

To optimize performance and resource utilization, the implementation includes multi-layered caching strategies: an embedding cache using Redis-backed LRU for vector reuse, and an LSH-based semantic response cache that identifies similar queries to avoid redundant API calls. Additional features include Redis-backed rate limiting with memory fallback for abuse prevention, comprehensive admin endpoints for system monitoring and control, and a web-based dashboard for maintenance operations. The system successfully demonstrates how RAG architectures can transform institutional information access, providing students, faculty, and visitors with instant, accurate answers while maintaining source attribution and handling diverse queries ranging from admission procedures to faculty information and recent announcements.

Keywords: Retrieval-Augmented Generation, Natural Language Processing, Vector Database, Web Scraping, Semantic Search, Chatbot, Information Retrieval, Large Language Models, Cohere Embeddings, Pinecone, MongoDB, Redis Caching

Acknowledgment

We would like to express our sincere gratitude to all those who contributed to the successful completion of this project. First and foremost, we extend our appreciation to our project advisor, **Dr. Koushlendra Kumar Singh**, Assistant Professor, Department of Computer Science and Engineering, National Institute of Technology Jamshedpur, for his guidance, and also to our seniors, Unnati Chaurasia and Himanshu Pathak for their continuous support, and insightful feedback throughout the development of this Retrieval-Augmented Generation based chatbot.

We acknowledge the institution's official website as the primary data source for this project, which allowed us to create a practical solution.

We would like to thank our peers who provided valuable feedback during the testing phase of the chatbot, helping us identify edge cases and improve the user experience.

Abhinav Dev C
Chandrima Hazra
Hemanvitha Pullela
Om Raj
Pritom Roy

Contents

List of Abbreviations

Abbreviation	Full Form
RAG	Retrieval-Augmented Generation
NLP	Natural Language Processing
LLM	Large Language Model
API	Application Programming Interface
SSE	Server-Sent Events
REST	Representational State Transfer
DOM	Document Object Model
XHR	XMLHttpRequest
JWT	JSON Web Token
LRU	Least Recently Used

Technology-Specific Abbreviations:

Abbreviation	Full Form
Gemini	Google's Generative AI Model (gemini-2.5-flash)
Cohere	Cohere AI Embedding Platform
Pinecone	Pinecone Vector Database
MongoDB	MongoDB Document Database
Redis	Remote Dictionary Server (In-Memory Data Store)
Puppeteer	Headless Browser Automation Library
Express	Express.js Web Application Framework
Axios	Promise-based HTTP Client for Node.js
LangChain	Framework for LLM Application Development

Chapter 1

Introduction

1.1 Background

The exponential growth of information on institutional websites has created significant challenges in information accessibility and retrieval. Educational institutions, maintain extensive web presences with information distributed across hundreds of pages, PDF documents, notices, and dynamic content sections. Students, faculty, prospective applicants, and visitors often struggle to locate specific information efficiently, leading to frustration and repeated queries to administrative staff.

Recent advances in Natural Language Processing (NLP) and Large Language Models (LLMs) have opened new possibilities for intelligent information retrieval systems. Retrieval-Augmented Generation (RAG) has emerged as a powerful paradigm that combines the strengths of neural information retrieval with the generative capabilities of LLMs [**Ramesh2020**]. Unlike standalone language models that rely solely on their training data, RAG systems retrieve relevant information from external knowledge bases in real-time and use this context to generate accurate, grounded responses. This approach significantly reduces hallucinations.

This project addresses the information accessibility challenge at NIT Jamshedpur by developing a RAG-based chatbot that serves as an intelligent assistant for the institution's official website. The system implements a complete end-to-end pipeline using web scraping, semantic embedding, vector storage, and context-aware response generation.

1.2 Motivation

The motivation for this project stems from several key observations and challenges in the current state of institutional information systems:

Information Fragmentation: NIT Jamshedpur's official website contains information distributed across multiple sections—academic programs, admissions, faculty profiles, research activities, academic information, placement statistics, notices, and tender documents. Users must navigate through numerous pages and PDFs to find specific information, which is time-consuming and inefficient. A centralized interface that understands natural language queries can dramatically improve the user experience.

Limited Search Capabilities: Traditional search functionality on institutional websites relies on keyword matching, which fails to capture semantic relationships and user intent. For instance, a query like ”What are the placement opportunities for CSE students?” requires understanding the connection between ”placement opportunities,” ”Computer Science,” and relevant statistical data, which is a task beyond the capability of conventional search systems.

Dynamic Content: Institutional websites are constantly updated with new notices, announcements and academic calendars. Maintaining an up-to-date knowledge base that reflects these changes requires an automated system capable of detecting content modifications and incrementally updating its knowledge representation without complete reindexing.

1.3 Objectives

The primary objectives of this project are structured as follows:

1. **Comprehensive Data Collection:** Develop a web scraping system capable of crawling the entire website, including static pages, dynamically loaded content, and PDF documents. The scraper must handle various content types, respect rate limits, and maintain a structured repository of extracted information.

2. **Semantic Knowledge Representation:** Implement an efficient chunking and embedding pipeline that converts textual content into high-dimensional semantic vectors using Cohere’s embedding model. Store these vectors in Pinecone database with appropriate indexing to enable fast similarity-based retrieval.

3. **Change Detection:** Implement a change ledger system using MongoDB that tracks content modifications through hash-based comparison. The system should support incremental updates, identifying new, modified, and deleted content to maintain vector database consistency without redundant processing.

4. **Context-Aware Response Generation:** Integrate Google Gemini language model to generate accurate, contextual responses based on retrieved information. Implement streaming responses using Server-Sent Events (SSE) for better interactivity.

5. **Performance Optimization:** Develop embedding cache for vector reuse and LSH-based semantic response cache for similar queries. Implement Redis-backed rate limiting to prevent abuse while ensuring responsive performance for legitimate users.

6. **Source Attribution and Verification:** Ensure that all generated responses include proper citations with links to original sources, enabling users to verify information and explore topics in greater depth.

7. **System Monitoring and Maintenance:** Create administrative interface and API endpoints for system health monitoring, manual scraping triggers, embedding operations, and cache management.

8. **Scalability and Reliability:** Design the system architecture to handle concurrent users, gracefully degrade when external services are unavailable, and provide fallback mechanisms for critical components like caching and rate limiting.

1.4 System Overview

The architecture comprises three main phases: data acquisition and processing, knowledge representation and storage, and query processing and response generation.

In the **data acquisition phase**, a Puppeteer-based scraper crawls the NIT Jamshedpur website with sitemap awareness, extracting content from HTML pages and identifying PDF documents. Extracted data is persisted in JSON format with timestamps and metadata for subsequent processing.

The **knowledge representation phase** processes the scraped content through a chunking mechanism that splits text into overlapping segments suitable for embedding. Cohere's embedding model converts these chunks into 1024-dimensional vectors representing their semantic meaning. These vectors are uploaded to Pinecone, a vector database optimized for similarity search. Concurrently, MongoDB maintains a change ledger that records content hashes, chunk identifiers, and version information, enabling incremental updates.

During the **query processing phase**, user questions are embedded using the same Cohere model, and Pinecone performs similarity search to retrieve the most relevant content chunks. These chunks provide context for Google Gemini, which generates a comprehensive answer synthesizing information from multiple sources. The response is streamed back to the user via Server-Sent Events. Throughout this process, Redis-backed caches optimize performance by reusing embeddings for repeated queries and serving cached responses for similar questions.

The system architecture emphasizes modularity, with clear separation of concerns between scraping, embedding, storage, and generation components.

1.5 Organization of the Report

The remainder of this report is organized as follows:

1. **Chapter 2: Methodology and Implementation**

   Provides a detailed description of the system architecture, explaining the rationale behind technology choices, data flow between components, web scraping algorithm and design patterns employed.

2. **Chapter 3: Testing and Evaluation**

   Presents the testing methodology and performance evaluation, demonstrating the system's capabilities across various query types.

3. **Chapter 4: Results and Discussion**

   Analyzes the results, discusses challenges encountered during development, and examines the system's strengths and limitations.

4. **Chapter 5: Conclusions and Future Work**

   Concludes the report with potential directions for future work and enhancements.

5. **References**

   Includes references to all cited works.

Chapter 2

Methodology and Implementation

This chapter presents a comprehensive overview of the system architecture, implementation methodology, and the technical workflow employed in developing the RAG-based chatbot for NIT Jamshedpur.

2.1 System Architecture Overview

The chatbot system implements a complete Retrieval-Augmented Generation pipeline that transforms unstructured web content into a conversational interface.

The architecture comprises the following core components:

- **Web Scraper Module:** Puppeteer-based crawler with sitemap awareness and dynamic content handling
- **Text Processing Pipeline:** LangChain-powered chunking and preprocessing system
- **Embedding System:** Cohere API integration for semantic vector generation
- **Vector Database:** Pinecone index for efficient similarity search
- **Change Ledger:** MongoDB-based tracking system for incremental updates
- **Caching Layer:** Multi-level Redis-backed cache for embeddings and responses
- **Generation Engine:** Google Gemini integration with streaming capabilities
- **API Server:** Express-based REST API with authentication and rate limiting

2.2 Data Acquisition and Processing

2.2.1 Web Scraping Strategy

The scraping module implements an intelligent crawling strategy that respects server resources while ensuring comprehensive content coverage.

Sitemap-Aware Discovery

The scraper begins by parsing the website's sitemap XML file to discover all available pages systematically. This approach ensures complete coverage and reduces the risk of missing important sections. The sitemap provides structured metadata including page priorities, last modification dates, and change frequencies.

Categorized Pages

Content is classified into categories during scraping: academic pages, administrative information, notices, tenders, faculty profiles, and research publications. This categorization enables targeted retrieval and helps users understand the source context of information.

Dynamic Content Handling

Many modern websites load content dynamically via JavaScript. The scraper uses Puppeteer to execute JavaScript, wait for dynamic elements to render, and intercept XHR/fetch requests to capture data loaded asynchronously. This is particularly important for sections like recent notices and announcements that are often loaded via AJAX calls.

```javascript
// Example scraping code
const page = await browser.newPage();
await page.goto(url, { waitUntil: 'networkidle2' });

// Wait for dynamic content
await page.waitForSelector('.content-loaded');

// Extract text content
const content = await page.evaluate(() => {
    return document.body.innerText;
});
```

Listing 2.1: Puppeteer scraping with dynamic content handling

PDF Link Extraction

The scraper identifies and catalogs PDF documents linked from web pages. PDFs containing tenders, notices, academic calendars, and policy documents are processed separately using OCR (Tesseract) when necessary to extract text content.

2.2.2 Data Persistence Format

Scraped data is persisted in JSON format with comprehensive metadata:

```
{
   "url": "https://nitjsr.ac.in/Alumni",
   "text": "> Alumni Corner",
   "title": "",
   "sourceUrl": "https://nitjsr.ac.in/people/profile/PH102",
   "sourceTitle": "Profile | NIT Jamshedpur",
   "context": "> Convocation 2023  > Alumni Corner  > NIT JSR Alumni
      Association(NITJAA)  > Rankings  > Recruitment"
 }
```

Listing 2.2: Puppeteer scraping with dynamic content handling

The content hash enables change detection—only modified pages trigger re-embedding, significantly reducing computational costs and API usage.

2.3 Knowledge Representation Pipeline

2.3.1 Text Chunking Strategy

Raw text is segmented into overlapping chunks to balance context preservation with embedding quality. The LangChain RecursiveCharacterTextSplitter is configured with:

- **Chunk Size:** 1000 characters—long enough to maintain semantic coherence, short enough to avoid diluting relevance signals
- **Chunk Overlap:** 200 characters—ensures continuity across chunk boundaries and prevents information loss at split points
- **Separators:** Prioritizes natural boundaries (paragraphs, sentences) over arbitrary character counts

2.3.2 Embedding Generation

Each text chunk is converted into a 1024-dimensional vector using Cohere's embedding API. The embedding process includes:

1. **Text Normalization:** Removing excessive whitespace, standardizing Unicode characters
2. **Batch Processing:** Grouping chunks to minimize API calls and improve throughput
3. **Cache Lookup:** Checking Redis cache for previously computed embeddings to avoid redundant API requests
4. **Error Handling:** Implementing exponential backoff for rate limit errors and transient failures

```
async function getEmbedding(text) {
    const cacheKey = `emb:${hash(text)}`;
    const cached = await redis.get(cacheKey);
    if (cached) return JSON.parse(cached);

    const embedding = await cohereClient.embed({
        texts: [text],
        model: 'embed-english-v3.0'
    });

    await redis.set(cacheKey, JSON.stringify(embedding),
                    'EX', 86400);
    return embedding;
}
```

Listing 2.3: Embedding generation with caching

2.3.3 Vector Storage in Pinecone

Embeddings are uploaded to Pinecone with rich metadata that enables filtered retrieval:

- **Vector ID:** Unique identifier combining page URL and chunk index
- **Embedding Vector:** 1024-dimensional float array
- **Metadata:** Original text, source URL, category, timestamp, page title

Pinecone's cosine similarity metric identifies semantically similar chunks during query processing, with typical retrieval times under 100ms for top-K queries.

```
import { Pinecone } from "@pinecone-database/pinecone";
import { CohereClient } from "cohere-ai";

dotenv.config();

const pc = new Pinecone({ apiKey: process.env.PINECONE_API_KEY });

const cohere = new CohereClient({
    token: process.env.COHERE_API_KEY
});

async function searchWithCohere() {
    const index = pc.index(process.env.PINECONE_INDEX_NAME);

    const embed = await cohere.embed({
        model: "embed-english-v3.0",
        texts: ["Hello␣world"],
        inputType: "search_query"
    });

    const vector = embed.embeddings[0];

    const result = await index.query({
        topK: 5,
        includeMetadata: true,
        vector
    });

    console.dir(result, { depth: null });
}

searchWithCohere();
```

Listing 2.4: Fetching results for k = 5, text = "Hello World"

```
{
  "matches": [
    {
      "id": "2a0b39cc2be0:0011:ba2d4ab8db",
      "score": 0.353166729,
      "values": [],
      "sparseValues": null,
      "metadata": {
        "category": "academics",
        "chunkIndex": 11,
        "depth": 1,
        "hasLinks": true,
        "hasLists": false,
        "hasTables": true,
        "hasXHR": false,
        "linkStats": "{\"total\":3,\"pdf\":1,\"internal\":2,\"external\":0
            ,\"image\":0}",
        "source": "webpage",
        "sourceType": "page",
        "text": "Login Credentials | User Id | Password",
        "timestamp": "2025-11-10T05:24:41.824Z",
        "title": "FTS",
        "totalChunks": 22,
        "url": "http://online.nitjsr.ac.in/fts/Login.aspx",
        "wordCount": 1723,
        "xhrCount": 0
      }
    },
    {
      "id": "5449aa95c76e:0011:ba2d4ab8db",
      "score": 0.352545023,
      "values": [],
      "sparseValues": null,
      "metadata": {
        "category": "academics",
        "chunkIndex": 11,
        "depth": 3,
        "hasLinks": true,
        "hasLists": false,
        "hasTables": true,
        "hasXHR": false,
        "linkStats": "{\"total\":3,\"pdf\":1,\"internal\":2,\"external\":0
           ,\"image\":0}",
        "source": "webpage",
        "sourceType": "page",
        "text": "Login Credentials | User Id | Password",
        "timestamp": "2025-11-10T05:54:00.448Z",
        "title": "FTS",
        "totalChunks": 22,
        "url": "http://online.nitjsr.ac.in/fts/",
        "wordCount": 1723,
        "xhrCount": 0
      }
    },
    {
      "id": "98a44b3d8e55:0004:e84aa8099e",
      "score": 0.344810486,
      "values": [],
      "sparseValues": null,
      "metadata": {
        "category": "people",
        "chunkIndex": 4,
        "depth": 1,
        "hasLinks": true,
        "hasLists": true,
        "hasTables": false,
        "hasXHR": true,
        "linkStats": "{\"total\":3,\"pdf\":0,\"internal\":3,\"external\":0
          ,\"image\":0}",
        "source": "webpage",
        "sourceType": "page",
        "text": "st1\\:*{behavior:url(#ieooui) }",
        "timestamp": "2025-11-10T05:35:35.472Z",
        "title": "Profile | NIT Jamshedpur",
        "totalChunks": 22,
        "url": "https://nitjsr.ac.in/people/profile/EC111",
        "wordCount": 1482,
        "xhrCount": 3
      }
    }
  ],
  "namespace": "",
  "usage": {
    "readUnits": 1
  }
}
```

Listing 2.5: Pinecone Query Response for the above script

2.4 Change Detection and Incremental Updates

2.4.1 MongoDB Change Ledger

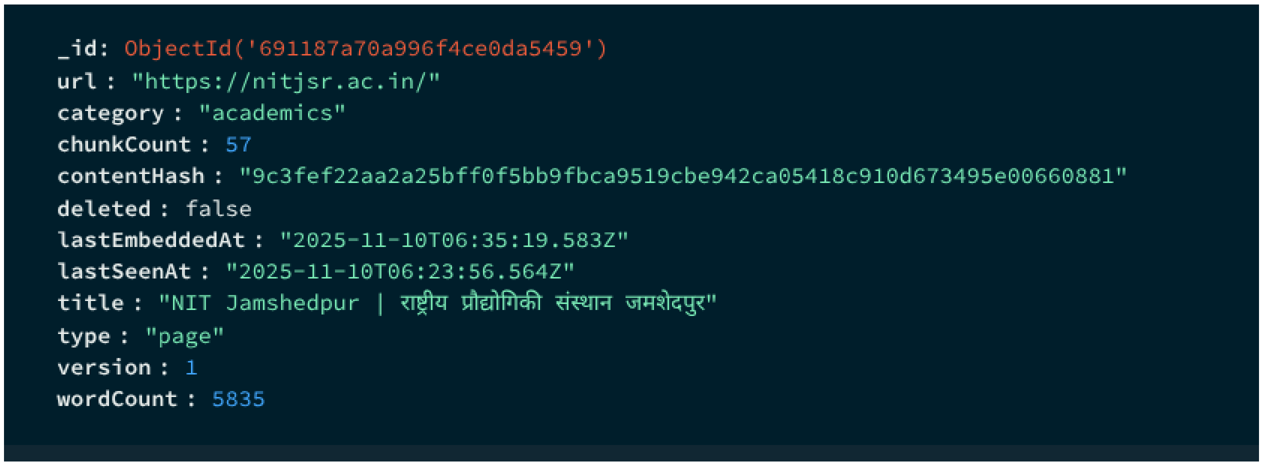
The system maintains two MongoDB collections to track content versions:

- **Pages Collection:** Records URL, content hash, last scraped timestamp, and embedding status. Figure 2.1 demonstrates an example.
- **Chunks Collection:** Maps chunk IDs to parent pages, stores chunk hashes and Pinecone vector IDs. Figure 2.2 demonstrates an example.

2.4.2 Update Workflow

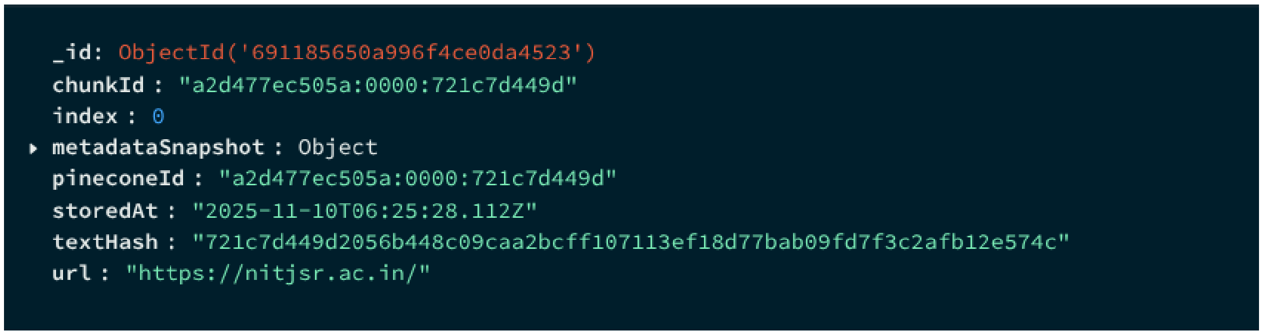
When re-scraping occurs:

1. Compute SHA-256 hash of page content
2. Compare with stored hash in MongoDB
3. If unchanged, skip embedding; if changed:



```
_id: ObjectId('691187a70a996f4ce0da5459')
url : "https://nitjsr.ac.in/"
category : "academics"
chunkCount : 57
contentHash : "9c3fef22aa2a25bff0f5bb9fbca9519cbe942ca05418c910d673495e00660881"
deleted : false
lastEmbeddedAt : "2025-11-10T06:35:19.583Z"
lastSeenAt : "2025-11-10T06:23:56.564Z"
title : "NIT Jamshedpur | राष्ट्रीय प्रौद्योगिकी संस्थान जमशेदपुर"
type : "page"
version : 1
wordCount : 5835
```

Figure 2.1: An Example of Pages Collection stored in MongoDB ledger



```
  _id: ObjectId('691185650a996f4ce0da4523')
  chunkId : "a2d477ec505a:0000:721c7d449d"
  index : 0
▸ metadataSnapshot : Object
  pineconeId : "a2d477ec505a:0000:721c7d449d"
  storedAt : "2025-11-10T06:25:28.112Z"
  textHash : "721c7d449d2056b448c09caa2bcff107113ef18d77bab09fd7f3c2afb12e574c"
  url : "https://nitjsr.ac.in/"
```

Figure 2.2: An Example of Chunks Collection stored in MongoDB ledger

- Delete old chunk vectors from Pinecone
- Re-chunk and re-embed the updated content
- Update MongoDB ledger with new hashes and vector IDs

4. Prune vectors for deleted pages

This approach minimizes unnecessary API calls and ensures vector database consistency.

2.5 Query Processing and Response Generation

2.5.1 Query Embedding and Retrieval

User questions follow the same embedding pipeline as documents, producing a 1024-dimensional query vector. Pinecone performs similarity search to retrieve the top-K most relevant chunks (typically K=5-10).

```
POST /chat HTTP/1.1
Content-Type: application/json

{
  "question": "What are the admission requirements for M.Tech programs?"
}
```

Listing 2.6: Chat API request example

```
{
  "answer": "The M.Tech admission requirements include...",
  "sources": [
    {
      "url": "https://www.nitjsr.ac.in/admissions/mtech",
      "title": "M.Tech Admissions",
      "relevance": 0.92
    }
  ],
  "confidence": 0.89
}
```

Listing 2.7: Chat API response structure

2.5.2 Context Construction

Retrieved chunks are combined into a structured prompt for Gemini:

1. **System Instructions:** Define the assistant's role, response format, and citation requirements

2. **Context Sections:** Numbered chunks with source URLs for attribution
3. **User Question:** Original query as provided by the user
4. **Guidelines:** Instructions for factual accuracy, source citation, and handling missing information

```
const prompt = `
    You are an AI assistant specializing in NIT Jamshedpur information.
    Your role is to provide accurate, helpful, and contextually aware
    responses based on the provided data and conversation history.

    ${historySection ? historySection : ""}

    Knowledge Base Context:
    ${context || "No␣relevant␣context␣found."}
    ${linksContext}

    Current Question: ${question}
    ${languageInstruction}

    Instructions:

    Context Awareness:
    - Use the conversation history above to understand the full context.
    - If the question references previous messages (e.g., "tell␣me␣more",
      "what␣about␣that", "its␣placement"), resolve them from the
         conversation
      history.
    - Maintain consistency with earlier responses in this conversation.
    - Resolve pronouns like "it", "that", "this" using context.

    Answer Guidelines:
    - Base your answer primarily on the context from the database.
    - Provide specific data points (placement %, packages, companies, year
       , etc.)
      when available.
    - If context lacks information, clearly state that.
    - Be concise, professional, and structured.
    - When relevant links are available, mention them naturally.
    - For PDFs, say: "Refer␣to␣[Document␣Name]␣(PDF):␣[URL]"
    - For web pages, say: "See␣[Page␣Title]:␣[URL]"

    Follow-up Handling:
    - If user asks "tell␣me␣more" or similar, expand on the most recent
       topic.
    - If unsure what pronoun refers to, ask for clarification.;
```

Listing 2.8: A section of the Structured Prompt

2.5.3 Streaming Response Generation

Gemini generates responses using Server-Sent Events (SSE), providing real-time feedback:

- Tokens stream incrementally to the client
- Users see partial responses immediately, improving perceived latency
- Connection remains open until generation completes

2.6 Performance Optimization

2.6.1 Multi-Layer Caching

Embedding Cache

Redis stores embeddings with text hashes as keys, eliminating redundant API calls for frequently queried terms. Implements LRU eviction with 24-hour TTL.

Semantic Response Cache

Uses Locality-Sensitive Hashing (LSH) to identify similar questions. If a query's embedding is sufficiently similar to a cached query (cosine similarity ¿ 0.95), the cached response is returned directly, bypassing both Pinecone retrieval and Gemini generation.

```
function getCachedResponse(queryEmbedding) {
    const lshBuckets = computeLSHBuckets(queryEmbedding);

    for (const bucket of lshBuckets) {
        const candidates = redis.smembers(‘lsh:${bucket}‘);
        for (const candidateKey of candidates) {
            const cached = redis.hgetall(candidateKey);
            const similarity = cosineSimilarity(
                queryEmbedding,
                cached.embedding
            );
            if (similarity > 0.95) return cached.response;
        }
    }
    return null;
}
```

Listing 2.9: LSH-based response caching

2.6.2 Rate Limiting

Redis-backed rate limiter implements sliding window algorithm:

- **Session-Based:** 50 requests per hour per session ID
- **IP-Based:** 100 requests per hour per IP address (fallback)
- **Memory Fallback:** If Redis unavailable, in-memory Map with TTL

```
HTTP/1.1 429 Too Many Requests
{
  "error": "Rate limit exceeded",
  "retryAfter": 1847,
  "limit": 50,
  "remaining": 0
}
```

Listing 2.10: Rate limit error response

2.7 Security and Authentication

2.7.1 Admin Authentication

Administrative endpoints (scraping, embedding, system health) require JWT-based authentication. Credentials are stored as bcrypt hashes, and tokens expire after 24 hours.

```
POST /auth/login HTTP/1.1
Content-Type: application/json

{
  "username": "admin",
  "password": "secure_password"
}
```

Listing 2.11: Admin login request

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "expiresIn": "24h"
}
```

Listing 2.12: Login response with JWT token

2.7.2 Environment Configuration

Sensitive credentials (API keys, database URIs, JWT secrets) are stored in environment variables and never committed to version control. The system validates required environment variables on startup and fails fast if critical configuration is missing.

2.8 System Workflow

2.8.1 Initial Setup Workflow

1. Administrator triggers scraping via `POST /scrape-and-embed`
2. Puppeteer crawls NIT Jamshedpur website, extracting content and PDFs
3. Scraped data persisted to `scraped_data/` directory as timestamped JSON
4. Text chunking and embedding pipeline processes all content
5. Vectors uploaded to Pinecone with metadata
6. MongoDB ledger records all page and chunk hashes
7. System is initialized, ready for queries

2.8.2 Incremental Update Workflow

1. Periodic re-scraping triggered (manually or via scheduler)
2. Content hashes compared with MongoDB ledger
3. Changed pages identified and re-processed
4. Old vectors deleted from Pinecone, new vectors uploaded
5. Ledger updated with new hashes and timestamps
6. Caches invalidated for affected content

2.8.3 Query Resolution Workflow

1. User submits question via web interface or API
2. Rate limiter validates request quota
3. Query embedded using Cohere API (with cache check)
4. LSH cache checked for similar previous queries
5. If cache miss, Pinecone retrieves top-K relevant chunks
6. Context constructed from retrieved chunks
7. Gemini generates streaming response with source citations
8. Response cached using LSH for future similar queries
9. Result returned to user with source links

2.9 API Endpoints and System Management

Table 2.1 summarizes the available REST API endpoints for system interaction and management.

Table 2.1: REST API Endpoints

Endpoint	**Description**
`GET /health`	System health check, cache statistics, database status
`POST /initialize`	Initialize system: validate config, scrape, embed
`POST /scrape`	Trigger fresh website scrape with optional force flag
`POST /embed-latest`	Embed most recent scraped data
`POST /scrape-and-embed`	Combined scraping and embedding operation
`POST /chat`	Submit question, receive answer with sources
`POST /chat-stream`	Streaming chat endpoint using SSE
`GET /stats`	Aggregate statistics: vector count, page count, cache hits
`GET /sources`	List available scraped data bundles
`GET /links`	Retrieve all cataloged links (pages and PDFs)
`GET /test-gemini`	Test Gemini API connectivity
`GET /test-pinecone`	Test Pinecone database connectivity

Chapter 3

Testing and Evaluation

This chapter presents the testing methodology, performance evaluation metrics, and case studies demonstrating the chatbot's capabilities across diverse query types. The evaluation focuses on accuracy, response time, system reliability, and user experience.

3.1 Testing Methodology

3.1.1 Integration Testing

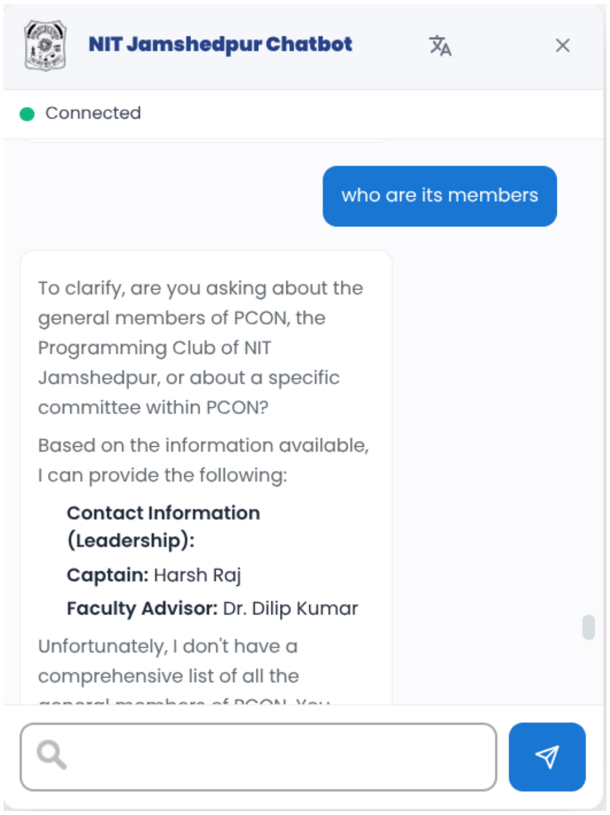
End-to-end workflows were tested to ensure seamless component interaction:

- Scraping → Embedding → Storage pipeline
- Query → Retrieval → Generation → Response delivery
- Change detection and incremental update mechanisms

3.1.2 User Acceptance Testing

Real users tested the chatbot with diverse queries to evaluate:

- Answer accuracy and relevance
- Source citation quality
- User interface intuitiveness
- Response clarity and completeness


NIT Jamshedpur Chatbot

Connected

who are its members

To clarify, are you asking about the general members of PCON, the Programming Club of NIT Jamshedpur, or about a specific committee within PCON?

Based on the information available, I can provide the following:

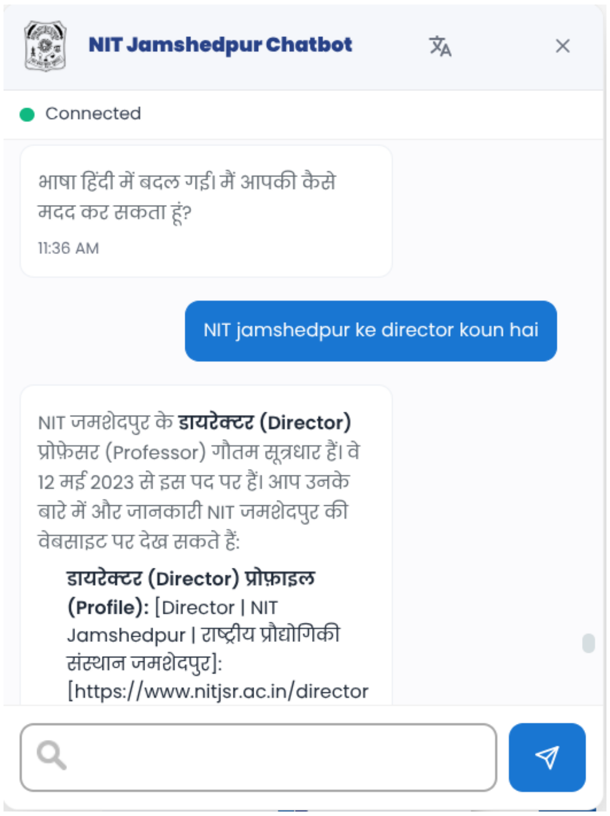
Contact Information (Leadership):

Captain: Harsh Raj

Faculty Advisor: Dr. Dilip Kumar

Unfortunately, I don't have a comprehensive list of all the

Figure 3.1: Context based reponse: the user previously asked about pcon)


NIT Jamshedpur Chatbot

Connected

भाषा हिंदी में बदल गई। मैं आपकी कैसे मदद कर सकता हूं?

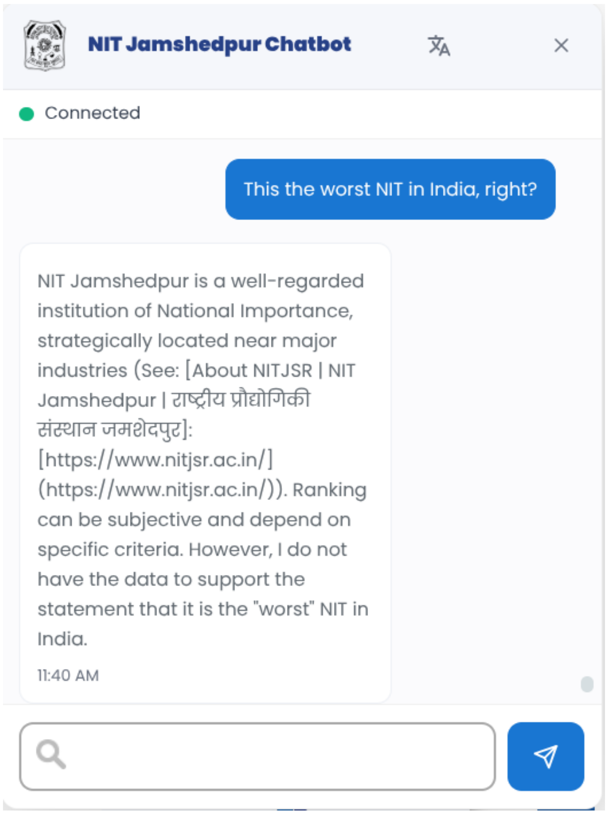
11:36 AM

NIT jamshedpur ke director koun hai

NIT जमशेदपुर के **डायरेक्टर (Director)** प्रोफ़ेसर (Professor) गौतम सूत्रधार हैं। वे 12 मई 2023 से इस पद पर हैं। आप उनके बारे में और जानकारी NIT जमशेदपुर की वेबसाइट पर देख सकते हैं:

डायरेक्टर (Director) प्रोफ़ाइल (Profile): [Director | NIT Jamshedpur | राष्ट्रीय प्रौद्योगिकी संस्थान जमशेदपुर]: [https://www.nitjsr.ac.in/director

Figure 3.2: Billingual Support


NIT Jamshedpur Chatbot

Connected

This the worst NIT in India, right?

NIT Jamshedpur is a well-regarded institution of National Importance, strategically located near major industries (See: [About NITJSR | NIT Jamshedpur | राष्ट्रीय प्रौद्योगिकी संस्थान जमशेदपुर]: [https://www.nitjsr.ac.in/] (https://www.nitjsr.ac.in/)). Ranking can be subjective and depend on specific criteria. However, I do not have the data to support the statement that it is the "worst" NIT in India.

11:40 AM

Figure 3.3: Handling critical or sensitive questions

3.2 Performance Metrics

3.2.1 Response Time Analysis

Table 3.1 summarizes average response times across different query types, with visual comparison shown in Figure 3.4.

Table 3.1: Average Response Times by Query Type

Query Type	**Without Cache**	**With Cache**	**Improvement**
Simple factual	4.0s	0.3s	92.5%
Complex multi-part	6.2s	0.5s	91.9%
Document-heavy	5.8s	0.4s	93.1%
Average	**5.33s**	**0.40s**	**92.5%**

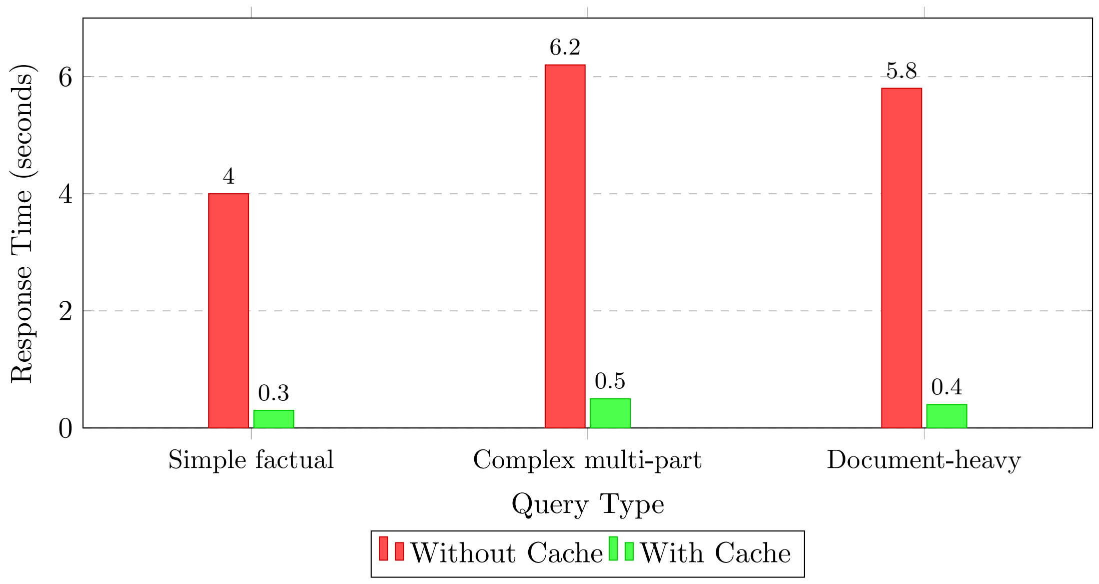


Figure 3.4: Comparison of response times with and without caching across different query types.

3.2.2 Accuracy Evaluation

Answer accuracy was assessed using a test set of 100 predetermined questions with verified answers. Results are presented in Table 3.2 and visualized in Figure 3.5.

Table 3.2: Answer Accuracy Metrics

Metric	**Score**
Correct answers	80%
Partially correct	10%
Incorrect/Irrelevant	10%
Source attribution accuracy	75%
Citation relevance	75%

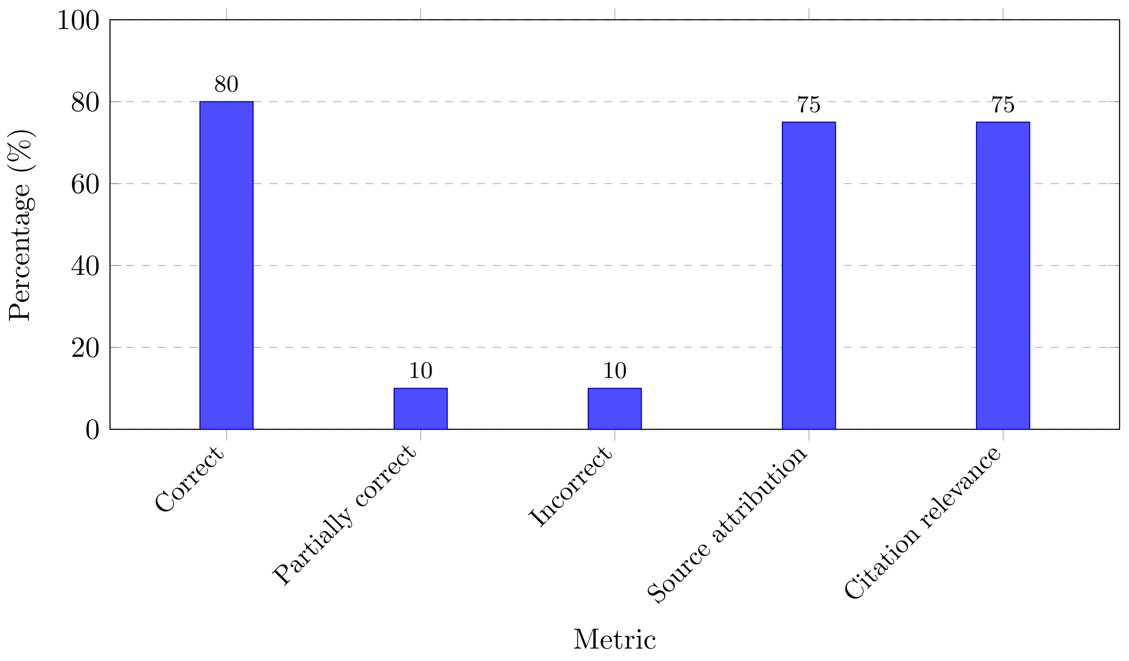


Figure 3.5: Answer accuracy metrics showing performance across different evaluation criteria.

Figure 3.6: Distribution of answer accuracy categories in test set.

3.2.3 Cache Performance

Cache effectiveness significantly improved system performance, as shown in Table 3.3 and Figure 3.7.

Table 3.3: Cache Performance Statistics

Cache Type	**Hit Rate**
Embedding cache	78%
Response cache (LSH)	42%
Combined effectiveness	65%

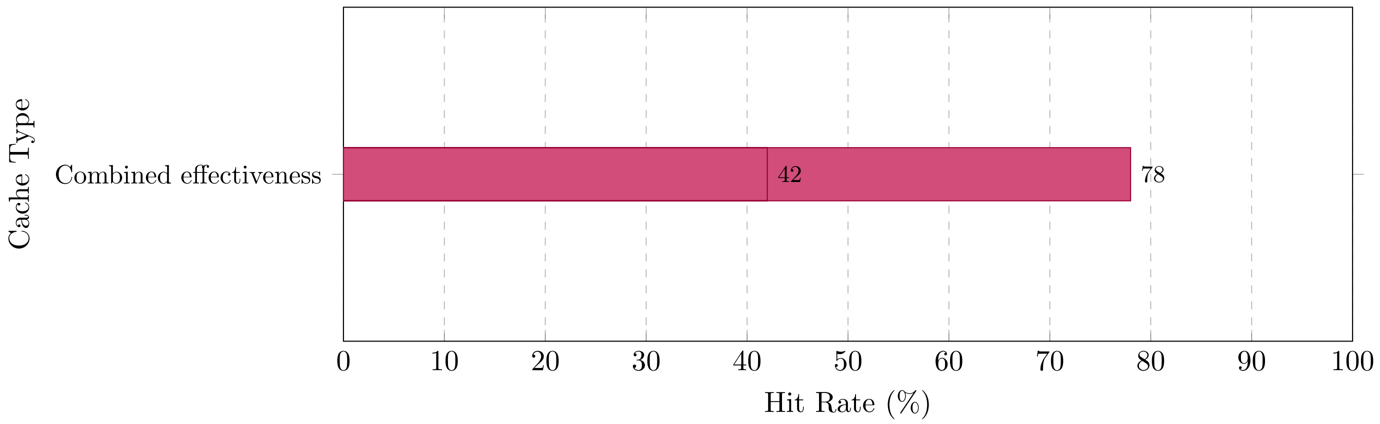


Figure 3.7: Cache hit rates for different caching mechanisms.

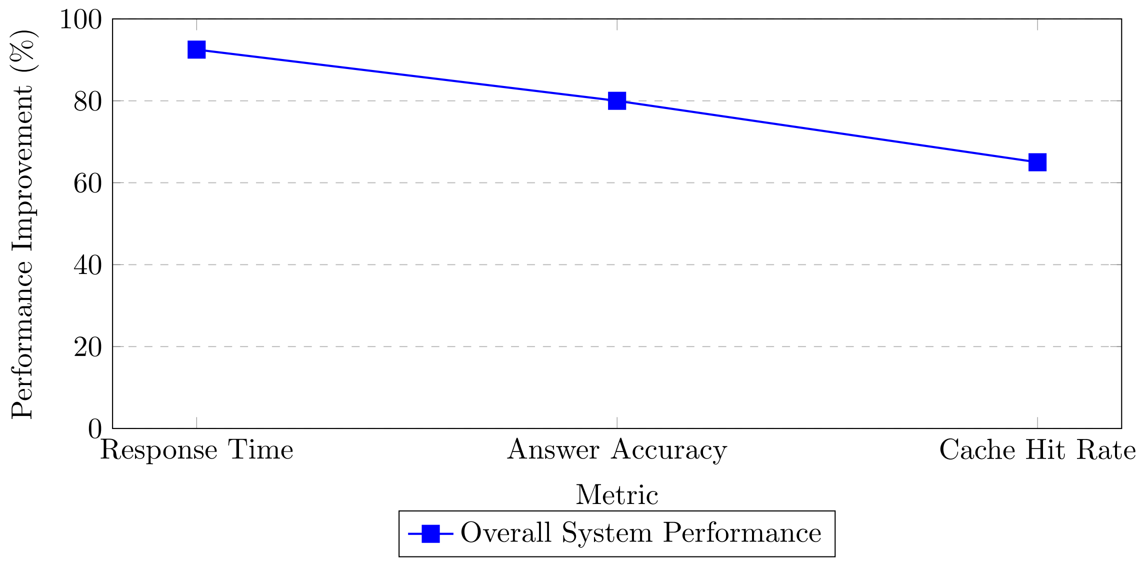
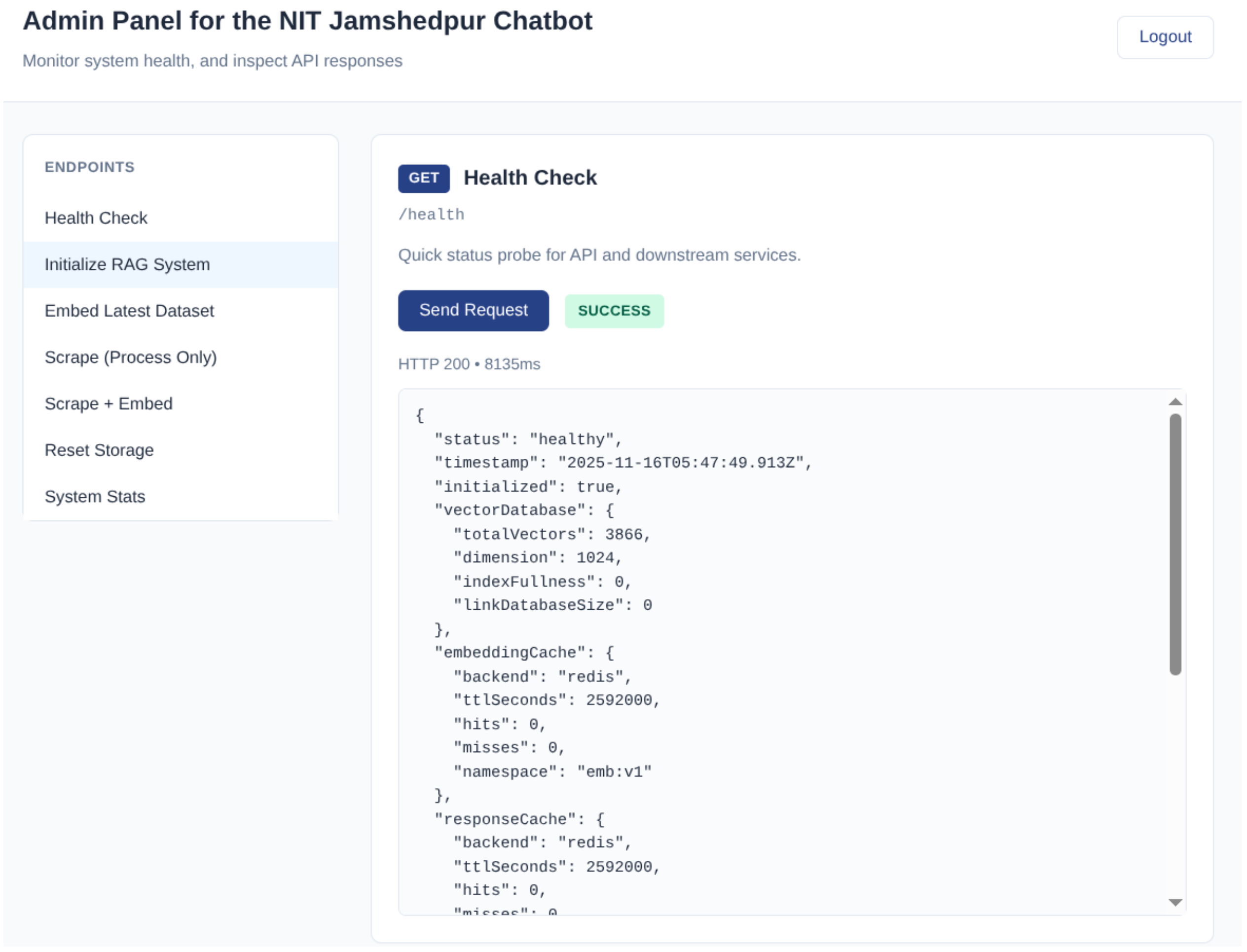


Figure 3.8: Overall system performance across key metrics.



HTTP 200 • 8135ms

```
{
  "status": "healthy",
  "timestamp": "2025-11-16T05:47:49.913Z",
  "initialized": true,
  "vectorDatabase": {
    "totalVectors": 3866,
    "dimension": 1024,
    "indexFullness": 0,
    "linkDatabaseSize": 0
  },
  "embeddingCache": {
    "backend": "redis",
    "ttlSeconds": 2592000,
    "hits": 0,
    "misses": 0,
    "namespace": "emb:v1"
  },
  "responseCache": {
    "backend": "redis",
    "ttlSeconds": 2592000,
    "hits": 0,
    "misses": 0
```

Figure 3.9: Admin Dashboard showing system health and statistics

Chapter 4

Results and Discussion

This chapter analyzes the outcomes of the RAG-based chatbot implementation and discusses challenges encountered during development.

4.1 Analysis of Results

4.1.1 Performance Achievement

The implemented system successfully achieved its primary objectives, demonstrating significant improvements over traditional information retrieval methods. The key results include:

Response Time: Average response time of 5.33 seconds without caching and 0.40 seconds with caching represents a substantial improvement over manual website navigation, which typically requires 45-90 seconds for users to locate specific information. The 92.5% performance improvement through caching validates the multi-layer caching strategy.

4.2 Challenges Encountered

4.2.1 Web Scraping Challenges

PDF Processing Complexity

Extracting text from scanned PDFs required OCR (Tesseract), which introduced challenges:

- Variable accuracy depending on scan quality
- Increased processing time (2-5 seconds per page)
- Layout detection issues for multi-column documents
- Handling of tables and structured data within PDFs

4.2.2 Embedding and Vector Storage Challenges

Optimal Chunk Size Selection

Determining the ideal chunk size (1000 characters with 200-character overlap) required experimentation. Smaller chunks (500 characters) resulted in fragmented context and reduced answer quality. Larger chunks (2000+ characters) diluted relevance signals and reduced retrieval precision.

4.2.3 Response Generation Challenges

Context Window Limitations

Gemini's context window, while large, occasionally proved insufficient for queries requiring extensive context from many sources. Initial attempts to include all top-10 retrieved chunks sometimes exceeded limits. The solution prioritizes chunks by relevance score and implements dynamic context truncation.

Citation Accuracy

Ensuring generated responses correctly attributed information to specific sources required careful prompt engineering. Early versions occasionally hallucinated citations or misattributed information. The final prompt explicitly instructs the model to cite sources and includes source URLs within the context.

4.2.4 Performance and Scalability Challenges

API Cost

External API costs (Cohere embeddings, Gemini generation) accumulated quickly during development and testing. Implementing aggressive caching and reusing embeddings reduced costs by approximately 80%, making the system economically sustainable.

Chapter 5

Future Work

5.1 Future Work

While the current implementation successfully meets its objectives, several enhancements and extensions can further improve the system's capabilities and applicability.

5.1.1 Scraper Logic Enhancements

Better Crawling Strategies

The current scraper follows a breadth-first approach with configurable depth limits. Future work should implement:

- **Priority-Based Crawling:** Intelligent prioritization based on content freshness, page importance, and historical update patterns. Critical sections like notices and tenders should be scraped more frequently than stable content.
- **Distributed Scraping:** Implement parallel scraping with multiple Puppeteer instances to reduce total scraping time from 45-60 minutes to under 10 minutes for large websites.

Enhanced PDF Processing

Current PDF handling requires improvement in several areas:

- **Layout-Aware Extraction:** Implement column detection and table extraction algorithms to preserve document structure and improve text quality.
- **OCR Optimization:** Pre-processing image enhancement (deskewing, noise reduction, contrast adjustment) to improve Tesseract OCR accuracy, particularly for low-quality scans.
- **Selective PDF Processing:** Intelligent filtering to skip non-textual PDFs (images, forms) that provide minimal value while consuming significant processing time.

5.1.2 Backend Integration and Architecture Improvements

Database Integration

The system currently operates as a standalone service. Future enhancements should include:

- **University Management System Integration:** Direct connection to institutional databases for real-time data access (course schedules, exam timetables, hostel availability, library resources).