Executive Summary

Our project, sponsored by Nexteer Automotive, is undertaken by the AI Bots team — a group of six students from Carnegie Mellon University's MISM and MISM-BIDA programs. We share a common goal of gaining hands-on experience with AI while delivering solutions that align with our sponsor's objectives.

The challenge we are addressing is that Nexteer's internal organization currently relies on multiple individual chatbots, each designed for a specific domain. However, there is a lack of a centralized system to efficiently direct user queries to the appropriate chatbot. To address this, we designed and developed a centralized chatbot dispatcher capable of efficiently routing user queries to the appropriate chatbot based on domain expertise. Our solution uses advanced redirection logic, including a metadata structure and similarity search mechanisms, to accurately identify the nature of the query and direct it to the most suitable agent. This approach streamlines the query resolution process, reduces response times, and enhances overall user satisfaction by ensuring that employees receive timely and accurate assistance.

Project Goals and Objectives

Nexteer Automotive, headquartered in Auburn Hills, Michigan, operates globally, with locations in countries like Poland and China. As a leader in motion control technology, Nexteer specializes in automotive steering and driveline systems. Their mission is to accelerate mobility that is safe, green, and exciting through innovative motion control solutions.

Following our initial discussions with Nexteer, we identified several pain points within their internal chatbot system:

- **Domain Identification Issues**: Current chatbots struggle to accurately determine the domain and skill required for a user’s question, leading to misdirected queries
- **Inefficient Query Process**: The current system is inefficient, reducing user satisfaction
- **Inaccurate Responses**: Employees cannot get the correct answers to their questions, hindering their ability to resolve issues promptly and impacting their workflow

To address these challenges, our primary objective is to develop a **chatbot dispatcher** that streamlines the process of routing queries to the most appropriate chatbot. The dispatcher will incorporate a metadata structure to identify the domain of user queries and determine the best-suited chatbot based on skill and access level.

Our solution will integrate seamlessly with Nexteer’s existing chatbots, enhancing their functionality rather than replacing them. Once implemented, the dispatcher will ensure that users receive accurate answers within the same conversation, improving efficiency and user experience.

Value and Business Impact

Our project delivers a Proof of Concept that provides Nexteer with a robust redirection logic system for their chatbot framework. Alongside this deliverable, we offer comprehensive research and competitor analysis that demonstrates why our chosen methods are the most effective for Nexteer's needs.

By developing and validating different approaches to chatbot redirection logic, we help Nexteer save both time and financial resources that would otherwise be spent on hiring personnel to conduct extensive research. Our solution streamlines the chatbot experience by consolidating multiple functions into a unified dispatcher, thereby reducing the time users spend retrieving knowledge and information.

Furthermore, our experimentation with various large language models (LLMs) ensures that Nexteer can implement the most suitable model for their specific needs. This not only enhances accuracy and efficiency but also minimizes computational costs, which scale with usage. Ultimately, our work contributes to improved operational efficiency, cost savings, and a more seamless user experience.

Data Architecture

The data architecture underpinning our technical solution is designed to model and analyze agents within an automated system. At the heart of this architecture is the Agent, the core entity that represents the various functionalities and capabilities of the automated systems within the framework.

Within the Agent's data model, we focus on three key fields of abstraction: **Description**, **Capabilities**, and **Specialized Keywords**. These fields are used to encapsulate the essential characteristics of each agent, providing a structured approach to understanding its role and expertise.

1. **Description**:
   This field offers a high-level summary of what the Agent is designed to do. It defines the core function of the agent, such as natural language processing, task automation, data visualization, or more abstract features like emotional intelligence or creative problem-solving. The Description allows for an easy understanding of the Agent's intended purpose in the system.
2. **Capabilities**:
   This field captures the agent's core functional abilities. It specifies what the Agent can accomplish, such as "Providing personalized product recommendations" or "Automating routine business processes". These capabilities give insight into how the Agent contributes to the overall system and its usefulness in various operational scenarios.
3. **Specialized Keywords**:
   The Specialized Keywords field includes a set of domain-specific terms or knowledge areas in which the Agent possesses expertise. These could range from programming languages and data analysis techniques to customer service best practices. By capturing this information, we provide a more nuanced understanding of the Agent's specialization, helping identify the best agent for specific user needs.

Branching out from the Agent, the system includes several other key data entities that interact with and contribute to understanding the Agent's role: **Users**, **Conversations**, **Messages**, and **DispatcherLogs**. These entities help build a complete picture of the Agent's function within the broader system, contributing to efficient user-agent interactions and data management.

Example: Organizational Information Bot

Consider an example of an *Organizational Information Bot*, designed to provide insights into company hierarchy, roles, and operations. This bot could answer questions on organizational structure, such as identifying department managers or understanding department functions.

- **Description**:
    - Retrieve organizational role and hierarchy information
    - Identify departmental heads and management structure
    - Access details on departmental functions and responsibilities
    - Provide contacts for department-specific inquiries
- **Capabilities**:
    - Department hierarchy
    - Organizational roles
    - Department managers
    - Management structure
    - Organizational contacts
    - Department functions
    - Role responsibilities
    - Leadership structure
    - Employee roles
    - Organizational chart
    - Interdepartmental communication
- **Specialized Keywords**:
    - Organizational structure
    - Management hierarchy
    - Departmental roles
    - Contact information

Challenges with the Original Schema

The original system schema posed several challenges due to its limited capture of bot functionalities. It only included basic bot details, making the understanding of the agent's specific capabilities and expertise difficult. This lack of metadata also led to inefficient bot matching, where users were often paired with unsuitable agents, leading to slower resolution times and suboptimal interactions.

Moreover, as the bot system scaled, the absence of structured metadata created significant challenges in maintaining and customizing the agents, often requiring extensive schema modifications. The need for more granular, flexible, and structured data became evident as the system grew.

Database Schema: ai_agent Table

The following SQL statement defines the table structure for storing Agent data, central to our solution:

```
Unset
CREATE TABLE ai_agent (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT DEFAULT NULL,
    capability TEXT DEFAULT NULL,
    description TEXT DEFAULT NULL,
    is_public INTEGER NOT NULL DEFAULT 1,
    online INTEGER NOT NULL,
    agent_type TEXT DEFAULT NULL
);
```

This table captures essential details about each agent, including its name, capabilities, description, status (whether it is public or online), and agent type. The table is designed to provide a flexible structure for efficient data retrieval and scaling as the system grows.

For more information on the database schema and related operations, please refer to the full script here:

- create_table.sql
  https://github.com/Rugz007/capstone/blob/main/backend/database/create_table.sql

Conclusion

The enhanced data architecture offers a robust solution for modeling agents with clear, structured metadata that allows for efficient agent management, better user-agent matching, and scalability for future growth. By improving our schema and metadata structure, we address previous challenges and lay a strong foundation for future development.

Auto Metadata Generation

The **Auto Metadata Generation** component streamlines the creation of structured metadata for agents, ensuring efficient query routing and scalability. It combines text extraction, LLM-based processing, and storage in a vector database. Nexteer can integrate this functionality using the provided API to create metadata for any number of new chatbots or to update the existing metadata of chatbots.

Steps in Auto Metadata Generation

1. **Text Extraction**
    - Extracts text from PDF documents using **PyMuPDF (fitz)**
    - Combines all text to create a unified input for metadata generation
2. **Metadata Processing**
    - Uses **LangChain** with a locally hosted LLM to process text
    - Outputs structured metadata fields:
        - **Description**: Summarizes the agent's purpose
        - **Capabilities**: Lists the agent's functional abilities
        - **Specialized Keywords**: Extracts key domain-specific terms
3. **Metadata Storage**
    - Stores generated metadata in **ChromaDB**, enabling similarity searches.
    - Ensures quick retrieval for query routing
4. **API Integration**
    - Exposes POST /metadata to dynamically process and store metadata for agents
    - Simplifies integration with Nexteer's systems

How Nexteer Can Use This Component

- Dynamically generate metadata for new agents by providing raw text/documents through the API
- Improve query routing by leveraging structured metadata stored in the vector database.
- Scale the system by efficiently onboarding new agents or updating existing metadata

This component is designed for flexibility and scalability, with implementation details available in the scripts:

- metadata.py
  https://github.com/Rugz007/capstone/blob/main/backend/documents/metadata.py#L7
- app.py
  https://github.com/Rugz007/capstone/blob/main/backend/app.py#L31

Similarity Search in Query Routing

Traditional search methods often fail to understand the context and intent behind user queries. Lexical search, while fast, misses semantic relationships, and full semantic search can be computationally expensive. Similarity search offers a balanced approach, providing contextual understanding with reasonable efficiency.

Key motivations for implementing similarity search include:

1. Contextual Matching: Identifies semantically relevant results
2. Efficiency: Faster than full semantic search
3. Scalability: Handles large datasets effectively
4. User-Centric: Adapts to varied user queries

Steps to Implement the Similarity Search System

1. **Embedding Generation**:
   Used an HTTP-based embedding model (HTTPEmbeddingModel) to generate vector representations of queries and documents

Example code snippet:

```python
class HTTPEmbeddingModel(Embeddings):
    def __init__(self, api_url: str = "http://127.0.0.1:1234/v1/embeddings",
model_name: str = "nomic-embed-text-v1.5"):
        self.api_url = api_url
        self.model_name = model_name

    def get_embedding(self, text: str) -> List[float]:
        payload = {
            "model": self.model_name,
            "input": text
        }
        response = requests.post(self.api_url, json=payload,
headers={"Content-Type": "application/json"})
        # ... (error handling and response processing)
        return embedding_data[0].get("embedding", [])
```

2. **Vector Database**:
   Stored document embeddings in a vector database (ChromaDB) for efficient retrieval.
3. **Similarity Calculation**:
   Applied cosine similarity to find the closest match between user queries and agent/document embeddings

4. **Query Routing**:
   Integrated with a triage agent for final query routing based on similarity scores

Component Interaction Architecture

Document Processing Pipeline

The system begins with document ingestion, where various data sources and documents enter the processing pipeline. These documents are processed through our HTTP-based embedding model (HTTPEmbeddingModel), which transforms the textual content into high-dimensional vector representations. Once transformed, these vector embeddings are systematically stored in ChromaDB, our vector database, creating an efficient searchable index of document representations.

Query Processing Flow

Users' queries undergo a similar transformation process when interacting with the system. Each query is passed through the same embedding model to generate vector representations comparable to the stored document vectors. This consistency in processing between documents and queries ensures accurate similarity comparisons and relevant search results.

Database Integration

At the core of our architecture lies ChromaDB, serving as the central vector storage system. It not only stores both document and query embeddings, but also facilitates efficient vector similarity computations. The database implements sophisticated Top-K nearest neighbor search functionality, enabling quick retrieval of the most relevant documents based on vector similarity metrics.

Search and Retrieval Process

The search process is initiated when a query enters the system. The embedding model first generates query vectors, which ChromaDB uses to perform vector similarity calculations. The system then returns the Top-K nearest neighbors based on cosine similarity measurements. These results are subsequently passed to the triage agent, which makes the final routing decisions based on the similarity scores and other relevant factors.

System Communication Protocol

Our architecture implements a robust communication protocol where components interact via HTTP/REST APIs. The embedding model exposes specific endpoints for vector generation, while ChromaDB provides interfaces for vector storage and retrieval operations. The system employs asynchronous processing to handle large-scale document ingestion efficiently, ensuring smooth operation even under heavy loads.

This architecture ensures seamless data flow while maintaining system modularity and scalability, with each component having well-defined responsibilities and interfaces that facilitate easy maintenance and future enhancements.

Advantages and Limitations

Pros:

1. Semantic Understanding: Captures contextual relationships
2. Versatility: Applicable across various domains (text, images, audio)
3. Scalability: Efficiently handles large-scale datasets
4. Personalization: Enables tailored recommendations
5. Dimensionality Handling: Effectively processes high-dimensional data

Cons:

1. Implementation Complexity: Requires significant computational resources
2. Embedding Quality Dependence: Performance relies on vector representations
3. Interpretability Challenges: May lack intuitive explanations for results
4. Sensitivity to Noise: Can be affected by outliers
5. Privacy Concerns: Handling sensitive data in vector form poses risks

Potential Applications for Nexteer

1. **Customer Support**: Route customer queries to the most relevant department or knowledge base
2. **Product Recommendations**: Suggest similar or complementary automotive parts based on user searches
3. **Technical Documentation**: Improve search functionality within internal documentation systems
4. **Quality Control**: Identify similar defect reports or maintenance issues across different product lines

For detailed implementation, use this script:

- embeddings.py https://github.com/Rugz007/capstone/blob/162f86e0632e4ae1ef4494fc9cc6d4c4fdfaa815/backend/documents/embeddings.py#L6

Technical Implementation

Frontend

The front end is a testing ground for a modern chat interface, built with Next.js to leverage server-side rendering and static site generation for fast performance. Real-time response handling ensures seamless interactions.

TypeScript provides type safety for reliable development, while Tailwind CSS simplifies styling with a utility-first approach. Concurrency support ensures the interface handles multiple interactions efficiently, making it a robust platform for testing real-time chat features.

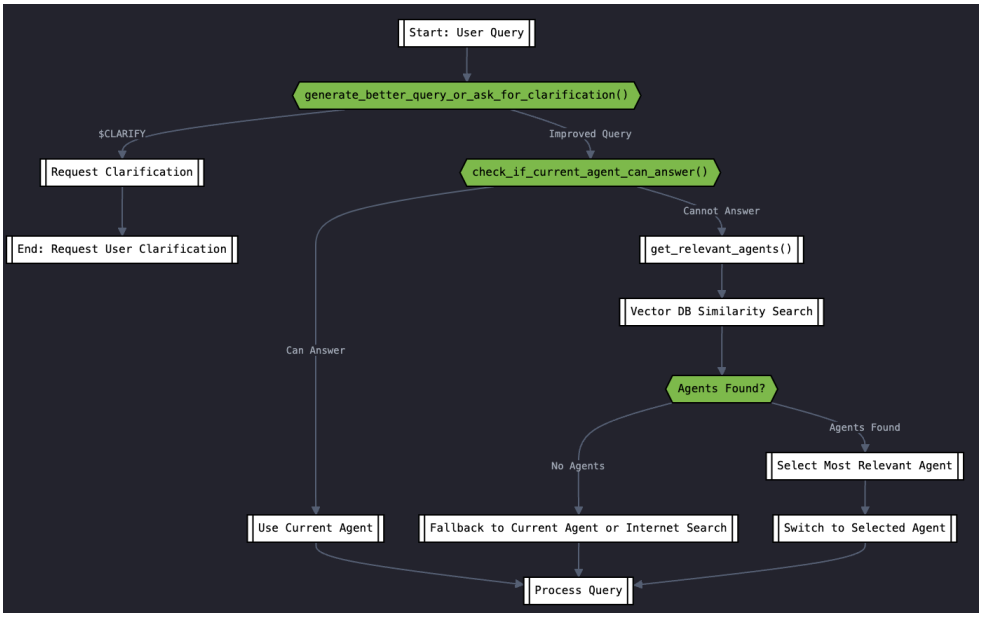
Backend

The backend is built using Python with FastAPI, providing a robust and efficient REST API for seamless client-server communication.

Key components include:

- **LMStudio**: Enables local LLM inference while adhering to OpenAI API standards for compatibility. (This can be switched to use an OpenAI API anytime - uses the same standard)
- **Modular Design**: Ensures flexibility and ease of extension.
- **Data Storage**: Utilizes SQLite for metadata management and ChromaDB for vector storage, supporting efficient data retrieval and processing.
- **Data population**: Data has already been populated in the SQLite DB for testing purposes. Currently, it supports three agents - Organizational Information, Customer Database Search, and Internet Search. It can be expanded to include more agents as necessary.

This architecture emphasizes scalability, modularity, and performance for real-time applications.



Backend Query Processing Flow

The backend query flow is designed to ensure efficient query handling, clarification, and resolution through a structured, modular approach. The process emphasizes intelligent query enhancement, agent selection, and fallback mechanisms to guarantee accurate responses.

Flow Overview

1. User Query Initiation
    a. The process begins with the user submitting a query.
2. Query Refinement
    a. The query undergoes improvement through generate_better_query_or_ask_for_clarification().
    b. If the query remains ambiguous, clarification is requested from the user (Request Clarification), ending the current cycle.
3. Agent Capability Check
    a. The refined query is passed to check_if_current_agent_can_answer().
    b. If the current agent can answer, it processes the query using the existing agent.
4. Agent Retrieval for Unanswered Queries
    a. If the current agent cannot handle the query, the system invokes get_relevant_agents() to identify other potential agents.
    b. A vector database similarity search is performed to locate agents with relevant capabilities.
5. Agent Selection
    a. Agents Found: The most relevant agent is selected (Select Most Relevant Agent) and switched (Switch to Selected Agent) to process the query.
    b. No Agents Found: The system falls back to either the current agent or an internet search for query resolution.
6. Query Processing
    a. The selected or fallback agent processes the query, ensuring a seamless user experience.

More details are in the GitHub repo README.md and code comments in general.
https://github.com/Rugz007/capstone/blob/main/README.md

Models Comparison

Model	Check if the user query is good enough	Check if the current agent can answer well	Get the most relevant agent from similarity search	Speed	Costs (Dollars per 1M tokens)
Qwen Coder 2.5 7B	✅	✅	✅	⚡⚡⚡	0.3
Qwen 2.5 Instruct 14B	✅	✅	✅	⚡⚡	0.4
Llama 3.2 3B	✅	⚠️	✅	⚡⚡⚡⚡	0.06
Llama 3.2 1B	⚠️	❌	⚠️	⚡⚡⚡⚡	0.06
Open AI 4o mini	✅	✅	✅	⚡⚡	0.6

Competitor Analysis

This section briefly outlines how the framework of our dispatcher, the IntelliBot Pro, compares to the framework of OpenAI's Swarm.

OpenAI Swarm Framework Overview

Structure:

- Multi-agent system with lightweight collaboration.
- Agents are dynamically assigned tasks based on specialization.
- Context is retained throughout agent transitions.

Highlights:

- Efficient agent collaboration.
- Seamless handoffs between agents for complex tasks.

IntelliBot Pro Framework

Components:

- RAG Bots (Retrieval-Augmented Generation Bots).
- API Communication.
- Internet Search Bot.

Key Features:

- Real-time document updates.
- Automated metadata generation.
- Optimized accuracy and efficiency.

The OpenAI Swarm Framework is designed for lightweight, general-purpose collaboration with multi-agent systems, ideal for simpler and generalized tasks. On the other hand, the IntelliBot

Pro Framework focuses on enterprise-grade triage resolution with domain-specific expertise, dynamic real-time updates, and fallback mechanisms like internet search to ensure accuracy and efficiency.

Lessons Learned

Working on this Capstone Project with Nexteer provided us with invaluable insights into the complexities and challenges of building AI systems in production. Unlike theoretical models that assume clean, structured data, we faced the reality of working with unstructured and inconsistent inputs. For example, the **lack of clean data for testing** required us to generate **synthetic data**, ensuring robust model evaluation and performance tuning. This experience emphasized the importance of innovative data preprocessing methods when working in real-world scenarios where clean datasets are unavailable.

Another key challenge was addressing **ambiguous user queries**, which are common in practical deployments. Users often provide incomplete or vague inputs, requiring the system to intelligently handle such cases. We tackled this by adding **clarifying question mechanisms**, allowing the system to request more context before processing the query. This iterative approach improved user-agent interactions and ensured the chatbot dispatcher provided relevant and accurate responses.

Finally, integrating modern AI components with **legacy systems** underscored the need for flexibility and adaptability in production environments. By building **modular APIs**, we enabled seamless communication between systems, demonstrating that modular architectures are essential for scalability and long-term maintainability. These challenges taught us the importance of balancing technical innovation with practical constraints, a lesson we believe will be invaluable as we continue to work on AI solutions in the future.

Suggestions for Future Work

As the chatbot dispatcher continues to evolve, the next steps should focus on ensuring user safety and establishing robust monitoring mechanisms. These improvements will enhance system reliability, build user trust, and offer valuable insights for ongoing optimization.

Integrate Feedback Loop for Monitoring

- **Explicit User Feedback**: Implement mechanisms to capture user feedback on chatbot performance directly. For example, allow users to rate responses or flag issues, enabling the system to identify areas needing improvement.
- **Scoring System**: Develop a scoring system that aggregates user ratings and monitors trends in chatbot performance. This will act as an early warning system to alert administrators about potential performance drops.

Add Safeguards and Guardrails for User Safety

- **User Query Filtering**: Introduce filters that analyze user queries to block harmful or inappropriate inputs, ensuring the chatbot only processes safe and relevant queries.
- **Specialized Language Model**: Integrate a lightweight language model designed to detect hallucinations or biased responses. This will ensure responses are accurate, unbiased, and aligned with Nexteer's safety standards.

By focusing on these areas, Nexteer can further enhance the system's robustness and user satisfaction while maintaining the highest safety and reliability standards.

Appendix 1: Access to our GitHub Repository

To help you get started with our project repository, please follow these steps:

1. **Request Repository Access**:
   Contact our repository manager **Rugved Somwansh** to request access to the repository.
   Email: rugvedsomwanshi007@gmail.com
2. **Access the Repository**:
   Once access is granted, visit the repository at: https://github.com/Rugz007/capstone

   Clone the repository to your local machine using the following command:

```
Unset
git clone https://github.com/Rugz007/capstone.git
```

3. **Follow the Setup Guide**:

   Refer to the **Quick Start** and **Installation Instructions** in this README to set up both the frontend and backend environments.
4. **Troubleshooting and Support**:
   If you encounter any issues, feel free to reach out to our support team or repository manager for assistance.

We look forward to seeing you succeed with the **Nexteer AI Dispatcher system**! 🚀