ShopWise Solutions Documentation

Overview

ShopWise Solutions, a rapidly expanding e-commerce company based in Austin, Texas, is developing an AI-driven customer support assistant for its online platform. This assistant aims to provide effective first-line support for customer inquiries regarding products, orders, returns, and refunds. The assistant is integrated with our existing e-commerce infrastructure to enhance customer experience while minimizing manual intervention.

This document outlines the solution architecture, decisions made during development, tools and frameworks used, and the overall approach to meet the requirements.

Solution Architecture

High-Level Overview

The system comprises a frontend, a backend, and the AI assistant interface. The entire solution is hosted on an AWS EC2 instance to ensure reliability and scalability. The solution is designed as follows:

- **Frontend**: Built using TypeScript, React, and Vite to provide a fast and responsive user experience.
- **Backend**: Implemented using Bun, which facilitates efficient JavaScript and TypeScript runtime performance, connecting with the AI models and database.
- **AI Integration**: The AI assistant leverages the OpenAI model through its API, focusing on natural language processing to provide accurate responses.
- **Containerization**: Docker is used for containerizing the entire application, making it portable and easily deployable across different environments.

Detailed Components

1. **Frontend**: Vite was chosen for its fast build process and modern development capabilities, supporting TypeScript and React seamlessly. The user interface ensures a smooth interaction experience with the AI assistant, where customers can ask questions about products, order status, returns, and more.

2. **Backend**: The backend is powered by Bun to leverage its modern JavaScript and TypeScript capabilities for optimal runtime performance and fast startup times. It serves as the main orchestration layer, interacting with the AI models and managing customer queries.

3. **AI Model**: After a comprehensive evaluation of different large language models (LLMs), including Ollama 3.2 (1B and 5B parameters), Granite3, and T5, we concluded that OpenAI's GPT-4 offered the best balance of performance, robustness, and accuracy for our use case. OpenAI's models provide state-of-the-art natural language understanding and avoid common issues such as hallucinations.

4. **Containerization and Deployment**: Docker was selected for its scalability, containerization efficiency, and support for microservices-based architecture. Docker ensures that each component (frontend, backend, and AI assistant) runs in isolated environments, reducing compatibility issues.

5. **Hosting**: We deployed our solution on an AWS EC2 instance, which provides high availability and scalability for the application's production environment.

Technology Decisions

LLM Selection

We reconsidered multiple LLMs to determine the best fit for the customer support assistant. Our initial considerations included using the LLM locally. However, we ultimately opted for a cloud-based approach using OpenAI's API for enhanced scalability and ease of integration.

We also considered using the LLM for local inference, specifically for the search and embedding phases. Initially, we explored models like **nomic-embed** and **all-MiniLM-L6-v2** for embeddings. Ultimately, OpenAI's GPT-4 provided the most consistent results for our requirements, as it offered superior contextual understanding.

Our initial considerations included:

- **Ollama 3.2** (1B, 5B parameters): Offered decent language capabilities but lacked some contextual depth required for complex customer queries.
- **Granite3 and T5**: These models provided promising results for certain types of conversations but required extensive fine-tuning.

Ultimately, we opted for **OpenAI's GPT-4** because:

- **Accuracy and Robustness**: It provides accurate responses, supports multi-turn dialogues, and avoids hallucinations better than other models.
- **Ease of Integration**: OpenAI's API made it simple to integrate into our existing infrastructure, without the need for extensive custom fine-tuning.
- **Scalability**: The cloud-hosted API allowed for easy scaling of requests, fitting well with our goal to offer a responsive support system.

TypeScript vs Python

We chose **TypeScript** over Python for the following reasons:

- **Strong Typing**: TypeScript's static typing helps catch errors during development, making the codebase more maintainable and reducing runtime errors.

- **Frontend Integration**: Since we use React for the frontend, using TypeScript across both frontend and backend brought consistency and made collaboration more streamlined.

Docker and Scalability

- Docker was chosen over competitors for containerization due to its wide adoption, extensive community support, and compatibility with AWS EC2. It allows easy orchestration, ensuring the application can be scaled and maintained effectively.
- Docker Compose is used to define and manage multi-container setups, including the frontend, backend, and supporting services.

Vector Database and Context Management

To manage conversation context and ensure relevant responses, we implemented the following:

- **ChromaDB**: We used **ChromaDB** as a vector database to store embedded data from the datasets (`synthetic-product-data.csv` and `synthetic-orders-data.csv`). This allows for efficient semantic search and context-aware responses.
- **Embeddings**: We utilized `nomic-embed` and `all-MiniLM-L6-v2` for embedding data. These embeddings were stored in ChromaDB for effective vector similarity search, supporting context management and accurate response generation.
- **In-Memory Context Handling**: Each user request and response is kept in-memory to provide a consistent conversational experience. This allows the model to maintain context across multi-turn interactions and deliver more personalized responses.

Development Process

Approach

1. **Planning and Research**: Initial research involved evaluating the right AI model, development frameworks, and tools. We documented all our considerations and decided on the stack based on performance, compatibility, and scalability.
2. **Backend Implementation**: Developed using Bun with TypeScript, the backend serves as the central coordination layer. Integration with OpenAI's API was prioritized to ensure accurate natural language understanding.
3. **Frontend Implementation**: Built using React and TypeScript to offer an intuitive user interface for customers to interact with. Vite was used to optimize the development process.
4. **Deployment**: Docker was used for containerization, and AWS EC2 was selected for hosting to ensure availability and scalability.

Dataset Integration

Our solution integrates two provided datasets: `synthetic-product-data.csv` and `synthetic-orders-data.csv`. These datasets are used to handle customer queries regarding product information, availability, and order tracking. The integration is done as follows:

- **Product Dataset**: The `synthetic-product-data.csv` contains information about various products, such as product names, categories, and prices. This data is used by the AI assistant to provide product comparisons, availability checks, and suggestions.
- **Order Dataset**: The `synthetic-orders-data.csv` contains order-related information, including order status, shipping details, and return eligibility. This dataset is used to respond to customer inquiries related to order tracking, return status, and order history.
- **Data Access**: The backend reads from these datasets in real-time to provide accurate responses, ensuring that customers receive up-to-date information regarding their queries.
- **ChromaDB for Embeddings**: The embeddings for product and order data are stored in ChromaDB. This enables semantic searches to provide the most contextually relevant information based on user prompts.

Performance and Optimization

Performance Metrics

Our solution aims to meet several performance metrics as outlined in the requirements:

- **Accuracy**: The AI model is tested against a set of standard prompts to ensure accurate responses. We also ensure that the model avoids hallucinations by providing fact-checked information only from the provided datasets.
- **Response Time**: The solution is optimized for low latency. By using Bun for the backend, which offers fast startup times, and deploying on an AWS EC2 instance, we minimize the response time. Docker helps isolate services, reducing interference and ensuring consistent performance.

Performance Implementation and Improvement

- **Caching**: To reduce response times for frequently asked questions, caching strategies are employed. Data from common queries are stored temporarily to allow faster retrieval.
- **Asynchronous Processing**: The backend handles requests asynchronously, allowing multiple queries to be processed simultaneously without affecting performance.
- **Scalability**: The use of Docker containers allows us to scale individual components (e.g., backend services, AI integration) horizontally based on load, ensuring that increased traffic does not degrade performance.
- **Monitoring and Metrics**: AWS CloudWatch is used to monitor the application's performance. Metrics such as response times, error rates, and resource usage are tracked to identify bottlenecks and optimize the system continuously.

Semantic Search and Similarity Evaluation

- **Semantic Evaluation**: Each user prompt undergoes a semantic evaluation using **cosine similarity** to assess its relevance to the context stored in-memory. This evaluation helps determine the most appropriate response by

considering previous interactions.

- **Similarity Threshold**: We used a **similarity index of 0.25** as a threshold for determining whether a response is contextually and semantically relevant. This threshold was chosen based on benchmarking tests that evaluated different similarity scores. The benchmark revealed that 0.25 provided the best balance between avoiding irrelevant matches and maintaining flexibility for varied user queries.

Flow of User Interaction

1. **User Prompt**: The user enters an initial prompt through the frontend.
2. **System Prompts**: The application has three predefined system prompts that handle different types of requests:
    - **Generic Prompt**: Manages general inquiries.
    - **Product-Specific Prompt**: Focuses on questions related to products.
    - **Order-Specific Prompt**: Handles queries related to orders and shipping.
3. **In-Memory Context**: Each request and response is kept in memory, allowing the model to maintain the conversation context.
4. **Semantic Evaluation**: The model performs semantic evaluation using cosine similarity on the in-memory context to generate a contextually relevant response.
5. **Custom Response**: Based on the similarity evaluation, the model provides a custom response tailored to the user's prompt.

Submission Requirements

Deployment and GitHub Repository

- **Website**: The AI-powered assistant is deployed on an accessible AWS EC2 instance, with a dedicated URL for accessing the service.
- **GitHub Repository**: The complete source code is available in a public GitHub repository. The repository includes comprehensive documentation, code comments, and a detailed README that guides users through the setup and usage of the solution.

Testing Instructions

- **Accessing the Assistant**: Users can interact with the deployed AI assistant via the provided URL. The README in the GitHub repository contains step-by-step instructions for accessing the website.
- **Local Testing**: For those unable to access the live deployment, instructions for setting up the solution locally are included. Docker Compose is used to simplify the local setup, allowing users to run the frontend, backend, and AI components in isolated containers.
- **Usage Examples**: The GitHub repository also includes usage examples demonstrating different types of queries the assistant can handle, such as product comparisons, order tracking, and return eligibility checks.

Documentation and ADRs

To ensure the architecture and decisions are properly captured, **Architecture Decision Records (ADRs)** have been used for key decisions. This includes the selection of LLM, TypeScript over Python, Docker for containerization, and AWS as the hosting environment.

Architecture Decision Records (ADRs)

1. **LLM Selection (ADR-001)**

    - **Context**: Various LLMs were considered to provide the best customer support experience.
    - **Decision**: Chose OpenAI's GPT-4 over Ollama 3.2, Granite3, and T5.
    - **Reasoning**: GPT-4 offers superior accuracy, multi-turn dialogue support, and ease of integration via API.
    - **Consequences**: Enabled faster development with minimal fine-tuning required, but with a reliance on an external API service.

2. **Programming Language (ADR-002)**

    - **Context**: Consideration between TypeScript and Python for backend development.
    - **Decision**: Chose TypeScript over Python.
    - **Reasoning**: TypeScript provides strong typing, aligns well with the frontend (React), and facilitates maintainable code.
    - **Consequences**: Improved code consistency across the project but required the team to adapt to TypeScript-specific practices.

3. **Containerization Technology (ADR-003)**

    - **Context**: Needed a containerization solution for deployment and scalability.
    - **Decision**: Chose Docker for containerization.
    - **Reasoning**: Docker's wide adoption, community support, and compatibility with AWS EC2 made it an ideal choice for containerizing the solution.
    - **Consequences**: Simplified deployment and scaling but introduced the need for Docker expertise within the team.

4. **Hosting Environment (ADR-004)**

    - **Context**: Hosting solution needed to be reliable and scalable.
    - **Decision**: Chose AWS EC2 for hosting.
    - **Reasoning**: AWS EC2 provides a scalable, reliable, and well-supported environment for hosting our solution.
    - **Consequences**: Provided a robust hosting environment but added cloud infrastructure costs.

Conclusion

Our AI-powered customer support assistant is built to be robust, scalable, and easy to use. Each component of the system was selected and implemented with a focus on reliability, responsiveness, and a seamless customer experience. We believe this solution meets the requirements outlined by ShopWise Solutions and provides a scalable approach for future expansions.