Customer Ticket Classification System Documentation

Introduction

This document provides comprehensive documentation for the AI-powered Customer Ticket Classification System. This system leverages LangChain, OpenAI, and MongoDB to automatically categorize customer support tickets by severity and generate appropriate responses based on priority levels. It is designed to streamline customer support operations by intelligently classifying incoming tickets, providing confidence scores, and offering a robust API for integration with existing systems.

1. Overview

The Customer Ticket Classification System is an AI-driven solution aimed at automating and enhancing customer support workflows. By integrating cutting-edge technologies like LangChain for AI orchestration, OpenAI for intelligent classification and response generation, and MongoDB for data persistence (optional, with in-memory storage for demo purposes), the system provides a robust and scalable platform for managing customer inquiries.

1.1 Key Features

- **Intelligent Classification**: Utilizes OpenAI GPT models via LangChain to accurately classify tickets by category and priority.
- **Severity-Based Responses**: Generates contextually appropriate and tailored responses based on the determined ticket severity.
- **Memory Integration**: Maintains conversation context using LangChain's memory capabilities, leading to improved classification accuracy over time.
- **Confidence Scoring**: Provides a confidence score for each classification, flagging low-confidence predictions for manual review, ensuring accuracy and reliability.

- **REST API**: Offers a comprehensive and well-documented RESTful API with validation, error handling, and rate limiting, facilitating seamless integration with existing customer support platforms.
- **Batch Processing**: Supports efficient processing of multiple tickets simultaneously, enhancing throughput for high-volume environments.
- **Analytics**: Provides real-time statistics and trends on ticket classifications, enabling data-driven insights into support operations.
- **Admin Controls**: Includes administrative functionalities for configuration management and system monitoring.

1.2 Priority Levels and Response Styles

The system defines four distinct priority levels, each associated with specific use cases, response styles, and Service Level Agreements (SLAs):

Priority	Use Cases	Response Style	SLA
Critical	System outages, security breaches, data loss	Urgent, apologetic, action-oriented	Immediate
High	Major functionality broken, VIP customer issues	Professional, empathetic, solution-focused	2 hours
Medium	Minor bugs, standard customer requests	Helpful, friendly, informative	24 hours
Low	General questions, feature requests	Courteous, educational, patient	48 hours

This structured approach ensures that customer inquiries are handled with the appropriate urgency and tone, aligning with business priorities and customer expectations.

2. Technology Stack

The Customer Ticket Classification System is built upon a modern and robust technology stack, primarily focusing on AI and data management. The core components include:

2.1 LangChain

LangChain is a powerful open-source framework designed to simplify the development of applications powered by large language models (LLMs). It provides a structured way to chain together various components, such as LLMs, prompt templates, and memory modules, to create complex and context-aware applications. In this system, LangChain is instrumental in:

- **Orchestrating AI interactions**: Managing the flow of information between the application and the OpenAI models.
- **Handling conversational memory**: Maintaining context across multiple interactions, allowing the AI to provide more accurate and relevant classifications and responses.
- **Integrating with external data sources**: Although not extensively used for external data in this specific implementation beyond the ticket information, LangChain's capabilities extend to integrating with various data sources for richer context.

2.2 OpenAI

OpenAI provides the advanced artificial intelligence models that are at the heart of the classification and response generation capabilities of this system. Specifically, the system leverages OpenAI's GPT (Generative Pre-trained Transformer) models for:

- **Intelligent Ticket Classification**: Analyzing the subject and description of incoming tickets to determine their category and priority.
- **Contextual Response Generation**: Crafting appropriate and nuanced responses based on the classified severity and the conversation history.
- **Confidence Scoring**: Providing a measure of certainty for each classification, which is crucial for identifying cases that may require human intervention.

2.3 MongoDB

MongoDB is a popular NoSQL document database that offers high performance, high availability, and easy scalability. While the system can operate with in-memory storage for

demonstration purposes, MongoDB is the recommended solution for persistent data storage in a production environment. Its key benefits for this system include:

- **Flexible Schema**: MongoDB's document-oriented nature allows for flexible data models, which is well-suited for storing diverse ticket information and classification results.
- **Scalability**: It can easily scale horizontally to handle large volumes of data and high traffic, making it suitable for growing customer support operations.
- **Integration**: Seamless integration with Node.js applications, providing efficient data access and manipulation.

2.4 Other Technologies

In addition to the core technologies, the system also utilizes:

- **Node.js**: The JavaScript runtime environment for building the backend application.
- **Express.js**: A fast, unopinionated, minimalist web framework for Node.js, used for building the REST API.
- **TypeScript**: A superset of JavaScript that adds static typing, enhancing code quality and maintainability.
- **Zod**: A TypeScript-first schema declaration and validation library, used for robust API request validation.

3. Installation and Setup

This section outlines the steps required to set up and run the Customer Ticket Classification System. Follow these instructions carefully to get the system operational.

3.1 Prerequisites

Before proceeding with the installation, ensure you have the following software installed on your system:

- **Node.js**: Version 18 or higher. You can download it from the official Node.js website.
- **OpenAI API Key**: A valid API key from OpenAI is required for the AI classification and response generation functionalities. You can obtain one by signing up on the OpenAI platform.
- **MongoDB (Optional)**: While the system can use in-memory storage for demonstration purposes, a MongoDB instance is recommended for persistent data storage in a production environment. You can install MongoDB locally or use a cloud-hosted service like MongoDB Atlas.

3.2 Installation Steps

1. **Extract the Project Files**: Begin by extracting the project files to your desired directory. If you obtained the project from a Git repository, clone it using the following command:
2. **Install Dependencies**: Navigate to the project root directory and install the necessary Node.js dependencies using npm:
3. **Configure Environment Variables**: The system relies on environment variables for configuration, particularly for the OpenAI API key. A template file `.env.example` is provided. Copy this file to `.env` :
4. **Start the Server**: Once the dependencies are installed and the environment variables are configured, you can start the development server:
5. **Run the Demo (Optional)**: To see the classification system in action with sample tickets, you can run the demo script:

4. API Endpoints

The system exposes a comprehensive RESTful API for managing customer tickets, performing analytics, and administering the system. All API interactions are performed over HTTP/S.

4.1 Core Ticket Operations

These endpoints are used for submitting, retrieving, listing, and providing feedback on customer tickets.

4.1.1 Submit a New Ticket

- **Endpoint**: `POST /api/tickets`
- **Description**: Submits a new customer ticket for classification and response generation.
- **Content-Type**: `application/json`
- **Request Body Example**:
- **Expected Response Format**:

4.1.2 Get Specific Ticket

- **Endpoint**: `GET /api/tickets/{ticketId}`
- **Description**: Retrieves details of a specific ticket using its unique ID.

4.1.3 List Tickets with Filters

- **Endpoint**: `GET /api/tickets?priority={priority}&category={category}&limit={limit}`
- **Description**: Lists tickets, with optional filtering by priority, category, and limiting the number of results.
- **Example**: `GET /api/tickets?priority=Critical&category=Technical&limit=10`

4.1.4 Submit Feedback

- **Endpoint**: `PUT /api/tickets/{ticketId}/feedback`
- **Description**: Allows submission of feedback on a ticket's classification, enabling correction and system improvement.
- **Content-Type**: `application/json`
- **Request Body Example**:

4.1.5 Batch Process Tickets

- **Endpoint**: `POST /api/tickets/batch`
- **Description**: Processes multiple tickets in a single request.
- **Content-Type**: `application/json`
- **Request Body Example**:

4.2 Analytics Endpoints

These endpoints provide access to real-time statistics and trends related to ticket classifications.

4.2.1 Get Classification Statistics

- **Endpoint**: `GET /api/analytics/stats`
- **Description**: Retrieves overall classification statistics.

4.2.2 Get Trends Over Time

- **Endpoint**: `GET /api/analytics/trends?period={period}`
- **Description**: Provides trends in ticket classification over a specified period (e.g., `day`, `week`, `month`).

4.3 Admin Endpoints

These endpoints are for system administration and require an `x-api-key` header for authentication.

4.3.1 Get System Configuration

- **Endpoint**: `GET /api/admin/config`
- **Description**: Retrieves the current system configuration.
- **Required Header**: `x-api-key: your-api-key`

4.3.2 Update Confidence Threshold

- **Endpoint**: `PUT /api/admin/config/confidence-threshold`
- **Description**: Updates the confidence threshold for flagging tickets for manual review.
- **Required Header**: `x-api-key: your-api-key`
- **Content-Type**: `application/json`
- **Request Body Example**:

4.3.3 Clear AI Memory

- **Endpoint**: `POST /api/admin/memory/clear`
- **Description**: Clears the AI's conversational memory.
- **Required Header**: `x-api-key: your-api-key`

4.3.4 System Health Check

- **Endpoint**: `GET /api/admin/health`
- **Description**: Checks the overall health and status of the system.
- **Required Header**: `x-api-key: your-api-key`

5. Project Structure

The project is organized into a modular and logical structure to enhance maintainability and scalability. Below is an overview of the main directories and their contents:

Plain Text

```
customer-support-agent/
├── INSTALLATION.md        # Detailed installation guide
├── README.md              # Project overview and quick start
├── POSTMAN_GUIDE.md       # Guide for using the Postman collection
├── package.json           # Node.js project metadata and dependencies
├── package-lock.json      # Records the exact versions of dependencies
├── postman-collection.json# Postman collection for API testing
├── tsconfig.json          # TypeScript compiler configuration
├── .env.example           # Example environment variables file
```

```
└── src/
    ├── app.ts                # Express application setup and middleware
integration
    ├── server.ts             # Main server entry point
    ├── demo.ts               # Script for running a classification demo
with sample tickets
    ├── demo-enhanced.ts      # Enhanced demo script
    ├── classification/       # Contains AI classification logic
    │   ├── chains.ts         # LangChain orchestration for AI models
    │   ├── engine.ts         # Core classification engine logic
    │   ├── engine.test.ts    # Unit tests for the classification engine
    │   ├── memory.ts         # Manages conversational context/memory for
LangChain
    │   └── prompts.ts        # Defines AI prompt templates for
classification and response generation
    ├── config/               # Configuration files
    │   └── swagger.ts        # Swagger/OpenAPI documentation configuration
    ├── middleware/           # Express middleware functions
    │   ├── auth.ts           # Authentication middleware (e.g., for API
keys)
    │   ├── error.ts          # Centralized error handling middleware
    │   └── validation.ts     # Request validation middleware using Zod
    ├── routes/               # Defines API routes and their handlers
    │   ├── admin.ts          # Routes for administrative tasks
    │   ├── analytics.ts      # Routes for retrieving analytics and
statistics
    │   └── tickets.ts        # Routes for core ticket operations (submit,
get, list, feedback, batch)
    ├── types/                # TypeScript type definitions and interfaces
    │   └── ticket.ts         # Defines the structure of ticket objects and
related types
    └── utils/                # Utility functions
        └── textCleaner.ts    # Utility for cleaning text inputs
```

6. Configuration

The system's behavior can be customized through various environment variables. These variables are typically set in the `.env` file in the project root directory.

6.1 Environment Variables

Variable Name	Description	Required	Default Value	Example Val

`OPENAI_API_KEY`	Your OpenAI API key for accessing AI models.	Yes	None	`your_actual_`
`MONGODB_URI`	Connection string for your MongoDB instance. If not provided, in-memory storage is used.	No	In-memory	`mongodb://l`
`NODE_ENV`	Node.js environment (e.g., `development`, `production`). Affects logging verbosity.	No	`development`	`production`
`PORT`	The port on which the Express server will listen.	No	`3000`	`8000`
`CONFIDENCE_THRESHOLD`	The minimum confidence score for a classification to be considered reliable. Below this, tickets are flagged for manual review.	No	`70`	`80`

`API_KEY`	A secure API key for accessing administrative endpoints.	No	None	`your-secure-`

6.2 Classification Categories

The system is pre-configured to classify tickets into the following categories:

- **Technical**: Software bugs, system errors, integration issues.
- **Billing**: Payment problems, subscription issues, refunds.
- **General Inquiry**: Questions, information requests, how-to guides.
- **Bug Report**: Confirmed software defects requiring fixes.
- **Feature Request**: Suggestions for new functionalities or improvements.
- **Account**: Issues related to user login, profile changes, or permissions.

These categories can be customized by modifying the AI prompt templates in `src/classification/prompts.ts`.

7. Available Scripts

The `package.json` file defines several convenient scripts to help with development, testing, and running the application:

Script Name	Command	Description
`npm run dev`	`ts-node src/server.ts`	Starts the development server with hot reloading, ideal for active development.
`npm run demo`	`ts-node src/demo.ts`	Runs a classification demo using a set of sample tickets.
`npm run demo:enhanced`	`ts-node src/demo-enhanced.ts`	Runs an enhanced version of the classification demo.

`npm run build`	`tsc`	Compiles the TypeScript source code into JavaScript, outputting to the `dist` directory.
`npm run start`	`node dist/server.js`	Starts the production server using the compiled JavaScript files.
`npm run test`	`jest`	Executes unit tests defined in the project.

8. Testing Examples

This section provides `curl` command examples to test different functionalities and severity levels of the ticket classification system. These examples demonstrate how to interact with the API endpoints.

8.1 Test Different Severity Levels

8.1.1 Critical Priority Ticket

This example demonstrates submitting a ticket that should be classified with `Critical` priority due to its urgent nature and impact.

Bash

```
curl -X POST http://localhost:8000/api/tickets \
  -H "Content-Type: application/json" \
  -d '{ \
    "subject": "URGENT: Complete system outage - all users affected", \
    "description": "Our entire platform is down. All customers are getting
500 errors when trying to log in. This started 30 minutes ago and is
affecting all users.", \
    "customerInfo": {"priority": "VIP"}, \
    "source": "email" \
  }'
```

8.1.2 Low Priority Ticket

This example demonstrates submitting a general inquiry ticket that should be classified with `Low` priority.

Bash

```bash
curl -X POST http://localhost:8000/api/tickets \
  -H "Content-Type: application/json" \
  -d '{ \
    "subject": "How to export my data?", \
    "description": "Hi, I would like to know how I can export all my data
from your platform. I need it for my records.", \
    "source": "web" \
  }'
```

8.2 Expected Response Format

Upon successful submission of a ticket, the API will return a JSON response similar to the following, containing the `ticketId`, `classification` details (category, priority, confidence, summary, suggested response, tags, reasoning), `processingTime`, and a flag indicating if `needsManualReview`.

JSON

```json
{
  "success": true,
  "data": {
    "ticketId": "TICKET-1704123456789-abc123def",
    "classification": {
      "category": "Technical",
      "priority": "Critical",
      "confidence": 95,
      "summary": "Complete system outage affecting all users",
      "suggestedResponse": "We are immediately escalating this critical
system outage to our senior engineering team. We understand the severity of
this issue and are treating it as our highest priority. Our team is actively
working to restore service and we will provide updates every 15 minutes until
resolved.",
      "tags": ["outage", "system", "critical", "urgent"],
      "reasoning": "This is a critical system failure requiring immediate
attention due to complete service disruption affecting all users."
    },
    "processingTime": 1250,
    "needsManualReview": false
  }
}
```

9. Performance Metrics

The system is designed for efficient processing and provides insights into its performance characteristics:

- **Classification Time**: Typically ranges from 800 to 2000 milliseconds per ticket, depending on the complexity of the input and the load on the OpenAI API.
- **Confidence Score**: For clear and unambiguous tickets, the system achieves a confidence score of 70-95%.
- **Throughput**: Capable of processing 30-60 tickets per minute, with built-in rate limiting to manage API usage.
- **Memory Usage**: The system maintains stable memory usage with periodic cleanup, with a base memory footprint of approximately 100MB plus additional memory for AI processing.
- **API Response Time**: Excluding the AI processing time, the API response time is typically less than 100 milliseconds.

10. Monitoring & Analytics

The system provides robust monitoring and analytics capabilities to give insights into its operation and performance:

- **Classification Accuracy Metrics**: Track how accurately tickets are being classified.
- **Processing Time Statistics**: Monitor the time taken to process tickets.
- **Category and Priority Distributions**: Visualize the distribution of tickets across different categories and priority levels.
- **Confidence Score Trends**: Observe trends in confidence scores over time to identify potential issues or improvements.
- **Low-Confidence Ticket Flagging**: Automatically flags tickets with low confidence scores for manual review, ensuring that critical cases are not mishandled.

11. Troubleshooting

This section addresses common issues that may arise during the installation or operation of the Customer Ticket Classification System.

11.1 Common Issues

1. **"Classification failed" errors**:
   - **Verify** `OPENAI_API_KEY` : Ensure your OpenAI API key is correctly set in the `.env` file.
   - **Check OpenAI API credits**: Confirm that your OpenAI API key has sufficient credits for usage.
   - **Internet connectivity**: Verify that your system has a stable internet connection to reach the OpenAI API.
2. **Rate limiting errors**:
   - **Wait for reset**: If you encounter rate limiting, wait for the rate limit window to reset.
   - **Reduce request frequency**: Adjust your application to send requests at a lower frequency.
   - **Check OpenAI usage limits**: Review OpenAI's official documentation for current rate limits and usage policies.
3. **Memory issues**:
   - **Clear classification memory**: Use the admin endpoint `POST /api/admin/memory/clear` to clear the AI's classification memory.
   - **Restart server**: Restarting the server (`npm run dev` or `npm run start`) will reset all in-memory state.
4. **Port already in use**:
   - If you encounter an error indicating that port 3000 (or your configured port) is already in use, you can either:

- Change the `PORT` environment variable in your `.env` file (e.g., `PORT=3001`).
- Identify and terminate the process currently using the port.

5. **Module not found errors**:
   - **Re-run** `npm install`: This often resolves missing dependency issues.
   - **Clean install**: Delete the `node_modules` directory and `package-lock.json` (if present), then run `npm install` again.
6. **TypeScript compilation errors**:
   - **Node.js version**: Ensure you are using Node.js version 18 or higher.
   - **Build check**: Run `npm run build` to check for specific TypeScript compilation errors and address them.

11.2 Debug Mode

For more detailed logging and debugging information, set the `NODE_ENV` environment variable to `development` in your `.env` file:

```
Plain Text

NODE_ENV=development
```

This will enable verbose logging, which can help in diagnosing issues.

12. Postman Testing

The project includes a Postman collection (`postman-collection.json`) that allows for easy testing of all API endpoints. Postman is a popular API platform for building and using APIs.

12.1 Importing the Collection

1. **Download Postman**: If you don't have Postman installed, download it from the official Postman website.

2. **Import Collection**: Open Postman and click on `File > Import` (or the `Import` button in the workspace). Select the `postman-collection.json` file from your project directory and import it.

12.2 Setting Up Environment Variables

To ensure the Postman collection works correctly, you need to set up environment variables within Postman:

1. **Create a New Environment**: In Postman, click on the `Environments` dropdown (usually at the top right) and select `Manage Environments`. Click `Add` to create a new environment.
2. **Add Variables**: Add the following variables to your new environment:
   - `baseUrl`: Set this to the URL where your server is running (e.g., `http://localhost:8000`).
   - `apiKey`: If you have configured an `API_KEY` in your `.env` file for admin endpoints, set this variable to that same key.
3. **Activate Environment**: Select your newly created environment from the `Environments` dropdown.

12.3 Running the Collection

Once the collection is imported and environment variables are set, you can run individual requests or the entire collection:

- **Individual Requests**: Expand the imported collection in the sidebar, select any request (e.g., `POST /api/tickets`), and click `Send` to execute it.
- **Run Collection**: To run all requests in the collection, click on the collection name in the sidebar and then click the `Run` button. This will open the Collection Runner, where you can configure and execute the tests.

13. Next Steps

Once the Customer Ticket Classification System is successfully running, consider the following steps for further development, integration, and production deployment:

1. **Customize Prompt Templates**: Tailor the AI behavior by modifying the prompt templates located in `src/classification/prompts.ts` to better suit your specific business needs and desired response styles.
2. **Adjust Confidence Thresholds**: Fine-tune the `CONFIDENCE_THRESHOLD` via the admin API (`PUT /api/admin/config/confidence-threshold`) to control when tickets are flagged for manual review, balancing automation with accuracy.
3. **Integrate with Existing Support Platform**: Connect the API endpoints with your current customer support ticketing system to automate ticket classification and response generation within your existing workflow.
4. **Set Up MongoDB for Persistent Storage**: For production environments, replace the in-memory storage with a persistent MongoDB database. Update the `MONGODB_URI` environment variable with your MongoDB connection string.
5. **Deploy to Production Environment**: Implement a robust deployment strategy for a production environment, considering aspects like load balancing, continuous integration/continuous deployment (CI/CD), and monitoring.
6. **Implement Proper User Authentication**: For multi-user environments, implement a more comprehensive user authentication and authorization system beyond simple API keys.
7. **Add Application Monitoring and Alerting**: Set up monitoring tools to track application performance, errors, and resource usage, with alerting mechanisms for critical issues.
8. **Implement Horizontal Scaling**: For high-volume scenarios, consider implementing horizontal scaling to distribute the load across multiple instances of the application.
9. **Enhance Security**: Further strengthen the application's security by adding more advanced rate limiting, input sanitization, and security headers.

14. License

This project is released under the MIT License. Please see the `LICENSE` file in the project root directory for full details.

15. Support

If you encounter any issues or have questions regarding the Customer Ticket Classification System, please follow these steps:

1. **Check the Troubleshooting Section**: Refer to Section 11 of this document for common issues and their solutions.
2. **Review Server Logs**: Examine the server logs for detailed error messages, which can provide valuable clues for debugging.
3. **Verify Required Files**: Ensure that all 21 required files, as listed in the `INSTALLATION.md` and `README.md` files, are present in your project directory.
4. **OpenAI API Key Validation**: Double-check that your OpenAI API key is valid and has sufficient credits.