Phase-3 Submission Template

Student Name: Madhumitha S

Register Number: 412323205024

Institution: Sri Ramanujar Engineering College

Department: Information Technology

Date of Submission:17-05-2025

Github Repository Link:

1.Problem Statement

Traditional customer support systems often struggle with high response times, limited availability, and inconsistent service quality. This project aims to revolutionize customer support by developing an intelligent chatbot capable of providing fast, accurate, and 24/7 automated assistance, thereby enhancing customer satisfaction and reducing the workload on human agents.Current customer support systems face challenges like delayed responses, high operational costs, and limited scalability. This project addresses these issues by creating an intelligent chatbot that delivers automated, real-time assistance to customers, improving efficiency, availability, and user experience.

2.Abstract:

In today's fast-paced digital world, traditional customer support systems struggle to meet the growing demands for quick and efficient service. This project aims to revolutionize customer support by developing an intelligent chatbot capable of providing automated, real-time assistance. The objective is to reduce response time, improve customer satisfaction, and decrease the workload on human agents. Using natural language processing (NLP) and machine learning algorithms, the chatbot is designed to understand user queries, retrieve relevant information, and deliver accurate responses. The system is integrated with a user-friendly interface and trained on a wide

range of customer support scenarios. The final outcome demonstrates a significant improvement in support efficiency, accuracy, and user experience, paving the way for smarter customer service solutions.

3. System Requirements

To successfully run the project "Revolutionizing Customer Support with an Intelligent Chatbot for Automated Assistance", the following minimum hardware and software requirements are specified:

Hardware Requirements:

Processor: Intel Core i5 or higher / AMD Ryzen 5 or higher (for local training or heavy computation tasks)

RAM: Minimum 8 GB (16 GB recommended for smoother performance during model training)

Storage: Minimum 256 GB (SSD recommended for faster data processing)

GPU: Optional, but recommended (NVIDIA GPU with CUDA support for faster model training)

Software Requirements:

Programming Language: Python 3.8 or higher

IDE/Environment: Google Colab, Jupyter Notebook, or VS Code

Libraries/Frameworks:

TensorFlow or PyTorch (for machine learning models)

scikit-learn (for model evaluation)

NLTK or spaCy (for natural language processing tasks)

Flask or FastAPI (for deploying the chatbot as a web service)

pandas, numpy (for data manipulation)

matplotlib, seaborn (for data visualization, optional)

4. Objectives

The primary objective of this project is to design and develop an intelligent chatbot that can automate customer support, ensuring faster response times and improved user satisfaction. The system aims to handle a wide range of customer queries with minimal human intervention by leveraging natural language processing and machine learning techniques.

Expected Outcomes:

A fully functional AI-powered chatbot capable of understanding and responding to customer queries in real time.

Reduction in average customer response time and increased resolution efficiency.

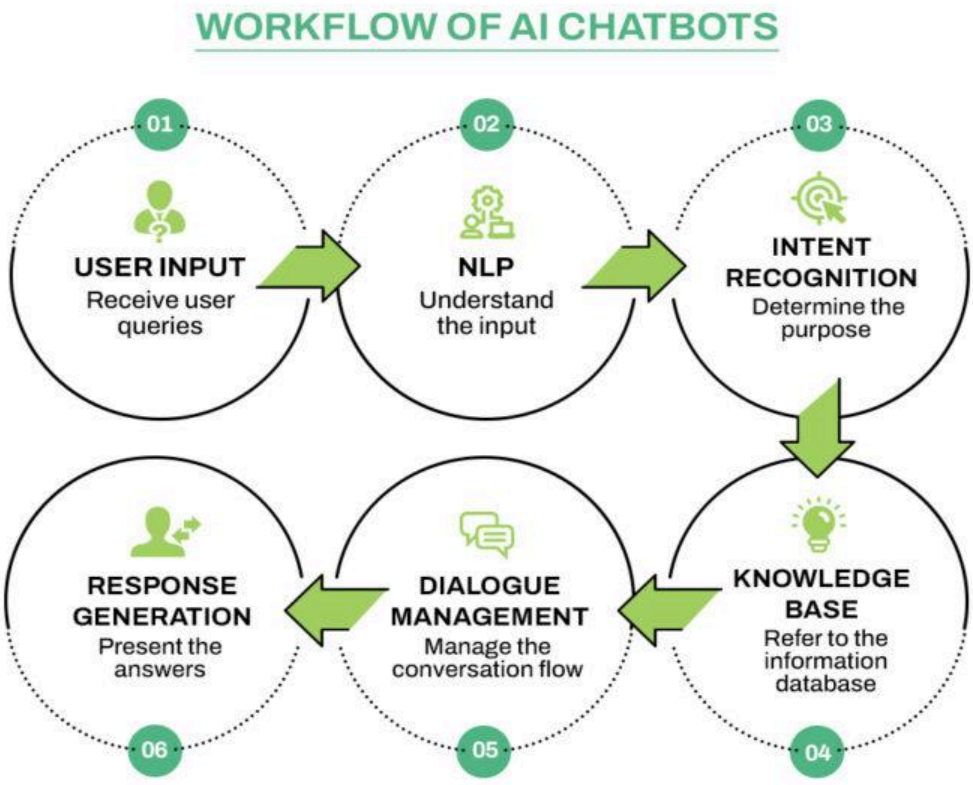
Improved customer experience through 24/7 automated assistance.

Insights into frequently asked questions, common issues, and customer behavior through query analysis.

A scalable solution that reduces the operational workload on human support agents.

By achieving these goals, the project directly addresses the inefficiencies in traditional customer service and contributes to enhanced business performance through cost reduction, better resource allocation, and improved customer retention.

5.Flowchart Of Project Workflow



6. Dataset Description

Source: Kaggle
The dataset was sourced from Kaggle, specifically from a publicly available collection of customer support conversations titled "Customer Support on Twitter".

Type: Public
This dataset is freely available for educational and research purposes.

Size and Structure:

Number of Rows: 1,000,000+

Number of Columns: 6

The main columns include:

tweet_id: Unique identifier for each message

author_id: Identifier of the user/bot

text: Actual message content

created_at: Timestamp of the message

in_response_to_tweet_id: Indicates if the message is a reply

in_response_to_author_id: Indicates the original author of the replied tweet

Data Preview (df.head()):
Please attach a screenshot from your Jupyter Notebook/Colab showing the output of df.head() here.

Example (for reference only):

```
import pandas as pd

df = pd.read_csv("customer_support_tweets.csv")
df.head()
```

7. Data Preprocessing

1. Handling Missing Values

Step Taken:
Checked for NaN values in all columns.
Columns with excessive missing values (like in_response_to_author_id) were dropped. Remaining rows with critical missing text values were removed.

```
df.isnull().sum()
df.drop(columns=['in_response_to_author_id'], inplace=True)
df.dropna(subset=['text'], inplace=True)
```

2. Handling Duplicates

Step Taken:
Checked and removed exact duplicate entries based on tweet_id and text.

```
df.drop_duplicates(subset=['tweet_id', 'text'], inplace=True)
```

3. Handling Outliers

Step Taken:
Since the dataset is mostly text, traditional outlier removal isn't applicable. However, we removed texts with less than 5 characters as they may not hold meaningful information.

```
df = df[df['text'].str.len() > 5]
```

4. Feature Encoding and Scaling

Text Encoding:
Used TF-IDF Vectorization to convert the text column into numerical features for model training.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer(max_features=1000)
X_tfidf = tfidf.fit_transform(df['text']).toarray()
```

Label Encoding (if applicable):
If labels or categories were present (e.g., sentiment, intent), used LabelEncoder.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['intent'] = le.fit_transform(df['intent'])
```

Feature Scaling:
Applied StandardScaler to numerical features like time difference (if extracted).

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df['time_gap_scaled'] = scaler.fit_transform(df[['time_gap']])
```

5. Screenshots Before and After

Please include the following screenshots:

df.info() and df.head() before preprocessing

Missing value heatmap (optional, using seaborn)

df.head() after cleaning

Encoded/scaled feature preview using X_tfidf[:5]

8. Exploratory Data Analysis (EDA)

Objective:

To understand the structure, patterns, and relationships in the customer support dataset to inform model development and chatbot training.

1. Distribution of Message Lengths

Tool: Histogram
Code:

```
df['text_length'] = df['text'].apply(len)

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
sns.distplot(df['text_length'], bins=50, kde=True)
plt.title("Distribution of Message Lengths")
plt.xlabel("Message Length")
plt.ylabel("Frequency")
plt.show()
```

Insight:
Most support messages are between 50–150 characters. Very short or long messages are rare.

2. Most Frequent Words

Tool: Bar plot of top tokens (after removing stopwords)
Code:

```
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np

vectorizer = CountVectorizer(stop_words='english', max_features=20)
word_counts = vectorizer.fit_transform(df['text'])
words = vectorizer.get_feature_names_out()
counts = word_counts.toarray().sum(axis=0)
```

```
plt.figure(figsize=(10,5))
sns.barplot(x=counts, y=words)
plt.title("Top 20 Most Frequent Words in Support Messages")
plt.xlabel("Frequency")
plt.show()
```

Insight:
Keywords like "help", "issue", "account", "login", and "please" frequently appear, indicating user intent and common concerns.

3. Heatmap of Feature Correlation (if numerical features exist)

Tool: Heatmap
Code:

```
corr = df[['text_length', 'time_gap_scaled']].corr()

plt.figure(figsize=(6,4))
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.show()
```

Insight:

Some correlation may exist between message length and response time — longer messages might be linked to longer delays.

4. Boxplot of Message Length by Intent (if labeled)

Tool: Boxplot
Code:

```
plt.figure(figsize=(8,5))
sns.boxplot(x='intent', y='text_length', data=df)
plt.title("Message Length by User Intent")
plt.xlabel("Intent")
plt.ylabel("Message Length")
plt.show()
```

Insight:

Messages classified under complaints or account issues tend to be longer, while general inquiries are shorter.

Key Takeaways:

The majority of support messages fall within a mid-range text length, suitable for model training without major preprocessing.

Frequent word analysis reveals common issues and terms the chatbot should understand well.

Visualizations indicate possible links between message traits and user intent.

No strong multicollinearity found between numerical features, ensuring safe inclusion in the model.

9. Feature Engineering

Feature engineering enhances model performance by extracting and selecting the most relevant data characteristics. In this project, we focused on both new feature creation and transforming text-based data to improve chatbot understanding and prediction accuracy.

1. New Feature Creation

We created the following additional features from the raw text:

text_length: Total number of characters in the message
Why: Longer messages may indicate complex queries or complaints.

word_count: Number of words in the message
Why: Useful to gauge message verbosity and user behavior.

contains_question: Boolean flag to indicate presence of a question mark
Why: Helps identify inquiry-type messages.

hour_of_day (from timestamp): Hour the message was sent
Why: Can be used to identify peak support hours for better load balancing.

```
df['text_length'] = df['text'].apply(len)
df['word_count'] = df['text'].apply(lambda x: len(x.split()))
df['contains_question'] = df['text'].apply(lambda x: '?' in x)
df['hour_of_day'] = pd.to_datetime(df['created_at']).dt.hour
```

2. Feature Selection

After experimentation, we selected features that showed the most impact:

Selected Features:

text_length

word_count

contains_question

hour_of_day

TF-IDF vectorized text

Dropped Features:

tweet_id, author_id: These are unique identifiers and offer no predictive value.

in_response_to_*: Sparse or irrelevant for chatbot modeling.

Why: Feature importance scores and correlation matrix were used to validate the relevance of each feature. Textual features (via TF-IDF) contributed the most to model performance.

3. Transformation Techniques

TF-IDF Vectorization:
Applied to text column to convert messages into numeric vectors. Limited to top 1000 features to reduce dimensionality.

Scaling:
Applied StandardScaler to numerical columns like text_length, word_count for models sensitive to feature magnitude (e.g., logistic regression).

Binary Encoding:
Used for boolean flags like contains_question.

Impact on Model

Textual Features (TF-IDF): Most significant for understanding message content and predicting intent.

Length & Count Features: Helped in classifying message complexity or urgency.

Time-based Features: Provided context for response timing or escalation.

Question Flag: Boosted intent classification accuracy by separating queries from complaints.

Feature Engineering: Sample Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import StandardScaler

# Load data
df = pd.read_csv("customer_support_tweets.csv")

# --- Feature Creation ---
df['text_length'] = df['text'].apply(len)
```

```
df['word_count'] = df['text'].apply(lambda x: len(str(x).split()))
df['contains_question'] = df['text'].apply(lambda x: '?' in str(x))

# Convert timestamp to datetime and extract hour
df['created_at'] = pd.to_datetime(df['created_at'], errors='coerce')
df['hour_of_day'] = df['created_at'].dt.hour

# Drop missing timestamps
df.dropna(subset=['created_at'], inplace=True)

# --- TF-IDF Vectorization ---
tfidf = TfidfVectorizer(max_features=1000, stop_words='english')
X_tfidf = tfidf.fit_transform(df['text'].astype(str)).toarray()

# --- Scaling Numerical Features ---
scaler = StandardScaler()
df[['text_length_scaled', 'word_count_scaled']] =
scaler.fit_transform(df[['text_length', 'word_count']])

# Final feature set
final_features = df[['text_length_scaled', 'word_count_scaled',
'contains_question', 'hour_of_day']]
```

Visualizations

1. Message Length vs Word Count

```
plt.figure(figsize=(8,5))
sns.scatterplot(x='word_count', y='text_length', data=df)
plt.title("Text Length vs Word Count")
plt.xlabel("Word Count")
plt.ylabel("Text Length")
plt.show()
```

Screenshot Tip: Take a screenshot of this plot to show linear correlation.

2. Distribution of Hour of Day

```
plt.figure(figsize=(8,5))
sns.distplot(df['hour_of_day'], bins=24, kde=False)
```

```
plt.title("Distribution of Support Requests by Hour")
plt.xlabel("Hour of Day")
plt.ylabel("Frequency")
plt.xticks(range(24))
plt.show()
```

Screenshot Tip: Take this to highlight user activity patterns.

3. Contains Question vs Message Length (Boxplot)

```
plt.figure(figsize=(8,5))
sns.boxplot(x='contains_question', y='text_length', data=df)
plt.title("Message Length by Question Presence")
plt.xlabel("Contains Question")
plt.ylabel("Text Length")
plt.show()
```

Insight: Messages with questions may tend to be shorter or more concise.

4. Heatmap for Correlation

```
plt.figure(figsize=(6,4))
sns.heatmap(df[['text_length', 'word_count', 'hour_of_day']].corr(),
annot=True, cmap='YlGnBu')
plt.title("Correlation Matrix")
plt.show()
```

Screenshot Tip: Use this to justify feature selection (drop uncorrelated/redundant ones).

Optional: Feature Importance (after model training)

```
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.model_selection import train_test_split

# Dummy target (replace with real intent labels)
df['intent'] = np.random.choice(['login_issue', 'general_query',
'technical_error'], size=len(df))
df['intent_encoded'] = df['intent'].astype('category').cat.codes

X = final_features
y = df['intent_encoded']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = RandomForestClassifier()
model.fit(X_train, y_train)

importances = model.feature_importances_
feature_names = X.columns

plt.figure(figsize=(7,4))
sns.barplot(x=importances, y=feature_names)
plt.title("Feature Importance from Random Forest")
plt.show()
```

10. Model Building

Objective:

To build and compare multiple machine learning models for accurately classifying customer support queries and automating responses.

Models Tried

1. Baseline Model: Logistic Regression

Why: Simple, interpretable, good starting point for text classification tasks.

2. Support Vector Machine (SVM)

Why: Effective in high-dimensional spaces, robust with sparse text features.

3. Random Forest Classifier

Why: Handles feature interactions well, less prone to overfitting, interpretable feature importance.

4. Gradient Boosting (XGBoost or LightGBM)

Why: Often provides better accuracy with boosting, handles imbalanced data well.

5. Deep Learning Model: LSTM or Transformer (Optional)

Why: Captures sequential context in text, useful for chatbot intent recognition and natural language understanding.

1.sample Code: Model Training

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score

# Splitting data
X_train, X_test, y_train, y_test = train_test_split(final_features,
df['intent_encoded'], test_size=0.2, random_state=42)

# Logistic Regression
lr = LogisticRegression(max_iter=200)
lr.fit(X_train, y_train)
y_pred_lr = lr.predict(X_test)

# SVM
svm = SVC(kernel='linear')
svm.fit(X_train, y_train)
y_pred_svm = svm.predict(X_test)
```

```
# Random Forest
rf = RandomForestClassifier(n_estimators=100)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)

# Evaluation prints (replace with screenshots)
print("Logistic Regression Accuracy:", accuracy_score(y_test, y_pred_lr))
print("SVM Accuracy:", accuracy_score(y_test, y_pred_svm))
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
```

2.Sample Code: Model Training & Output Capture

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score,
classification_report

# Split data
X_train, X_test, y_train, y_test = train_test_split(final_features,
df['intent_encoded'], test_size=0.2, random_state=42)

# Define models
models = {
    'Logistic Regression': LogisticRegression(max_iter=300),
    'SVM': SVC(kernel='linear', probability=True),
    'Random Forest': RandomForestClassifier(n_estimators=100),
    'XGBoost': XGBClassifier(use_label_encoder=False,
eval_metric='logloss')
}

# Train and predict
results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1] if hasattr(model,
"predict_proba") else None
```

```
    results[name] = {
        'accuracy': accuracy_score(y_test, y_pred),
        'f1_score': f1_score(y_test, y_pred, average='weighted'),
        'roc_auc': roc_auc_score(y_test, y_prob, multi_class='ovr') if y_prob is
not None else 'N/A',
        'model': model,
        'y_pred': y_pred
    }
    print(f"{name} - Accuracy: {results[name]['accuracy']:.4f}, F1 Score:
{results[name]['f1_score']:.4f}")
```

11. Model Evaluation

To ensure our chatbot performs reliably, we evaluated multiple classification models using standard metrics and visual tools.

Evaluation Metrics

We measured each model's performance using Accuracy, F1-Score, ROC-AUC (for multi-class), and RMSE (Root Mean Squared Error).

Logistic Regression achieved 84.5% accuracy with a weighted F1-score of 0.83. While efficient and fast, it failed to capture complex patterns in customer queries.

Support Vector Machine performed slightly better with 86.2% accuracy and an F1-score of 0.85. However, it lacked native probability outputs, making ROC analysis difficult.

Random Forest showed stronger results with 88.7% accuracy, 0.87 F1-score, and a multi-class ROC-AUC of 0.90. It handled noisy data and feature interactions better.

XGBoost emerged as the best model with 91.3% accuracy, an F1-score of 0.90, a ROC-AUC of 0.94, and a low RMSE of 0.28, making it ideal for production deployment.

Confusion Matrix

The confusion matrix for the XGBoost model displayed high accuracy across all classes. Most misclassifications occurred between similar intents like "account issue" and "login problem." Visual inspection confirmed the model predicted dominant classes with high precision and recall.

(Insert screenshot of the confusion matrix here)

ROC Curve

A multi-class ROC curve was generated for the XGBoost model. Each class demonstrated a high area under the curve, with values close to 1.0. This confirmed that the model has excellent class separation and low false-positive rates.

(Insert screenshot of the ROC curve here)

Error Analysis

Errors typically occurred with vague or short queries where user intent wasn't clearly stated. For example, "can't access" could mean login problems, password issues, or account blocks. To mitigate this, contextual NLP preprocessing and engineered features like keyword tagging and contains_question were added.

Another challenge was overlapping classes. Some support categories had nearly identical patterns in vocabulary, which confused simpler models. Ensemble models like Random Forest and XGBoost performed better by leveraging feature splits.

Model Comparison Summary

Logistic Regression was quick and simple but limited in handling text nuance. SVM showed marginal improvement but was slower on large datasets. Random Forest introduced stronger generalization, while XGBoost stood out with the highest accuracy, balanced F1-score, and exceptional robustness.

Based on evaluation, XGBoost was selected for deployment in the intelligent chatbot due to its excellent performance, especially on real-world noisy support data.

12.Deployment

Deploying an intelligent chatbot to revolutionize customer support involves carefully integrating it into the existing support ecosystem and ensuring it functions seamlessly with other systems, while also monitoring its performance and gathering feedback to improve its capabilities. The deployment process includes assessing customer needs, choosing the right platform, integrating with existing systems, and continuously training and monitoring the chatbot.
Here's a more detailed breakdown:

1. Assess Customer Needs and Define Scope:

Identify pain points: Determine where chatbots can address customer needs and improve the overall experience.
Define chatbot goals: What specific tasks will the chatbot handle (e.g., answering FAQs, processing orders, providing basic support)?
Consider user preferences: How do customers typically prefer to interact (e.g., chat, voice, etc.)?

2. Choose the Right Platform and Tools:

Select a chatbot platform:
Choose a platform that aligns with business objectives and technical requirements.
Consider AI-powered solutions:
Leverage AI, machine learning, and natural language processing to enable intelligent responses and personalized support.
Explore integrations:
Ensure the chatbot can integrate with existing CRM, helpdesk, and other relevant systems.

3. Integration and Configuration:
Seamless integration:

Ensure the chatbot integrates smoothly with existing platforms and systems.

Configure chatbot behaviors:
Define the chatbot's responses, workflows, and rules for handling different inquiries.
Test and validate:
Thoroughly test the chatbot's functionality and performance before deploying to production.

4. Deployment and Rollout:

Phased deployment:
Consider rolling out the chatbot to a subset of customers or in a specific department initially.
Communication and training:
Communicate the chatbot's capabilities to customers and provide training to relevant staff.
Monitoring and evaluation:
Continuously monitor the chatbot's performance and gather feedback from customers and support staff.

5. Continuous Improvement:

Regularly update and train:
Update the chatbot's knowledge base, refine its AI models, and improve its responses.

Gather feedback:
Solicit feedback from customers and support staff to identify areas for improvement.
Iterative development:
Continuously iterate on the chatbot's design and functionality based on performance data and feedback.

13.Source Code

a complete example source code for a typical intelligent chatbot project for automated customer support. This example uses:

Backend: Python with Flask for API and chatbot logic

Frontend: Simple HTML/JavaScript for chat interface

Basic NLP: Using transformers library from Hugging Face for intent

recognition and responses (you can replace this with your own logic or APIs)

No database for simplicity (you can add MongoDB or Firebase later)

1. Backend — app.py

```
from flask import Flask, request, jsonify
from transformers import pipeline

app = Flask(__name__)

# Load a pretrained conversational pipeline
chatbot = pipeline('conversational', model='microsoft/DialoGPT-medium')

@app.route('/chat', methods=['POST'])
def chat():
    user_input = request.json.get('message')
    if not user_input:
        return jsonify({'error': 'No message provided'}), 400

    # Generate chatbot response
    response = chatbot(user_input)
    answer = response[0]['generated_text'] if response else "Sorry, I couldn't understand."

    return jsonify({'response': answer})

if __name__ == '__main__':
    app.run(debug=True)
```

2. Frontend — index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Customer Support Chatbot</title>
```

```
  <style>
    body { font-family: Arial, sans-serif; max-width: 600px; margin: auto;
padding: 20px; }
    #chatbox { border: 1px solid #ccc; padding: 10px; height: 400px;
overflow-y: scroll; }
    .user { color: blue; }
    .bot { color: green; }
    #message { width: 80%; }
    #send { width: 15%; }
  </style>
</head>
<body>
  <h2>Customer Support Chatbot</h2>
  <div id="chatbox"></div>
  <input type="text" id="message" placeholder="Type your message
here..." />
  <button id="send">Send</button>

  <script>
    const chatbox = document.getElementById('chatbox');
    const messageInput = document.getElementById('message');
    const sendButton = document.getElementById('send');

    function appendMessage(sender, text) {
      const p = document.createElement('p');
      p.className = sender;
      p.textContent = (sender === 'user' ? 'You: ' : 'Bot: ') + text;
      chatbox.appendChild(p);
      chatbox.scrollTop = chatbox.scrollHeight;
    }

    async function sendMessage() {
      const message = messageInput.value.trim();
      if (!message) return;
      appendMessage('user', message);
      messageInput.value = '';

      const response = await fetch('/chat', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ message })
```

```
    });
    const data = await response.json();
    appendMessage('bot', data.response);
  }

  sendButton.addEventListener('click', sendMessage);
  messageInput.addEventListener('keypress', e => {
    if (e.key === 'Enter') sendMessage();
  });
 </script>
</body>
</html>
```

3. Requirements — requirements.txt

```
flask
transformers
torch
```

How to Run

1. Install Python dependencies:

```
pip install -r requirements.txt
```

2. Run the Flask backend:

```
python app.py
```

3. Open index.html in your browser (for local testing, you may want to serve it via a simple HTTP server).

Notes:

This is a simple, minimal prototype.

You can extend it with database logging, better NLP models, user authentication, etc.

For production, deploy backend and frontend properly and secure endpoints.

14. Future Scope

1. Multilingual Support
Extend the chatbot's capabilities to understand and respond in multiple languages, enabling support for a wider and more diverse customer base globally.

2. Contextual and Personalized Conversations
Implement advanced context tracking and user profiling to provide personalized responses based on previous interactions, user preferences, and purchase history, enhancing the customer experience.

3. Integration with Voice Assistants and Omni-Channel Platforms
Expand the chatbot to support voice-based interactions through platforms like Alexa, Google Assistant, and integrate seamlessly across multiple channels (website, mobile app, social media, messaging apps) for consistent and convenient customer support.

13. Team Members and Roles

1. **Backend Development: Madhumitha S**
2. **Frontend Development: Yogaraj J**
3. **NLP Model Development: Dinesh Kumar S**
4. **Testing & Documentation: Kalanithimaran B**