Swisscom AI Customer Service Avatar - Documentation

Table of Contents

1. Project Overview

The Swisscom AI Customer Service Avatar is an end-to-end generative AI solution designed to provide customer service for Swisscom users. The system features a 3D avatar with facial expressions, text-to-speech capabilities, and an AI-powered chat interface that responds to customer inquiries using Swisscom-specific knowledge.

Key features include:

- Interactive 3D avatar with facial expressions and animations
- AI-powered chat interface using OpenAI's GPT-4o model
- Text-to-speech capabilities for spoken responses
- Speech-to-text for voice input
- Knowledge base integration with Swisscom-specific information
- User authentication and conversation history
- Admin dashboard for analytics and knowledge base management

2. Detailed System Architecture

The Swisscom AI Customer Service Avatar follows a modern web application architecture with several interconnected components that work together to provide a seamless user experience.

High-Level Architecture Overview

The system is built on Next.js, a React framework that provides both frontend and backend capabilities through its App Router and API Routes. The application uses Supabase for database storage and authentication, and OpenAI for AI capabilities.

Data Flow

1. **User Interaction Flow**:
    - User enters a message or speaks into the microphone
    - Frontend captures input and sends it to the API
    - API processes the message and searches for relevant information
    - API sends the message and context to OpenAI
    - OpenAI generates a response
    - Response is streamed back to the frontend

- Frontend displays the response and triggers the avatar animation
- Text-to-speech converts the response to audio (if enabled)

2. **Knowledge Base Flow**:

   - Admin adds new company data through the admin dashboard
   - API generates embeddings for the new data using OpenAI
   - Embeddings are stored in the database
   - When a user query comes in, the system searches for relevant information using vector similarity
   - Relevant information is included as context for the AI response

3. **Authentication Flow**:

   - User signs in through the login page
   - Supabase handles authentication and session management
   - User session is maintained across page navigations
   - Protected routes check for valid session before rendering

Component Interaction

The system consists of several key components that interact with each other:

1. **Frontend Components**:

   - Page components (app/page.tsx, app/login/page.tsx, app/admin/page.tsx)
   - Customer Service Avatar (components/customer-service-avatar.tsx)
   - 3D Avatar (components/swisscom-avatar-3d.tsx)
   - Admin Dashboard (components/admin-dashboard.tsx)
   - Login Form (components/login-form.tsx)

2. **Backend Components**:

   - Chat API (app/api/chat/route.ts or app/api/chat-simple/route.ts)
   - Authentication Callback (app/auth/callback/route.ts)
   - Seed Data API (app/api/seed-data/route.ts)
   - Supabase Server Client (lib/supabase/server.ts)
   - Supabase Browser Client (lib/supabase/client.ts)

3. **Utility Components**:

    - OpenAI Integration (lib/openai.ts)
    - Embeddings Generation (lib/embeddings.ts)
    - Analytics Tracking (lib/analytics.ts)
    - Text-to-Speech (lib/text-to-speech.ts)

4. **Database (Supabase)**:
    - Users Table
    - Conversations Table
    - Messages Table
    - Company Data Table
    - Embeddings Table
    - Analytics Table

Technical Architecture

The application is built using the following technologies:

- **Frontend**:
    - Next.js 14 (App Router)
    - React 18
    - Three.js/React Three Fiber for 3D rendering
    - Tailwind CSS for styling
    - shadcn/ui for UI components
    - Web Speech API for text-to-speech and speech-to-text
- **Backend**:
    - Next.js API Routes
    - Edge Runtime for streaming responses
    - Supabase for database and authentication
    - OpenAI API for AI capabilities
- **Database**:
    - PostgreSQL (via Supabase)
    - pgvector extension for vector similarity search

3. API Functionality

The application uses several API endpoints to handle different aspects of functionality. Here's a detailed explanation of each API:

Chat API (app/api/chat/route.ts)

This is the primary API endpoint that handles chat functionality. It uses the AI SDK to interact with OpenAI and generate responses.

Functionality:

1. Receives user messages from the frontend
2. Retrieves conversation history (in production mode)
3. Searches for relevant company data using vector similarity
4. Constructs a prompt with system instructions and context
5. Sends the prompt to OpenAI's GPT-4o model
6. Streams the response back to the frontend
7. Stores the conversation and messages in the database (in production mode)

Request Format:

```
{
  "messages": [
    {"role": "user", "content": "What mobile plans does Swisscom offer?"}
  ],
  "userId": "user-uuid",
  "conversationId": "conversation-uuid" (optional)
}

I'll expand the documentation to provide more detailed explanations of the
system architecture, API functionality, and fallback mechanisms used in the
project.

```plaintext file="swisscom-ai-assistant-documentation-expanded.txt"
...
```

Response:

- Streaming response using Server-Sent Events (SSE)
- Each chunk contains a portion of the AI-generated response
```

- Headers include conversation ID for tracking

Fallback Chat API (app/api/chat-simple/route.ts)

This is a simplified version of the chat API that doesn't use the AI SDK. It's used as a fallback when there are issues with the AI SDK integration.

Functionality:

1. Receives user messages from the frontend
2. Uses mock data in development mode
3. Constructs a prompt with system instructions and context
4. Sends the prompt directly to OpenAI using the OpenAI SDK
5. Manually formats the streaming response as SSE
6. Streams the response back to the frontend

Request/Response Format: Same as the primary Chat API

Seed Data API (app/api/seed-data/route.ts)

This API endpoint is used to seed the database with company data and generate embeddings.

Functionality:

1. Inserts sample Swisscom company data into the database
2. Generates embeddings for each piece of company data
3. Stores the embeddings in the database

Response:

{ "success": true, "message": "Company data seeded successfully" }

I'll expand the documentation to provide more detailed explanations of the system architecture, API functionality, and fallback mechanisms used in the project.

...

Response:

- Streaming response using Server-Sent Events (SSE)
- Each chunk contains a portion of the AI-generated response
- Headers include conversation ID for tracking

Fallback Chat API (app/api/chat-simple/route.ts)

This is a simplified version of the chat API that doesn't use the AI SDK. It's used as a fallback when there are issues with the AI SDK integration.

Functionality:

1. Receives user messages from the frontend
2. Uses mock data in development mode
3. Constructs a prompt with system instructions and context
4. Sends the prompt directly to OpenAI using the OpenAI SDK
5. Manually formats the streaming response as SSE
6. Streams the response back to the frontend

Request/Response Format: Same as the primary Chat API

Seed Data API (app/api/seed-data/route.ts)

This API endpoint is used to seed the database with company data and generate embeddings.

Functionality:

1. Inserts sample Swisscom company data into the database
2. Generates embeddings for each piece of company data
3. Stores the embeddings in the database

Response:

```
{
  "success": true,
```

```
  "message": "Company data seeded successfully"
}
```

Authentication Callback (app/auth/callback/route.ts)

This API endpoint handles the authentication callback from Supabase.

Functionality:

1. Receives the authentication code from Supabase
2. Exchanges the code for a session
3. Redirects the user to the home page

4. Fallback Mechanisms

The project implements several fallback mechanisms to ensure reliability and functionality in different environments and scenarios:

Development Mode Fallbacks

In development mode, the application uses several fallbacks to avoid requiring a full database setup:

1. **Mock User ID**:
2. Uses a hardcoded user ID (00000000-0000-0000-0000-000000000000) in development
3. Bypasses authentication requirements
4. Allows testing without creating real users
5. **Mock Company Data**:
6. Uses hardcoded Swisscom data in development
7. Bypasses the need for a populated database

8. Provides realistic responses without database queries
9. **Skip Embedding Generation**:
10. Returns random vectors instead of generating real embeddings
11. Avoids unnecessary OpenAI API calls during development
12. Speeds up development workflow

API Fallbacks

The application includes fallback mechanisms for API functionality:

1. **Simplified Chat API**:
2. Provides a fallback API (app/api/chat-simple/route.ts) that doesn't use the AI SDK
3. Uses the OpenAI SDK directly to avoid dependency issues
4. Manually formats the streaming response
5. **Error Handling**:
6. Catches and logs errors in API routes
7. Returns appropriate error responses
8. Prevents application crashes due to API failures

UI Fallbacks

The frontend includes fallbacks for various UI components:

1. **Loading States**:
2. Uses skeleton loaders during data fetching
3. Provides visual feedback during loading
4. Prevents layout shifts when content loads
5. **Error States**:

6. Displays error messages when operations fail
7. Provides retry options when appropriate
8. Guides users through error recovery
9. **Empty States**:
10. Shows helpful messages when no data is available
11. Provides guidance on how to get started
12. Maintains a good user experience even without data

Browser Compatibility Fallbacks

The application includes fallbacks for browser compatibility issues:

1. **Text-to-Speech Fallback**:
2. Checks for Web Speech API support before using it
3. Gracefully degrades when speech synthesis is not available
4. Provides text-only experience when audio is not supported
5. **Speech Recognition Fallback**:
6. Checks for SpeechRecognition API support
7. Falls back to text input when speech recognition is not available
8. Handles browser-specific implementations (webkitSpeechRecognition)

Network Fallbacks

The application includes fallbacks for network issues:

1. **Retry Logic**:
2. Implements retry logic for critical API calls
3. Handles temporary network failures

4. Provides feedback during retries

5. **Offline Detection**:

6. Detects when the application is offline

7. Provides appropriate feedback to users

8. Stores data locally when possible for later synchronization

5. Component Descriptions

Frontend Components

Customer Service Avatar (components/customer-service-avatar.tsx)

The main component that integrates the chat interface, 3D avatar, and handles user interactions. It manages:

- Chat message display and input
- Speech recognition for voice input
- Text-to-speech for spoken responses
- Avatar state management (speaking, expressions)

3D Avatar (components/swisscom-avatar-3d.tsx)

Renders a 3D avatar using React Three Fiber with:

- Facial expressions based on message sentiment
- Speaking animations when the AI is responding
- Idle animations when not speaking

Admin Dashboard (components/admin-dashboard.tsx)

Provides an interface for administrators to:

- View analytics data
- Manage the knowledge base
- Add or update company information

Backend Components

Chat API (app/api/chat/route.ts or app/api/chat-simple/route.ts)

Handles chat requests by:

- Processing user messages
- Searching the knowledge base for relevant information
- Generating AI responses using OpenAI
- Storing conversation history

Embeddings Generation (lib/embeddings.ts)

Manages vector embeddings for semantic search:

- Generates embeddings for company data
- Stores embeddings in the database
- Provides search functionality for relevant information

Analytics Tracking (lib/analytics.ts)

Tracks user interactions and system performance:

- Records query response times
- Logs user satisfaction ratings
- Categorizes queries for analysis

6. Database Schema

Users Table

- id: UUID (primary key)
- email: String (unique)
- created_at: Timestamp
- role: String (user, admin)

Conversations Table

- id: UUID (primary key)
- user_id: UUID (foreign key to Users)
- title: String
- created_at: Timestamp
- updated_at: Timestamp

Messages Table

- id: UUID (primary key)
- conversation_id: UUID (foreign key to Conversations)
- role: String (user, assistant, system)
- content: Text
- created_at: Timestamp

Company Data Table

- id: UUID (primary key)
- category: String
- title: String
- content: Text
- keywords: Array
- created_at: Timestamp
- updated_at: Timestamp

Embeddings Table

- id: UUID (primary key)
- company_data_id: UUID (foreign key to Company Data)
- embedding: Vector
- created_at: Timestamp

Analytics Table

- id: UUID (primary key)
- user_id: UUID (foreign key to Users)
- query: Text

- response_time_ms: Integer
- category: String
- satisfaction_rating: Integer
- created_at: Timestamp

7. Setup Instructions

Prerequisites

- Node.js 18+ and npm/yarn
- Supabase account
- OpenAI API key

Environment Variables

Create a .env.local file with the following variables:

```
# Supabase
NEXT_PUBLIC_SUPABASE_URL=your_supabase_url
NEXT_PUBLIC_SUPABASE_ANON_KEY=your_supabase_anon_key
SUPABASE_SERVICE_ROLE_KEY=your_supabase_service_role_key
SUPABASE_ANON_KEY=your_supabase_anon_key

# OpenAI
OPENAI_API_KEY=your_openai_api_key
```

Database Setup

1. Create the following tables in Supabase:

2. users

3. conversations

4. messages

5. company_data

6. embeddings

7. analytics

8. Create the match_embeddings function in Supabase SQL:

```
CREATE OR REPLACE FUNCTION match_embeddings(
  query_embedding vector,
  match_threshold float,
  match_count int
)
RETURNS TABLE (
  id uuid,
  company_data_id uuid,
  title text,
  content text,
  category text,
  similarity float
)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY
  SELECT
    cd.id,
    e.company_data_id,
    cd.title,
    cd.content,
    cd.category,
    1 - (e.embedding <=> query_embedding) as similarity
  FROM
    embeddings e
  JOIN
    company_data cd ON e.company_data_id = cd.id
  WHERE
    1 - (e.embedding <=> query_embedding) > match_threshold
  ORDER BY
    similarity DESC
  LIMIT
    match_count;
END;
$$;
```

8. Development and Production Considerations

Development Mode

In development mode, the application:

- Uses a mock user ID to bypass authentication
- Skips database operations for embeddings and knowledge base search
- Uses mock Swisscom data for AI responses

To enable full functionality in development:

1. Create a local Supabase instance or use a development project
2. Seed the database with sample company data
3. Generate embeddings for the sample data

Production Mode

For production deployment:

1. Ensure all environment variables are properly set
2. Set NODE_ENV=production to enable all database operations
3. Seed the production database with actual Swisscom company data
4. Generate embeddings for all company data
5. Set up proper authentication and user management

Performance Considerations

- The OpenAI API has rate limits that may affect response times
- Large knowledge bases may require pagination or optimization
- 3D avatar rendering may be resource-intensive on low-end devices

9. Troubleshooting

Common Issues

"Cannot read properties of undefined (reading 'call')"

This is typically a webpack/bundling issue. Try:

- Clearing the Next.js cache: `rm -rf .next`
- Reinstalling dependencies: `rm -rf node_modules && npm install`

"Could not find the function public.match_embeddings"

The SQL function is missing in the database. Execute the SQL function creation script in the Supabase SQL editor.

"Key (user_id) is not present in table 'users'"

Foreign key constraint violation. Either:

- Create the user in the users table first
- In development, modify the code to skip database operations

Raw streaming data appearing in chat

The streaming response is not being properly processed. Ensure the customer-service-avatar.tsx component is correctly handling the streaming response.

Text-to-speech not working

The Web Speech API may not be supported in all browsers. Check browser compatibility or provide a fallback.