**Agentic AI Stack for Customer Communications**

# Design Document

# Document Version 1

# Prepared for: Capgemini

# Jeff Scoggins, Ploynapas Masi, Catherine Ayeronwi, Christopher Wilder, John Foster

Date: (10/14/2025)

Department: College of Computing and Software Engineering

Course: SWE 7903: Software Engineering Capstone

Professor: Dr. Reza Meimandi Parizi

**TABLE OF CONTENTS**

1. INTRODUCTION

1.1 Purpose

1.2 Scope

1.3 Overview

2. SYSTEM OVERVIEW

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

3.2 Design Rationale

4. DETAILED DESIGN

5. DATABASE (DATA) DESIGN

6. HUMAN INTERFACE DESIGN

6.1 UI design

6.2 UX design

### INTRODUCTION

## Purpose

This Software Design Document has been written to outline and define the backend architecture and system design of the Agentic AI Communication Stack project being developed for Capgemini during the Fall 2025 CCSE Capstone Class at Kennesaw State University. Consolidating the multiple design patterns and decisions into this document allows team members, college faculty, and our corporate sponsors to find all related designs in a central location for reference and potential future refinement.

## Scope

The Agentic AI Communication Stack has five goals its intended to accomplish: Implement a LLM classifier that can categorize inbound customer messages, develop a resolution logic for common customer queries, utilize different agents to send automated emails or SMS messages for order events, and simulate the overall communication flow for vendor-customer order systems, all while having documentation of each agents workflows, and decision-making. The scope surrounding these objectives are sized to ‘prove the concept’ of Agentic AI usage. Utilizing LangGraph and Gemini to control and power the Agentic AI LLMs, allows us to iterate quickly with updates and improvements to our project, and keep things running locally for privacy and cost concerns. Langchain is used to help regulate the agent logic for outbound notifications and inbound classification between agents, as well as the basic resolution steps that each agent follows. For customer communications sent from the agents, SendGrid is being used to simulate message sends. Streamlit is being used to power the UI and CLI interactions that are showcased in section 6 of this document. All agent logic and message logs are tracked inside a SQLite database alongside demonstration data for the application to interact with.

## Overview

This document contains the reference architecture and diagrams that the team has deemed necessary to document for the Agentic AI Communication Stack, including high-level system architecture, UML class plans, SQLite database schemas, as well as UI/UX design decisions and mockups.

### SYSTEM OVERVIEW

This project aims to design a simplified, modular Agentic AI communication stack that leverages autonomous agents and LLM-powered message classification to automate vendor-customer interactions such as order confirmations, shipping updates, and basic support. The goal is to build a proof-of-concept system using open-source tools and Azure services (within the student credit budget), demonstrating effective agent orchestration and scalable communication workflows. The proof-of-concept system utilizes a SQLite database to store all customer conversation data and will be easily adapted to systems with existing storage systems. This system will allow E-commerce vendors to provide effective and easily scalable customer support with minimal overhead and staff oversight. The system will utilize proven technologies such as LangGraph and SendGrid to allow for rapid development and easy adoption by prospective clients after the proof-of-concept phase is completed.

### SYSTEM ARCHITECTURE

## Architectural Design

The architectural structure for our system has been designed to meet the client’s requirements that mandated an agentic AI system consisting of a variety of “task-specific” agents and one “supervisor” agent that handles parsing initial user input, routing to the relevant task agent to complete the customer’s request, and referring the customer to a live agent if their request is outside the scope of the various sub task agents. The responsibilities of the various task agents are as follows: The Billing, Shipping, and Order agents all retrieve relevant customer information from the system database. The Email agent handles outgoing email messaging, to be used with other task agents that send external messages. The Account agent handles customer account information retrieval and modification requests, and the Messages agent retrieves customer chat history. Finally, the Policy agent is a special agent referenced by all other agents when parsing user inputs. The Policy agent utilizes a set of policies determined by the client to best answer customer requests while remaining within the confines of the client’s specified business rules. The user interface of the system is a simple Streamlit app. The system's SQLite database contains all necessary customer information, support chat history, and email receipts.

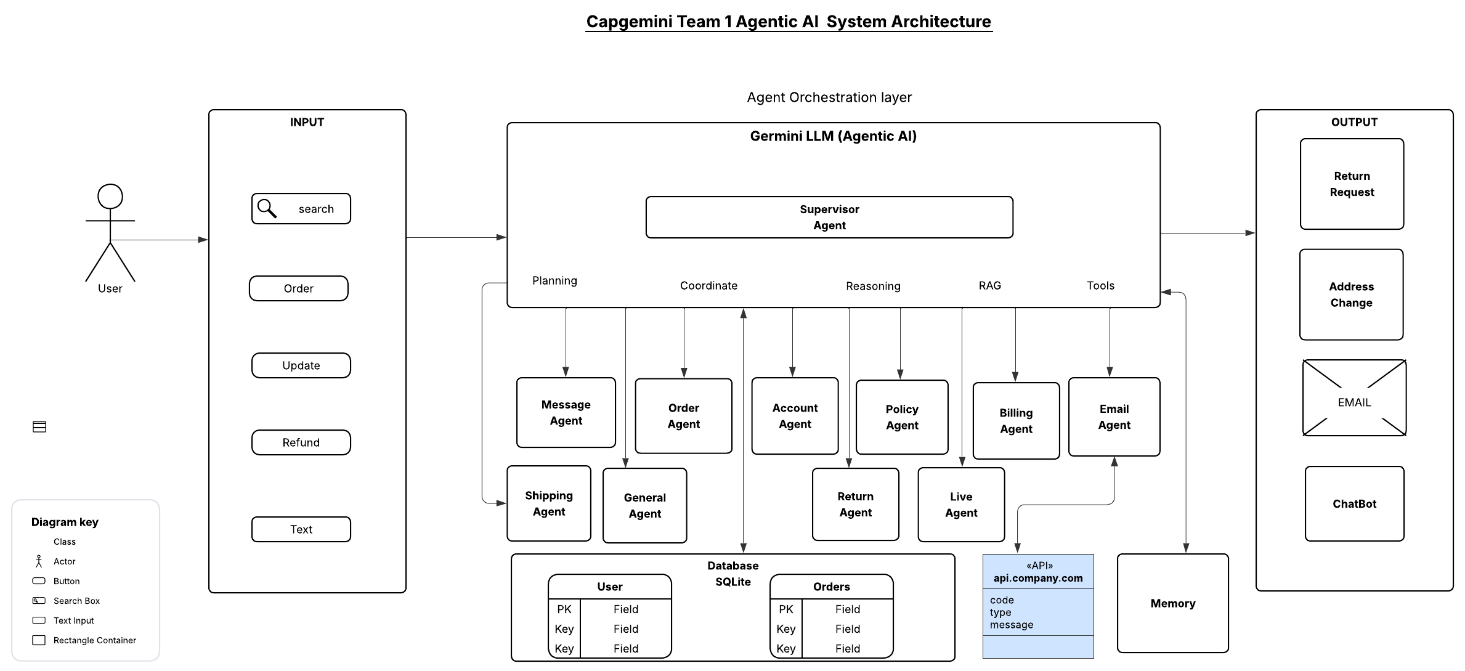


Figure 1. Systems Architecture of Team 1 Agentic AI Customer Service Solution

## Design Rationale

The Agentic AI architecture rationale is geared to build a modular, scalable, and independent system that can perceive its environment, reason, plan, and act autonomously to achieve the core objective of offering customer service support to users. The patterns considered for our project are dynamic reasoning, plan sequences, making decisions based on real-time data, reflection, and action. The trade-off was not to choose expensive agentic AI development software but rather to opt for free, low-cost, and open-source options due to the (within student credit budget) specified for this project, the fixed project timeline, project complexity, and team expertise. The core technology chosen is the LangGraph framework that helps to build agentic AI driven by LLMs and generative AI. The LLM used for our project is Gemini. Gemini LLM is an open-source AI agent, and it has superior performance and functionality in developing agentic AI. We considered Mistral LLM but realized evaluating it was slow, so we changed it and replaced it with Gemini. Streamlit is used for the UI, Python is used for the backend, and SQLite is used for the database as it is lightweight, scalable, and supports rapid data modification. The system’s technology stack was also designed to align with industry standards for agentic AI development.

### DETAILED DESIGN

### 

### Figure 2. AI Agentic Agent UML Diagram

The UML diagram represents a modular, AI agentic agent system designed to handle user interactions, order processing, messaging, and event logging in a scalable and intelligent way. The architecture is composed of several core components, each with distinct responsibilities and relationships.

**User Interaction**

The User class models the end-user of the system. Users can send messages, check orders, and request returns. These actions initiate interactions with various system components, including agents and logging mechanisms.

**Message Flow and Logging**

Messages sent by users are processed by the message agent, a specialized agent responsible for handling communication. Each message is logged via the *log event class*, which captures metadata such as timestamp, user ID, and event type. Additionally, it is stored in the *memory class* for future retrieval or context-aware processing. This flow ensures traceability and supports intelligent behavior through memory persistence.

**Order Management**

Orders placed by users are represented by the order class. Each order includes a unique ID, associated user ID, a list of items, and status tracking. Orders are routed through the *supervisor agent* to delegate tasks to appropriate *specialized agent* such as *billing agent*, *shipping agent*, or *return agent, etc.*

**Agent Hierarchy**

The system architecture employs a layered agent-based design to manage user interactions and backend processes efficiently. Each agent operates as an independent component with clearly defined responsibilities, enhancing modularity and scalability.

*- Supervisor agent* coordinator of the system. It manages task delegation, processes incoming requests, and determines which specialized agents should handle them. It also facilitates collaboration between multiple agents when a task spans multiple domains. When a request cannot be resolved by automated agents, the *Supervisor agent* escalates it to a *Live agent*.

*- Specialized agent* handles domain-specific tasks such as billing, messaging, shipping, and policy enforcement. Each agent is designed to operate independently, allowing for focused functionality and easier maintenance.

*- Live agent* represents a human fallback mechanism for handling complex or exceptional cases that automated agents cannot resolve. It is not part of the AI agent hierarchy but is connected to the *Supervisor agent* via a dashed line with an arrow, indicating a dependency for escalation. *Live agent* ensures that user issues are addressed when automation reaches its limits.

**Event Logging and Memory**

The log event class acts as a centralized logging mechanism for all agent and user activities. It is tightly integrated with user, message, and memory.

**Class relationships**

- Solid lines represent direct associations or ownership.

- Dashed lines with open arrows indicate dependencies or usage relationships.

- Directional arrows clarify the flow of control and data between classes.

### DATA (DATABASE) DESIGN

The database was designed as a SQLite database with the idea that the database needs to be lightweight, portable, and relational. The database is designed to allow entities in the Agentic AI system to become a relational system organized by a hierarchical structure. In this system, there are five tables. Customer, Order, Ordered Items, Payment, and AI Conversation logs. Each table is modeled as an entity, paired with attributes that define, and link relationships between the tables.

Table 1. Data Relation

|  |  |  |
| --- | --- | --- |
| **Tables** | **Data Relation** | **Purpose** |
| User | Primary Key: email | Uniquely identifies each customer and stores profile and login info. |
| Order | Foreign Key: order\_id -> users.email | Tracks customer orders and shipping |
| Ordered Items | Foreign Key: order\_item\_id -> orders.order\_id | Tracks individual items ordered |
| Payment | Foreign Key: payment\_id -> users.email Foreign Key: payment\_id -> orders.order\_id | Tracks financial transactions |
| AI Conversations | Foreign Key: conversation\_id -> users.email | Logs all AI communications |

Relationships enforced by foreign keys so deleting a user, cascades to their orders and conversations. As a result, a user should never be deleted from the database but simply marked inactive. This delivers normalized, queried, context-aware data structures for an AI empowered customer service representative. As a result of this design, each table captures a specific piece of information of their experience on the website and stores it in the database for the AI to process which will help to better serve the user if they have a question, or issue with the product.

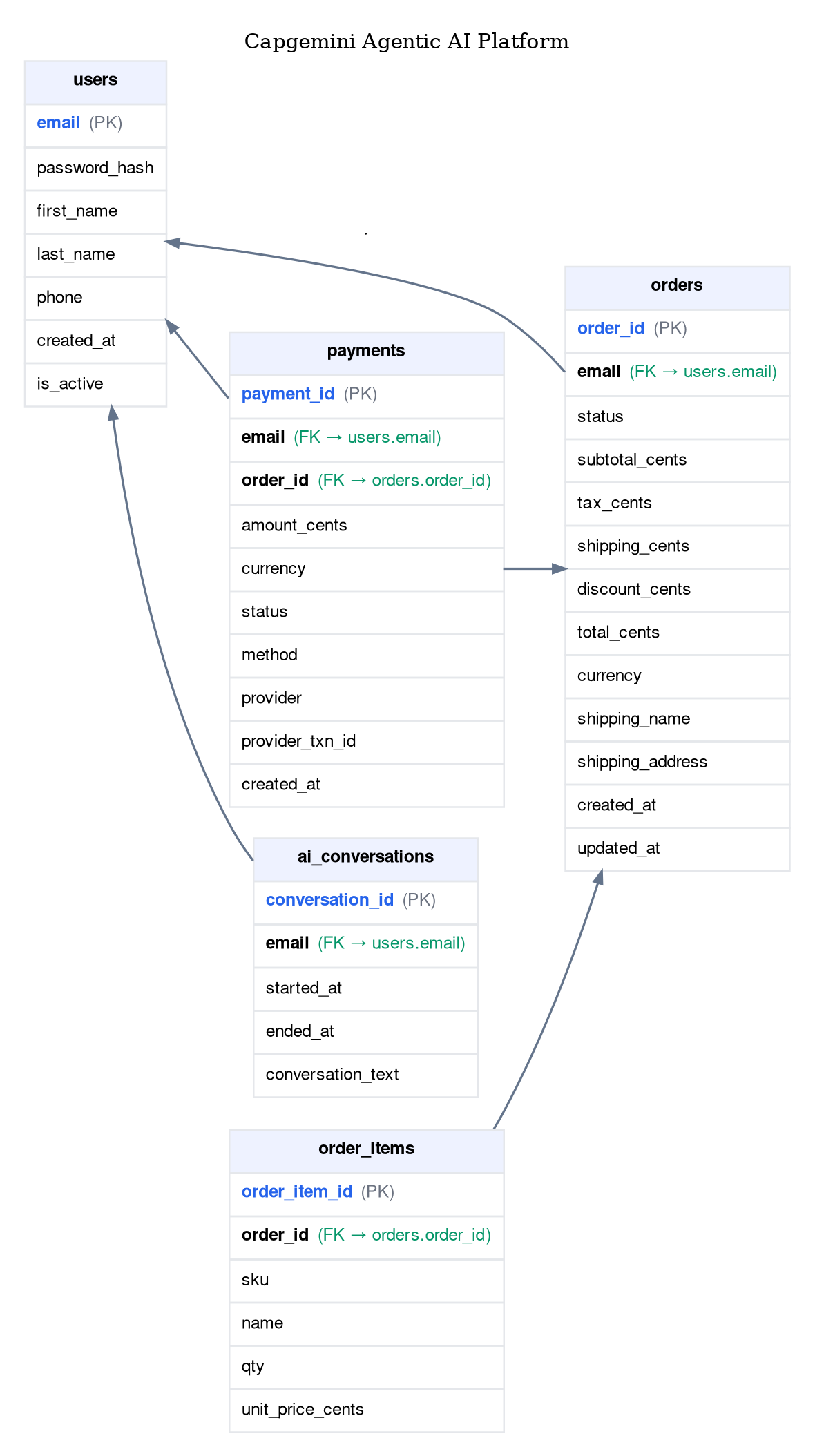


Figure 3. Database Relationship Table

### HUMAN INTERFACE DESIGN

## UI design

## 

## Figure 4. Sign-in UI Mockup Design

## 

## Figure 5. Signup UI Mockup Design

## 

## Figure 6. Agentic AI Customer Service UI Mockup

## 

## Figure 7. Agentic AI Customer Service Home Page UI Design

## 

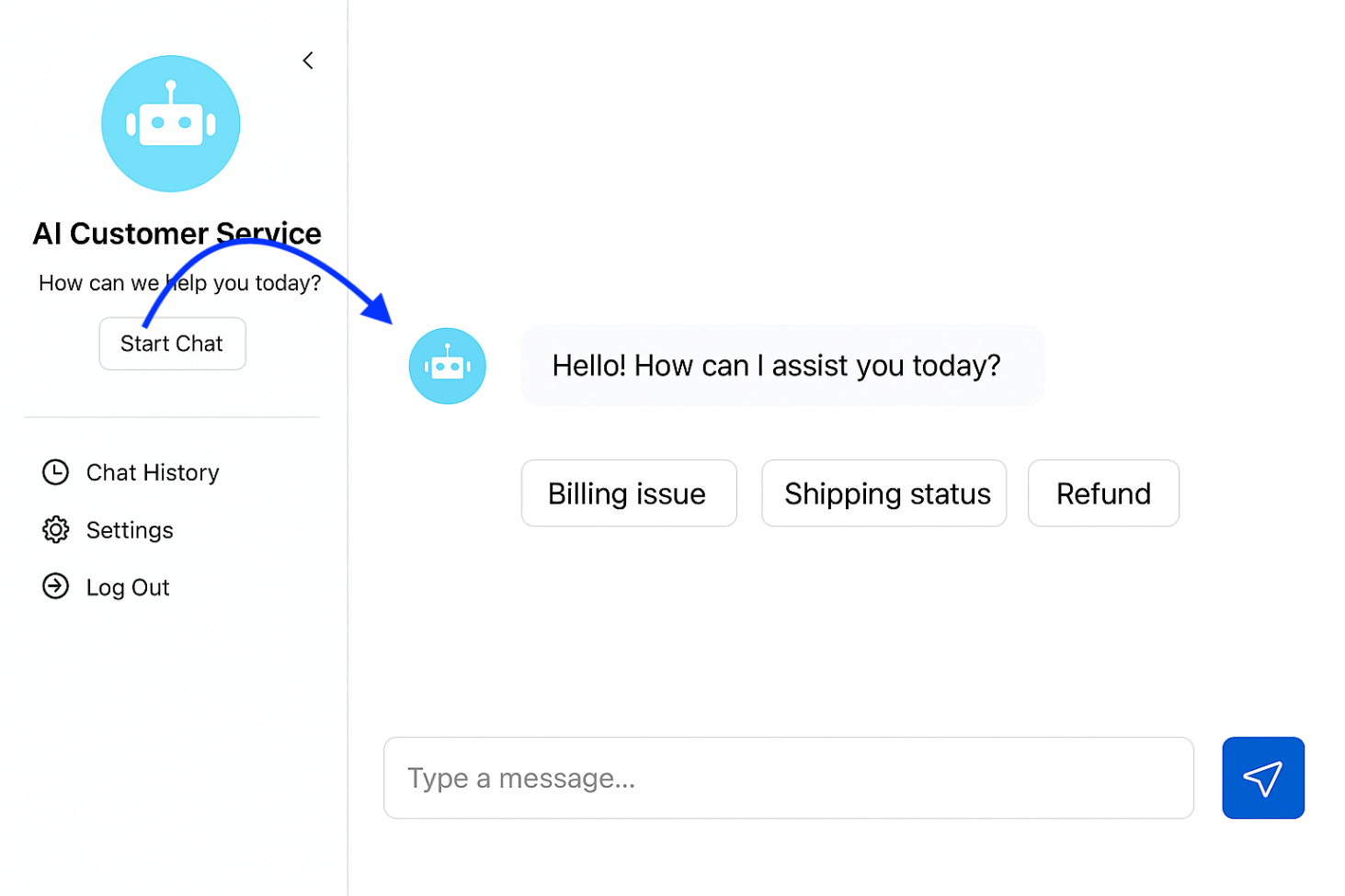
## UX design

## 

## Figure 8. Agentic AI Customer Service Home Page UX Design

## 

## Figure 9. Before Start Chat UX Design

Figure 10. Start Chat Clicked UX Design

## 

## Figure 11. Chat History UX Design

## 

## Figure 12. Settings UX Design

## 

## Figure 13. User Experience Flow Design