Software Design Description

6.1 Introduction

This document presents the architecture and design for my Travel Tailor Agent project. The project performs personalized travel itinerary generation by interpreting natural language user requests, fetching relevant data from external APIs (flights, weather, attractions), and presenting it within a conversational chat interface.

6.1.1 System Objectives

The primary objective of the Travel Tailor Agent is to provide users with an intuitive and personalized way to plan their trips from initial concept to a structured itinerary. This is achieved by offering a natural language interface to gather user travel criteria, automating the retrieval of key travel information (flight options, weather forecasts, popular local attractions), and presenting this information within a unified chat interface. I wanted to develop an app to simplify the initial travel planning steps, making relevant information easily accessible and allowing users to ask follow-up questions as if interacting with a human travel agent.

6.1.2 Hardware, Software, and Human Interfaces

Hardware Interfaces: The system relies on standard user input hardware such as keyboards and mice (or touch interfaces on mobile devices) for interaction with the web-based GUI. It also requires a standard internet connection for communication between the user's browser, the frontend server, and the backend server, and for the backend server to communicate with external APIs over the internet.

Software Interface:
Frontend Stack: The user interface is built using Next.js (a React framework) which provides serverside rendering capabilities. It is written in TypeScript for improved code maintainability and uses Tailwind CSS for styling. Shadcn UI components are integrated to provide pre-designed and responsive UI elements like buttons, input fields, and chat message containers. This frontend stack communicates with the backend via standard HTTP requests, specifically RESTful API calls.
Backend Stack: The application's core logic and API orchestration are handled by a backend service written in Python. This backend is built using the FastAPI framework for creating RESTful APIs, and served using the Uvicorn ASGI server. Pydantic is used for data validation and serialization, defining the structure of request bodies and response data exchanged between the frontend and backend, and for handling data received from external APIs. The backend is responsible for parsing user queries, making calls to external services, and formatting the responses.

Human Interface: The primary human interface is a web based graphical user interface (GUI). This interface consists of a chat window where users input their queries and view responses from the agent. It will display information such as text messages, flight information cards, weather details, and lists of local attractions. The interface is developed using Next.js in TypeScript, styled with Tailwind CSS, and utilizes shadcn UI components for interactive elements, providing a responsive and modern user experience accessible via a web browser.

External APIs: The backend interfaces with several external third-party APIs to retrieve travel information:

- Anthropic Claude 3.5 Haiku: Used as the core LLM to understand natural language queries, extract details, engage in conversational dialogue, and generate text content. The interaction is via API calls sending user prompts and receiving text responses.
- Google Flights API: Used to search for flight options based on origin, destination, and dates. The backend makes API requests with the given parameters and receives structured flight data in response.
- OpenMeteo weather API: Used to fetch weather forecasts for the specified destination and dates. The backend makes API requests with location and date parameters and receives weather data (temperature, conditions, etc.).
- Google Search API: Used to find popular attractions or general information about the destination. The backend makes search queries and receives search results, which are then processed to identify relevant attractions.

6.2 Architectural Design
I used a client server architecture for Travel Tailor Agent. The frontend serves as the client, running in the user's web browser and handles the user interface and interaction. The backend serves as the server, processes user requests, interacts with external APIs, and manages the application logic. Communication between the client and the server is handled via a RESTful API.

6.2.1 Major Software Components
Frontend User Interface (UI): The front end renders the chat interface, captures user input, displays agent responses, and handles user interactions. I used Next.js, React, TypeScript, shadcn UI components, and Tailwind CSS.
Backend API Server: The backend server is built with Python, FastAPI, and Uvicorn, and serves as the main entry point for requests from the frontend. It routes requests to appropriate internal logic, handles data validation (with Pydantic), and makes calls to other internal components and external services.
LLM Interaction Module: A component within the backend responsible for interfacing with the Anthropic Claude API. It formats user queries for the LLM, sends requests, and processes responses to extract structured data or generate conversational text.
External API Integration Modules (Flights, Weather, Search): The backend also interacts with external APIs (Google Flights, OpenMeteo, Google Search). Each module handles the specifics of constructing requests for its respective API, making the calls, and parsing the raw API responses into a consistent internal data format (using Pydantic models).
Data Processing and Consolidation Module: A component within the Backend that takes the parsed output from the LLM and the data retrieved from the various External API Integration

Modules, processes and consolidates this information, and formats it into the final response structure to be sent back to the Frontend.

6.2.2 Major Software Interactions

A user interacts with the frontend UI, typing a query into the chat input.
The frontend sends the user's query as an HTTP request to the backend API Gateway
The backend API Gateway/Orchestrator receives the request and routes the user's query to the LLM Interaction Module.
The LLM Interaction Module formats the query and sends it to the external Anthropic Claude API.
The Anthropic Claude API processes the request and returns a natural language response or structured information about the user's intent and extracted entities (destination, dates, etc.).
The LLM Interaction Module processes the LLM's response. If it needs to gather external data (flights, weather, attractions), it informs the backend. If it's a follow-up conversational query, it might generate the direct response text.
If external data is needed, the backend invokes the relevant External API Integration Modules (Flights, Weather, Search) based on the information extracted by the LLM.
Each External API Integration Module sends a request to its respective external API (Google Flights, OpenMeteo, Google Search).
The external APIs return data to their respective External API Integration Modules.
The External API Integration Modules parse the raw API responses into structured data (e.g., Pydantic models).
The backend collects the structured data from the External API Integration Modules and the processed output from the LLM Interaction Module.
The Data Processing and Consolidation Module processes and combines all the gathered data into a single, cohesive response structure intended for the user.
The backend sends the final structured response as an HTTP response back to the frontend.
The Frontend UI receives the response and updates the chat interface to display the agent's message, flight options, weather, and attractions.
This cycle repeats for each user input.

6.2.3 Architectural Design Diagrams

(As requested, specific diagram generation is omitted, but the types of diagrams that would be included here to illustrate the system architecture are listed below.)

Use Case Diagram: Illustrates the interaction between the primary actor (the User) and the system, showing key use cases like "Plan Trip", "Ask Follow-up Question", and "Receive Itinerary".
Top-Level Class Diagram: Shows the main classes within the system's logical components (e.g., ChatComponent, BackendAPIService, FlightAPIClient, WeatherAPIClient, SearchAPIClient, ItineraryDataModel) and their relationships.

Component Diagram: Depicts the physical components of the system (Frontend application, Backend service) and their dependencies on external systems (Anthropic Claude API, Google Flights API, OpenMeteo API, Google Search API).
Deployment Diagram: Shows how the system components are deployed on hardware nodes (e.g., User's Browser, Frontend Server, Backend Server, External API Servers).

6.3 CSC and CSU Descriptions
Travel Tailor Agent assists users in planning trips by providing flight info, weather, attractions, and recommendations. It follows a client server architecture with a Next.js frontend and a Python FastAPI backend.

- Frontend CSC
    - Responsible for the user interface, clientside logic, and interaction with the Backend CSC. Built with Next.js, React, and TypeScript, it provides the chat-based interface.
    - Page CSU
        - Implements main UI components and application logic, handling user input, state, and data flow between frontend parts.
    - UI Components CSU
        - Provides reusable UI elements like chat bubbles, cards, input fields, and components for displaying travel data, utilizing shadcn UI.
    - API Client CSU
        - Manages frontend-backend communication, making HTTP requests to retrieve flight, weather, and attraction data via defined API endpoints.
    - Claude Integration CSU
        - Provides functionality to interact with Claude AI specifically for frontend natural language understanding needs or direct text generation display, distinct from backend parsing.
- Backend CSC
    - Responsible for API endpoints, processing external data, and communicating with third-party services. Built with Python FastAPI, it orchestrates data retrieval and processing.
    - FastAPI Server CSU
        - Implements core server logic and API route handlers for backend services, processing incoming requests and returning responses, acting as the primary gateway.
    - External API Integration CSU
        - Manages connections to external services: SerpAPI (for flights), OpenMeteo (weather), and Claude AI (backend NLU/generation). It handles request formatting and raw response processing.
- Data Processing CSC
    - Responsible for transforming, normalizing, and enriching data across the application. Handles parsing, formatting, and data transformation for consistency and usability.

        - Travel Request Parsing CSU
            - Converts natural language travel requests into structured data using Claude AI, extracting key trip information like dates, origin, and destination.
        - City Normalization CSU
            - Handles normalization of city names and airport codes to ensure consistent representation for accurate API calls and display.
        - Weather Formatting CSU
            - Processes raw weather data, converts it into a user-friendly format for display, and generates natural language descriptions using AI or templates.
    - AI Integration CSC
        - Manages interactions with AI services, specifically Claude AI, supporting natural language processing and generation tasks.
        - Claude Client CSU
            - Provides a wrapper for the Anthropic Claude API, managing authentication, request formatting, and response parsing for backend AI interactions.
        - Question Answering CSU
            - Implements functionality leveraging Claude AI to answer general travel-related questions posed by the user, utilizing either retrieved data or the LLM's knowledge base.

6.3.1 Detailed Class Descriptions Section
TravelRequest Class
Fields:
start_date: String (YYYY-MM-DD) - Departure date.
end_date: String (YYYY-MM-DD) - Return date.
origin: String - Departure city name.
destination: String - Destination city name.
Represents the core structured travel request data used system-wide, holding essential trip information.

6.3.1.2 Message Class
Fields:
role: String ("user" or "assistant") - Message sender.
content: String (optional) - Text content.
flights: Object (optional) - Flight options data.
weather: Object (optional) - Weather data and description.
attractions: Object (optional) - Tourist attraction data.
id: String (optional) - Unique message identifier.
Represents chat interface messages, supporting text and rich data payloads for displaying flights, weather, and attractions.

6.3.1.3 Flight Class

Fields:
option: String - Flight option identifier.
price: String - Price formatted with currency.
duration: String - Flight duration.
details: String[] - Details like airline, flight number, times.
Encapsulates flight information from external APIs in a standard display format.

6.3.1.4 WeatherForecast Class
Fields:
date: String (YYYY-MM-DD) - Date of forecast.
maxTemperature: Number - Max temperature (F).
minTemperature: Number - Min temperature (F).
windSpeed: Number - Wind speed (mph).
precipitationProbability: Number - Probability of precipitation (%).
Represents daily weather forecast data for a location.

6.3.1.5 WeatherLocation Class
Fields:
latitude: Number - Latitude.
longitude: Number - Longitude.
timezone: String - Timezone identifier.
timezoneAbbreviation: String - Timezone abbreviation.
Contains geographical context for weather forecasts.

6.3.1.6 Attraction Class
Fields:
title: String - Attraction name.
reviews: Number or String - Review count/description.
rating: Number - Rating (0-5).
address: String - Physical location.
website: String - URL.
description: String - Brief description.
thumbnail: String - Image URL.
hours: String - Operating hours.
phone: String - Phone number.
place_id: String - Unique place identifier.
Represents tourist attractions with key information for travelers.

6.3.1.7 FlightRequest Class
Fields:
departure_location: String - Origin city/airport code.
arrival_location: String - Destination city/airport code.
departure_date_and_time: String (YYYY-MM-DD) - Departure date.
return_date: String (YYYY-MM-DD, optional) - Return date.

Backend structure for formatting requests sent to external flight search APIs.

6.3.1.8 WeatherRequest Class
Fields:
destination: String - City name.
startDate: String (YYYY-MM-DD) - Start date.
endDate: String (YYYY-MM-DD) - End date.
Backend structure for formatting requests sent to weather forecast APIs.

6.3.1.9 AttractionRequest Class
Fields:
city: String - City name.
Backend structure for formatting requests sent to attraction search APIs.

6.3.2 Detailed Interface Descriptions Section
6.3.2.1 Frontend-Backend Communication Interface
Uses HTTP REST API endpoints via FastAPI. Key endpoints handle parsing (/api/parse), updates (/api/update), data retrieval for flights (/api/flight), weather (/api/weather), attractions (/api/attractions), and general queries (/api/general_query). Communication uses JSON payloads (validated by Pydantic) with standard HTTP status codes for error handling.

6.3.2.2 Backend External API Interface
Facilitates backend communication with SerpAPI (Google Flights), OpenMeteo (Weather), and Claude AI. Backend modules handle API-specific request formatting, authentication, calls, and parsing raw responses into consistent internal formats.

6.3.2.3 User-Application Interface
The chat-based GUI allows user interaction via natural language input. Displays include standard text messages and specialized components for flight, weather, and attraction data. Supports "Gathering" mode for collecting details and "General QA" mode for answering questions.

6.3.3 Detailed Data Structure Descriptions Section
6.3.3.1 Message Array Structure
An array of Message objects representing the conversation history in the chat interface, acting as the main state container.

6.3.3.2 City Name Variations Mapping
A dictionary mapping common city name variations to standardized forms (e.g., nyc: "new york") for consistent processing.

6.3.3.3 Airport Code Mapping
A dictionary mapping city names to potential airport codes (e.g., "new york": ["JFK", "LGA", "EWR"]) for use in flight API requests.

6.3.3.4 Flight Segments Structure

A nested structure describing individual legs of a flight itinerary, including departure/arrival airports and times (departure: { airport: string; dateTime: string; formatted: string }; arrival: ...).

6.3.3.5 Weather Forecast Array

An array of WeatherForecast objects providing daily weather predictions, typically displayed in a scrollable interface.

6.4 Database Design and Description

Based on the current scope of the Travel Tailor Agent project, which focuses on real-time data retrieval and conversational interaction within a single session, a persistent database for storing user data, itineraries, or extensive conversation history is not currently implemented or required.