CODE and TESTING REPORT

for

Learnix

Prepared by Vincent Le, Bayasgalan Battogtokh, Devin Irby, Miguel Lima, Wyatt Stohr, Xander Thompson, Boscoe Lindholm

Project 1: AcademicGPT

March 12, 2024

Contents

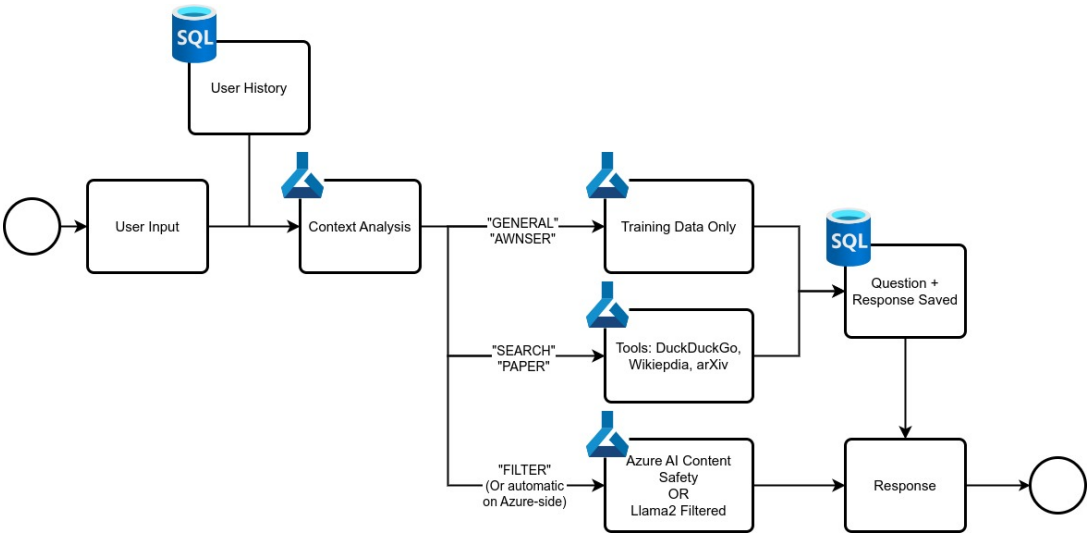

Figure 1: Conversation Code-Flow

1 Introduction

This code and testing report provides a comprehensive overview of a dynamic application, detailing both its front-end and back-end components. The application is built using a robust tech stack, featuring TypeScript, React, JavaScript, and Cascading Style Sheets for the front-end, while Flask, MySQL, and other technologies power the back end. The code structure is organized into clear sections, with each subsection unraveling the intricacies of the codebase.

Section 1 delves into the code architecture, highlighting the key files and components that constitute the front-end and back-end of the application. The front-end is constructed using React and TypeScript, offering insights into the main files such as `_app.tsx`, `WelcomePage.tsx`, `SignUpPage.tsx`, `MainPage.tsx`, `FullscreenSearch.tsx`, `Layout.tsx`, `Sidebar.tsx`, and `Topbar.tsx`. These files manage various functionalities, including user authentication, registration, and navigation elements, providing a structured and modular approach to the application's design.

Section 2 shifts focus to the testing aspect of the application, detailing test cases for critical functionalities. Each test case, such as the login screen, sign-up screen, and main screen, is thoroughly described, including associated requirements and the current status of testing. The notes highlight any pending issues due to connectivity challenges with the server, underlining the importance of validating these functionalities in subsequent testing phases.

1.1 Front-End Code

1.1.1 _app.txt

The front-end of the application was written in Typescript, uses React framework, JavaScript, and Cascading Style Sheet files to designate the style of each page. The Capacitor software is used to migrate the app to an android application. The main file, _app.tsx, is the driver of the application and is used to run the app. This file references the other pages such as the Main page, Results page, Sign up page, and Welcome page. There are some other minor pages like the Fullscreen Search page, Sidebar page, Topbar page and the Layout page.

```
import "@/styles/globals.css";
import '@/styles/topbar.css';
import '@/styles/sidebar.css';
import '@/styles/fullscreenSearch.css';
import '@/styles/welcomePage.css';
import '@/styles/signUpPage.css';
import { useState } from "react";
import type { AppProps } from "next/app";
import Cookies from 'js-cookie';
import WelcomePage from './welcomePage';
```

```
import SignUpPage from './signUpPage';
import { useRouter } from 'next/router';
import MainPage from './mainPage';
import Layout from "./comp/layout";
import ResultsPage from "./resultsPage";
```

This code is part of a React web application using Next.js. It imports CSS files for styling, external libraries like `js-cookie` for handling cookies, and various components representing different pages or sections of the application.

```
export interface User {
    email: string | undefined;
    username: string | undefined;
    id: string | undefined;
}

export interface SearchQuery {
    prompt: string;
    result: string | null;
    userId: string;
    queryId: string;
}
```

These interfaces provide a clear and typed structure for representing user and search query data within the application.

```
export const backendUrl = 'http://server.com/';
export var searchHistory: SearchQuery[] = [];
export var currentSearchQuery: SearchQuery | null = null;
export const currentUser: User = {
    email: Cookies.get('email'),
    username: Cookies.get('username'),
    id: Cookies.get('id')
};
export var searching = false;
```

These are search queries that should be displayed on the page.

```
    const [prompt, setDisplayedPrompt] = useState('');
    const [result, setDisplayedResult] = useState('');
```

Load the most recent search query

```
    if (!searchHistory.length) {
        var history = Cookies.get('searchHistory');
        searchHistory = JSON.parse(history || '[]');
    }
```

Check if the user is logged in

```
    var goToWelcomeUpPage = currentUser.email === undefined ||
        currentUser.username === undefined ||
        currentUser.id === undefined;
```

These two functions are for the results page by being passing to the layout, which displays the results. The first function is called when there is a prompt but no current search query, and the second shows the results of a previous search query.

While the Layout has functions that are called when a user makes a search or loads a search, these two functions are passed into the layout and called in there since these functions utilize the useState() hook, which re-renders the components using this hook. These functions are necessary for re-rendering the results page when the current search query has been updated.

```
const fetchResults = async (p: string) => {
    searching = true;
    // Set the prompt and the loading message
    setDisplayedPrompt(p);
    setDisplayedResult('');

    // Fetch the results from the backend
    console.log('Searching...');
    const response = await postToLlm(p).catch((error) => {
        console.error('Error:', error);
        searching = false;
        setDisplayedResult("I'm sorry, something went wrong. Please try again.");
    }).then((response) => {
        console.log('Finished search.');
        // Push the search query to the search history
        var query: SearchQuery = {
            prompt: p,
            result: response,
            userId: Cookies.get('id') || '',
            queryId: Date.now().toString()
        };
        pushSearchHistory(query);
        setDisplayedResult(response);
        searching = false;
    });
}
const showResults = (i: number) => {
    var query: SearchQuery | null = makeLatestSearchQuery(i);
    if (query) {
        setDisplayedPrompt(query.prompt);
        setDisplayedResult(query.result || '');
    }
}
```

This code conditionally renders different pages within a web application based on the current route (`router.pathname`) and certain conditions. If the route is `/signUpPage`, it renders the `/signUpPage` within a Layout component without navigation. If the variable `goToWelcomeUpPage` is true, it renders the `WelcomePage` similarly. If the route includes '`resultsPage`' and either the application is in a searching state or there is a current search query, it renders the `ResultsPage` within a Layout component with navigation. Otherwise, it defaults to rendering the `MainPage` within a Layout component with navigation.

```
// Sign up page
if (router.pathname === '/signUpPage') {
    return (
        <Layout navigation={false} showResults={(i: number) => showResults(i)}
            fetchResults={(p: string) => fetchResults(p)}>
            <SignUpPage />
        </Layout>
    );
}

// Welcome page
if (goToWelcomeUpPage) {
    return (
        <Layout navigation={false} showResults={(i: number) => showResults(i)}
            fetchResults={(p: string) => fetchResults(p)}>
            <WelcomePage />
        </Layout>
    );
}
if (router.pathname.indexOf('/resultsPage') != -1 && (searching || currentSearchQuery)) {
    // const prompt = router.pathname.split('?prompt=')[1];
```

```
        return (
            <Layout navigation={true} showResults={(i: number) => showResults(i)}
                fetchResults={(p: string) => fetchResults(p)}>
                <ResultsPage prompt={prompt} result={result}/>
            </Layout>
        );
    }

    return (
        <Layout navigation={true} showResults={(i: number) => showResults(i)} fetchResults={(p:
            string) => fetchResults(p)}>
            <MainPage />
        </Layout>
    );
}
```

Sends a prompt to the LLM and returns the response.

```
async function postToLlm(p: string) {
    let data = {
        question: p
    };
    const response = await fetch(backendUrl + 'api/llama/conversation', {
        method: 'POST',
        headers: {'Content-Type': 'application/json'},
        body: JSON.stringify(data),
    });
    if (!response.ok) {
        throw new Error('Network response was not ok');
    }
    const json = await response.json();
    return json.response;
}
```

Pushes the search query to the search history and the oldest search query gets removed if the history is full.

```
function pushSearchHistory(query: SearchQuery) {
    if (searchHistory.length >= 5) {
        searchHistory.shift();
    }
    searchHistory.push(query);
    currentSearchQuery = query;
    console.log(query);

    Cookies.set('searchHistory', JSON.stringify(searchHistory));
}
```

Brings the indexed query to the top of the search history

```
function makeLatestSearchQuery(i: number) {
    if (i >= 0 && i < searchHistory.length) {
        currentSearchQuery = searchHistory[i];
        var historyStart = searchHistory.slice(0, i);
        var historyEnd = searchHistory.slice(i + 1, searchHistory.length);
        searchHistory = historyStart.concat(historyEnd);
        searchHistory.push(currentSearchQuery);
        console.log(currentSearchQuery);
        Cookies.set('searchHistory', JSON.stringify(searchHistory));
        return currentSearchQuery;
    } else {
        return null;
```

```
    }
}
```

1.1.2 WelcomePage.tsx

`WelcomePage.tsx` has the authentication system with a `loginUser` function responsible for handling user login by sending credentials to a backend API. The `WelcomePage` component manages the user interface for the login page, interacting with the `loginUser` function and updating the UI based on the authentication response.

The following code imports all of the necessary modules and dependencies to construct a React component, including backend URL for API requests.

```
import React, { useEffect, useState } from 'react';
import Link from 'next/link';
import Cookies from 'js-cookie';
import { useRouter } from 'next/router';
import { backendUrl } from './_app';
```

`loginUser` function manages responses, sets browser cookies after a successful login, and transmits user credentials to the backend API to complete the authentication process.

```
async function loginUser(credentials: {email: string, password: string} ) {
    const response = await fetch(backendUrl + 'api/users/login', {
        method: 'POST',
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify(credentials),
    });
    if (response.status === 401) {
        console.log('Invalid credentials');
        return response.status;
    }
    if (!response.ok) {
        console.log('Server error');
        return response.status;
    }
    const data = await response.json();
    console.log(data);
    Cookies.set('email', data.user.email);
    Cookies.set('username', data.user.username);
    Cookies.set('id', data.user.id);
    Cookies.set('timestamp', Date.now().toString());
    return response.status;
}
```

`WelcomePage` handles the submission event, but mostly works with previous function (`loginUser`) to handle response and navigate them.

```
function WelcomePage() {
    const router = useRouter();
    const [formData, setFormData] = useState({
        email: '',
        password: ''
    });
    const [errorMessage, setErrorMessage] = useState('');
    const [title, setTitle] = useState('Welcome!');
    const [inactive, setInactive] = useState(false);

    const handleChange = (event: any) => {
        const { name, value } = event.target;
        setFormData(prevState => ({
            ...prevState,
            [name]: value
        }));
```

```
    };

    const onSubmitClick = async (event: any) => {
        event.preventDefault();
        if (formData.email === '' || formData.password === '') {
            setErrorMessage('Please fill in all fields');
            return;
        }

        setTitle('Logging in...');
        setErrorMessage('');
        setInactive(true);
        let response = await loginUser(formData);
        console.log(response);
        if (response == 200) {
            router.push('/');
            window.location.reload();
        } else if (response == 401) {
            setErrorMessage('Invalid credentials');
        } else {
            setErrorMessage(response.toString());
        }
        setTitle('Welcome!');
        setInactive(false);
    }
```

1.1.3 signUpPage.tsx

Imports all of the necessary modules and dependencies to construct a React component, including axios for HTTP requests.

```
'use client'
import React, { useState, Component, useEffect } from 'react';
import Link from 'next/link';
import { useRouter } from 'next/router';
import WelcomePage from './welcomePage';
import axios from 'axios';
import Cookies from 'js-cookie';
import { backendUrl } from './_app';
```

`SignUpPage` manages local state for form data (username, email, and password, toggles password visibility).

```
function SignUpPage() {
    const route = useRouter()
    const [showPassword, setShowPassword] = useState(false);

    const [data, setData] = useState({
        username: "",
        email: "",
        password: ""
    });

    const handlePasswordVisibility = () => {
        setShowPassword(!showPassword);
    };


    const handleChange = (e: any) => {
        const value = e.target.value;
        setData({
            ...data,
            [e.target.name]: value
```

```
        });
    };
```

`handleSubmit` function sends user data to the backend for registration, handles any possible errors, and browser cookies containing user data.

```
    const handleSubmit = (e: any) => {
        e.preventDefault();
        const userData = {
            username: data.email,
            email: data.email,
            password: data.password
        };
        axios.post(backendUrl + "api/users/register", userData).then((response) => {
            console.log(response.status, response.data.token);
        })
        .catch((error) => {
            if (error.response) {
                console.log(error.response);
                console.log("server responded");
            } else if (error.request) {
                console.log("network error");
            } else {
                console.log(error)
            }
        });
        Cookies.set('email', data.email);
        Cookies.set('username', data.username);
        Cookies.set('id', data.email);
        window.location.reload();
    };
```

Importation Code Allows importation and usage in additional application sections.

```
export default SignUpPage;
```

1.1.4 mainPage.tsx

defines a React functional component named `MainPage` that is responsible for rendering a welcome message

```
// MainPage.tsx
import React, { useState } from 'react';
import { currentSearchQuery, searchHistory } from './_app';

function MainPage() {
    var hasSearchHistory: boolean = searchHistory.length > 0;
    return (
        <div className="main-page">
            <div className="main-welcome">
                <p id="main-greeting">Welcome!</p>
                {!hasSearchHistory && <p id="main-sub-greeting">Make the first step!</p>}
                {hasSearchHistory && <p id="main-sub-greeting">Keep at it!</p>}
            </div>
        </div>
    )

export default MainPage;
```

1.1.5 layout.tsx

The file for the Layout component. Using this component will add the top navigation bar and side bar menu that are ever present throughout the app.

```
'use client'
import { useRouter } from 'next/router';
import { ReactNode, useEffect, useState } from 'react';
import TopBar from "./topbar";
import SideBar from "./sidebar";
import FullScreenSearch from "./fullscreenSearch";
import Cookies from 'js-cookie';
```

This interface specifies the expected properties for the Layout component,

```
interface LayoutProps {
    navigation: boolean
    children: ReactNode;
    showResults: (i: number) => void;
    fetchResults: (p: string) => void;
}
```

Controls the navigation elements toggling, the display and fetching of search results. The page layout encompasses key elements such as navigation, determining the visibility of the top bar and sidebar. It includes the `children` section, constituting the page's main content. Additionally, the functionality involves `showResults`, a function triggered upon clicking a sidebar search query, and `fetchResults`, a function to be invoked when users initiate a search.

```
const Layout: React.FC<LayoutProps> = ({ navigation, children, showResults, fetchResults }) =>
    {
```

The logic of the system is structured as follows: if there's no search query, the welcome page is shown. However, if a search query exists, the display varies based on the app's current state. If the app is awaiting search results, the prompt is shown with a loading display. Once the search results are ready, they are displayed on the page.

```
    const router = useRouter();

    const [isSidebarOpen, setIsSidebarOpen] = useState(false);
    const [isSearchOpen, setIsSearchOpen] = useState(false);

    // Sidebar overlay behavior
    const toggleSidebar = () => {
      setIsSidebarOpen(!isSidebarOpen);
    };

    const sidebarClose = () => {
      setIsSidebarOpen(false);
    };

    const sidebarSearchQuery = (x: number) => {
        showResults(x);
        sidebarClose();
        router.push('/resultsPage');
    }

    const sidebarHome = () => {
        sidebarClose();
        router.push('/');
    }
```

Additionally, `layout.tst` manages the ability to log out by clearing user cookies and refreshing the page.

```
  const sidebarLogout = () => {
       // TODO: Remove the router.push. The page should be reloaded with the layout
       //  redirecting to the welcomePage if the user is not logged in
       // router.push('/welcomePage');
       // localStorage.setItem('loggedIn', 'false');
       // Cookies.remove('id');
       // Cookies.remove('username');
       Cookies.remove('email');
       Cookies.remove('username');
       Cookies.remove('id');
       Cookies.remove('timestamp');
       window.location.reload();
       // router.push('/welcomePage');
   }

   // Search overlay behavior
   const toggleSearch = () => {
       setIsSearchOpen(!isSearchOpen);
   };

   const closeSearch = () => {
       setIsSearchOpen(false);
   };

   const doSearch = (p: string) => {
       closeSearch();
       fetchResults(p);
       router.push('/resultsPage');
   }
```

This section of code regulates the behavior of the search overlay by providing features to toggle, close and search.

```
   return (
       <div className="App">
           <TopBar showButtons={navigation} onMenuClick={toggleSidebar}
               onSearchClick={toggleSearch} />
           {navigation && <SideBar isOpen={isSidebarOpen}
               onClose={sidebarClose}
               onQueryClick={(x: number) => sidebarSearchQuery(x)}
               onHomeClick={sidebarHome}
               onLogoutClick={sidebarLogout}/>}
           {navigation && <FullScreenSearch isOpen={isSearchOpen} onClose={closeSearch}
               onSearch={doSearch} />}
           {children}
       </div>
   );
};

export default Layout;
```

1.1.6 sidebar.tsx

The component for the sidebar, used for easy access to the search history, logout, and home functionalities.

```
'use client'
import React from 'react';
import { searchHistory } from '../_app';

interface SidebarProps {
  isOpen: boolean;
  onClose: () => void;
```

```
  onQueryClick: (x: number) => void;
  onLogoutClick: () => void;
  onHomeClick: () => void;
}

const SideBar: React.FC<SidebarProps> = ({ isOpen, onClose, onQueryClick, onHomeClick,
    onLogoutClick }) => {
    // This is so the buttons is sorted from up to down, latest to oldest
    var history = searchHistory.slice().reverse();
    var historyInd: number[] = [];
    history.forEach((query, index) => {
        historyInd.push(index);
    });
    historyInd = historyInd.slice().reverse();

    // Set each button to a number. That number indexes the search history
    var dom: JSX.Element[] = [];
    history.forEach((query, index) => {
        dom.push(<button className="search-query-btn" key={historyInd[index]} onClick={() =>
            onQueryClick(historyInd[index])}><i className="fi
            fi-br-search"></i><text>{query.prompt}</text></button>);
    });

    return (
        <>
            <div className={`sidebar-overlay ${isOpen ? 'open' : ''}`} onClick={onClose}></div>
            <div className={`sidebar ${isOpen ? 'open' : ''}`}>
                <div className="top-btns">
                    <button className="close-btn" onClick={onClose}>
                        <i className="fi fi-rr-menu-burger"></i><text>LEARNIX</text>
                    </button>
                    {dom}
                </div>
                <div className="bottom-btns">
                    <button onClick={onLogoutClick}><i className="fi
                        fi-rr-exit"></i><text>Logout</text></button>
                    <button onClick={onHomeClick}><i className="fi
                        fi-rr-home"></i><text>Home</text></button>
                </div>
            </div>
        </>
    );
};

export default SideBar;
```

1.1.7 topbar.tsx

Visualization for the top bar that has search and menu button.

```
'use client'
import React from 'react';
import FullScreenSearch from './fullscreenSearch';

interface TopBarProps {
  showButtons: boolean;
  onMenuClick: () => void; // Add onMenuClick prop
  onSearchClick: () => void; // Add onSearchClick prop
}

// TODO:
//  - Replace placeholder text with actual logo
const TopBar: React.FC<TopBarProps> = ({showButtons, onMenuClick, onSearchClick}) => {
```

```
    return (
      <nav>
        <div>
          {showButtons && <button id="menu-btn" onClick={onMenuClick}>
            <i className="fi fi-rr-menu-burger"></i>
          </button>}
          {!showButtons && <div></div>}
          <p id="logo">LEARNIX</p>
          {showButtons && <button id="search-btn" onClick={onSearchClick}>
            <i className="fi fi-br-search"></i>
          </button>}
          {!showButtons && <div></div>}
        </div>
      </nav>
    );
  }

  export default TopBar;
```

1.2 Back-End Code - Database

The back end of the application used _, Flask, and MySQL. The Flask API sets up a server for the client to connect and interact with the model as well as fetch the stored data from previous interactions. The database is hosted on Planet Scale and is connected to through the Flask API via Sessions supplied from the SQL alchemy library.

1.2.1 Models

The models represent the data structure being stored or retrieved from the database. It makes for easy syntax while working with SQL alchemy and allows for the developer to avoid handcrafting SQL statements.

Query This code shows how the database table for the query entities is set up. The query has a surrogate primary key "id", a foreign key "user_id", and other data "question, response, and date." There is additionally a _repr_ function which will print out the data structure in the specified format and a to_dict function to help with parsing the data into a JSON format.

```
from flask import Blueprint, request, jsonify
import service.query as query_service

query_bp = Blueprint('query_bp', __name__)

@query_bp.route('/<int:user_id>', methods=['GET'])
def get_by_user_id(user_id):
    queries = query_service.get_by_user_id(user_id)
    return jsonify([query.to_dict() for query in queries])
```

This section gets recent queries, used in llama_complete.

```
@query_bp.route('/recent/<int:user_id>', methods=['GET'])
def get_recent(user_id):
    count = int(request.args.get('count', 5))
    offset = int(request.args.get('offset', 0))
    queries = query_service.get_recent(user_id, count, offset)
    print(queries)
    return jsonify([query.to_dict() for query in queries])
```

Adds into database, expects `user_id`, `question`, and `response` as a json object

```
@query_bp.route('/<int:user_id>', methods=['POST'])
def create(user_id):
```

```
    data = request.get_json()
    if data is None:
        return jsonify({'error': 'Invalid request, data is missing'}), 400
    question = data.get('question')
    response = data.get('response')
    success, message = query_service.create(user_id, question=question, response=response)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 400


@query_bp.route('/<int:query_id>', methods=['DELETE'])
def delete(query_id):
    success, message = query_service.delete_by_id(query_id)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 400
```

User This code shows how the database table for the user entities is set up. The query has a surrogate primary key "id" and other data "username, email, and password." There is additionally a _repr_ function which will print out the data structure in the specified format and a to_dict and to_dict_no_password function to help with parsing the data into a JSON format.

```
from flask import Blueprint, request, jsonify
import service.user as user_service

user_bp = Blueprint('user_bp', __name__)


@user_bp.route('/', methods=['GET'])
def get_all():
    users = user_service.get_all()
    return jsonify([user.to_dict() for user in users])


@user_bp.route('/<int:user_id>', methods=['GET'])
def get_by_id(user_id):
    user = user_service.get_by_id(user_id)
    if user:
        return jsonify(user.to_dict()), 200
    else:
        return jsonify({'error': 'User not found.'}), 404


@user_bp.route('/login', methods=['POST'])
def login():
    data = request.json
    success, user, message = user_service.login(data['email'], data['password'])
    if success:
        return jsonify({
            'message': message,
            'user': user.to_dict_no_password()
        }), 200
    else:
        return jsonify({
            'error': message
        }), 401


@user_bp.route('/<int:user_id>', methods=['DELETE'])
```

```
def delete_user(user_id):
    success, message = user_service.delete_by_id(user_id)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 404


@user_bp.route('/<int:user_id>', methods=['PUT'])
def update_user(user_id):
    data = request.json
    success, message = user_service.update(user_id, **data)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 400


@user_bp.route('/register', methods=['POST'])
def register():
    data = request.json
    username = data['username']
    password = data['password']
    email = data['email']
    success, message = user_service.register(username, email, password)
    if success:
        return jsonify({'message': message}), 201
    else:
        return jsonify({'error': message}), 400
```

1.2.2 Services

Services is the layer between the application's interaction with the user and the application's calls to the database. This layer allows for the fundamental business logic to be applied to any incoming request by the client so that errors are filtered out before being inserted into the database.

Query This code shows the provided services that a client can enact on the query tuples. Additionally, each message will return a Boolean value and a message to inform the client of success or errors that occur.

```
import logging
from datetime import date
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy import create_engine, desc
from sqlalchemy.orm import sessionmaker
from config import Config
from models.query import Query

engine = create_engine(
    Config.CONNECTION_TO_DATABASE,
    connect_args={
        "ssl": {
            "ssl_ca": "etc/ssl/cert.pem"
        }
    }
)

Session = sessionmaker(bind=engine)


def get_by_user_id(user_id):
    session = Session()
```

```
    queries = session.query(Query).filter_by(user_id=user_id).all()
    session.close()
    return queries


def get_recent(user_id, count, offset):
    session = Session()
    queries =
        session.query(Query).filter_by(user_id=user_id).order_by(desc(Query.id)).limit(count).offset(offset).al
    session.close()
    return queries


def create(user_id, question, response):
    session = Session()
    try:
        new_query = Query(user_id=user_id, question=question, response=response,
            date=date.today())
        session.add(new_query)
        session.commit()
        return True, "Query successfully created."
    except SQLAlchemyError as e:
        session.rollback()
        return False, f"Error creating query: {str(e)}"
    finally:
        session.close()


def delete_by_id(query_id):
    with Session() as session: # Use context manager for the session
        try:
            # Attempt to retrieve the query first to check existence
            query = session.query(Query).filter_by(id=query_id).one_or_none()
            if query is not None:
                return False, "Query not found."

            # Delete the found Query
            session.delete(query)
            session.commit()
            return True, "Query successfully deleted."
        except SQLAlchemyError as e:
            session.rollback()
            logging.exception("Failed to delete query: %s", e) # Log the exception
            return False, f"An error occurred: {e}"
```

User This code shows the provided services that a client can enact on the user tuples. Additionally, each message will return a Boolean value and a message to inform the client of success or errors that occur.

```
from datetime import date
from sqlalchemy import create_engine, or_
from sqlalchemy.orm import sessionmaker
from config import Config
from models.user import User
from models.query import Query
from sqlalchemy.exc import SQLAlchemyError
import re
import bcrypt
import logging

engine = create_engine(
    Config.CONNECTION_TO_DATABASE,
```

```
    connect_args={
        "ssl": {
            "ssl_ca": "etc/ssl/cert.pem"
        }
    }
)

Session = sessionmaker(bind=engine)


def get_all():
    session = Session()
    users = session.query(User).all()
    session.close()
    return users


def get_by_id(user_id):
    session = Session()
    user = session.query(User).filter(User.id == user_id).first()
    session.close()
    return user


def get_by_identifier(identifier):
    session = Session()
    user = session.query(User).filter(or_(User.username == identifier, User.email ==
        identifier)).first()
    session.close()
    return user


def login(identifier, password):
    session = Session()

    # Retrieve the user by username
    user = session.query(User).filter(or_(User.username == identifier, User.email ==
        identifier)).first()
    if not user:
        session.close()
        return False, None, "Invalid username or password."

    # Verify the password
    if bcrypt.checkpw(password.encode('utf-8'), user.password.encode('utf-8')):
        session.close()
        return True, user, "Login successful."
    else:
        session.close()
        return False, None, "Invalid username or password."


def register(username, email, password):
    session = Session()

    if not username or not email or not password:
        return False, "Missing required fields."

    # Check if username or email is already taken
    existing_user = session.query(User).filter(User.username == username).first()
    if existing_user:
        session.close()
        return False, "Username is taken."
```

```
    existing_email = session.query(User).filter(User.email == email).first()
    if existing_email:
        session.close()
        return False, "Email is taken."

    # Check if username and email length is within limits
    if len(username) > 25:
        session.close()
        return False, "Username must be at most 25 characters."
    if len(email) > 100:
        session.close()
        return False, "Email must be at most 100 characters."

    # Check if the password is strong
    strong, message = strong_password(password)
    if not strong:
        session.close()
        return False, message

    # Hash the password
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')

    # Create a new user with the hashed password
    new_user = User(username=username, password=hashed_password, email=email)
    session.add(new_user)
    session.commit()
    session.close()

    return True, "User registered successfully."


def delete_by_id(user_id):
    with Session() as session:
        try:
            # Attempt to retrieve the user first to check existence
            user = session.query(User).filter(User.id == user_id).one_or_none()
            if user is None:
                return False, "User not found."

            # Delete the user's dependent data
            session.query(Query).filter(Query.user_id ==
                user_id).delete(synchronize_session=False)

            # Delete the user
            session.delete(user)
            session.commit()
            return True, "User deleted successfully."

        except SQLAlchemyError as e:
            # Rollback in case of any SQLAlchemy errors
            session.rollback()
            logging.exception("Failed to delete user: %s", e) # Log the exception
            return False, f"An error occurred: {e}"


def update(user_id, **kwargs):
    with Session() as session: # Use context management for the session
        try:
            user = session.query(User).filter(User.id == user_id).one_or_none()
            if user is None:
                return False, "User not found."

            # Define a list of fields that are allowed to be updated
```

```
            allowed_updates = ['username', 'email', 'password'] # Add other fields as necessary

            for key, value in kwargs.items():
                if key in allowed_updates and hasattr(user, key):
                    if key == 'password': # Hash the new password before storing
                        value = bcrypt.hashpw(value.encode('utf-8'),
                            bcrypt.gensalt()).decode('utf-8')
                    setattr(user, key, value)

            session.commit()
            return True, "User successfully updated."
        except SQLAlchemyError as e:
            session.rollback() # Rollback in case of error
            logging.exception("Error updating user: %s", e) # Log the exception for debugging
            return False, f"Error updating user: {str(e)}"


def strong_password(password):
    """Check if the password is strong."""
    if len(password) < 8:
        return False, "Password must be at least 8 characters long."
    if not re.search("[a-z]", password):
        return False, "Password must contain at least one lowercase letter."
    if not re.search("[A-Z]", password):
        return False, "Password must contain at least one uppercase letter."
    if not re.search("[0-9]", password):
        return False, "Password must contain at least one digit."
    if not re.search("[!@#$%^&*(),.?\":{}|<>]", password):
        return False, "Password must contain at least one special character."
    return True, ""
```

1.2.3 Routes

Routes are the final layer of the back end. They are the portion of the application that allows for connections to be made between the client and the server.

Query This code shows the blueprint for the Query routes that are available to the client via HTTP requests and JSON formatted data.

```
from flask import Blueprint, request, jsonify
import service.query as query_service


query_bp = Blueprint('query_bp', __name__)


@query_bp.route('/<int:user_id>', methods=['GET'])
def get_by_user_id(user_id):
    queries = query_service.get_by_user_id(user_id)
    return jsonify([query.to_dict() for query in queries])


# Get recent queries, used in llama_complete
@query_bp.route('/recent/<int:user_id>', methods=['GET'])
def get_recent(user_id):
    count = int(request.args.get('count', 5))
    offset = int(request.args.get('offset', 0))
    queries = query_service.get_recent(user_id, count, offset)
    print(queries)
    return jsonify([query.to_dict() for query in queries])

# Adds into database, expects user_id, question, and response
```

```
# Expects a json object
@query_bp.route('/<int:user_id>', methods=['POST'])
def create(user_id):
    data = request.get_json()
    if data is None:
        return jsonify({'error': 'Invalid request, data is missing'}), 400
    question = data.get('question')
    response = data.get('response')
    success, message = query_service.create(user_id, question=question, response=response)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 400


@query_bp.route('/<int:query_id>', methods=['DELETE'])
def delete(query_id):
    success, message = query_service.delete_by_id(query_id)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 400
```

User This code shows the blueprint for the User routes that are available to the client via HTTP requests and JSON formatted data.

```
from flask import Blueprint, request, jsonify
import service.user as user_service

user_bp = Blueprint('user_bp', __name__)


@user_bp.route('/', methods=['GET'])
def get_all():
    users = user_service.get_all()
    return jsonify([user.to_dict() for user in users])


@user_bp.route('/<int:user_id>', methods=['GET'])
def get_by_id(user_id):
    user = user_service.get_by_id(user_id)
    if user:
        return jsonify(user.to_dict()), 200
    else:
        return jsonify({'error': 'User not found.'}), 404


@user_bp.route('/login', methods=['POST'])
def login():
    data = request.json
    success, user, message = user_service.login(data['email'], data['password'])
    if success:
        return jsonify({
            'message': message,
            'user': user.to_dict_no_password()
        }), 200
    else:
        return jsonify({
            'error': message
        }), 401
```

```
@user_bp.route('/<int:user_id>', methods=['DELETE'])
def delete_user(user_id):
    success, message = user_service.delete_by_id(user_id)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 404


@user_bp.route('/<int:user_id>', methods=['PUT'])
def update_user(user_id):
    data = request.json
    success, message = user_service.update(user_id, **data)
    if success:
        return jsonify({'message': message}), 200
    else:
        return jsonify({'error': message}), 400


@user_bp.route('/register', methods=['POST'])
def register():
    data = request.json
    username = data['username']
    password = data['password']
    email = data['email']
    success, message = user_service.register(username, email, password)
    if success:
        return jsonify({'message': message}), 201
    else:
        return jsonify({'error': message}), 400
```

1.3 Back-End Code - AI

The AI consists of two different parts to help the AI know which decisions to take, which tools it can use and how, as well as communication with the database server via Routes listed above.

1.3.1 Models

Consists of two definitions written in Python, the first being *Save Response To Server*, which saves user queries to the server, and the second being *Llama Complete*, which gives multiple prompts to guide the AI to determine tool usage, safety, and follow appropriate guidelines to be an academic AI with history injection, all determined by the user's prompt.

Save Response To Server listed below is how requests are saved to the server. To explain, the definition requires:

- Userid: Requires the userid of the user currently logged in, allows linking the userid to the question and response
- Question: The question being asked by the user
- Response: The response generated by the AI

```
def save_response_to_server(userid: int, question: str, response: str):
```

The definition uses the URL for the route listed above in the Back-End Code Database - Routes to connect to the database, with data formatting the inputted question and response as a JSON parsable blob.

```
    url = f"http://server.com/api/queries/save/{userid}"
    data = {
        "question": question,
```

```
    "response": response
}
```

From here, the question is built and sent with the URL from above, same with the data. The user is alerted if the response is saved or not saved depending on if the request was successful (Code 200) or unsuccessful.

```
response = requests.post(url, json=data)
if response.status_code == 200:
    return "Success - Saved to database"
else:
    return "Error - Didn't save to database"
```

Llama Complete This is the primary chain consisting of multiple moving parts to route user requests to the appropriate response and ensuring Llama has access to the appropriate user history, tools, and response format. To begin, the definition requires two things, with one optional:

- Question: The question being asked by the user
- Userid: The userid linked to the user signed in. If no userid is supplied, ID 62 is given as a dummy variable which is used during testing.
- Debug: A debug variable used to give verbose outputs as to what the AI is doing specifically at different points of the application. This is off by default and only used within testing scenarios.

```
def llama_complete(question: str, userid: int = 62, debug: bool = False):
```

The definition starts with a variable *fuse* being created, this will later tell the AI whether the response/question combo is appropriate to be saved within the application

```
fuse = False
```

Next is the memory for use in the chain, called *router_memory*. This uses LangChain with ConversationBufferWindowMemory to allow easy sorting of what input is a user question or AI-generated response, specifically it will only hold the last 4 pairs of questions and responses in memory and return_messages means that any history will be given to the AI in list format

```
router_memory = ConversationBufferWindowMemory(k=4, return_messages=True)
```

Then it will attempt to connect to the database to pull recent question/response pairs made by the user, using the URL for the server with formatting the data from JSON into a Python readable format. If there is data, it will add on the question/response pairs into router_memory, else if there is no memory (for example a new user), it will give an error to console and add in an example question into the user's history.

```
    try:
    url = f"http://server.com/api/queries/recent/{userid}"
    response = requests.get(url)
    data = response.json()
    # If there is data, then the context memory is loaded, else its just a empty memory
    if data:
        for i in range(len(data)):
            router_memory.save_context({"input": data[i].get("question")}, {"output":
                data[i].get("response")})
except:
    print("Error: Could not load recent queries, past history will have a note in of this")
    router_memory.save_context({"input": "Who are you?"}, {"output": "I am Learnix, an AI
        designed to assist with academic and research endeavors."})
```

After is the declaration of the LLM, and of an adjusted version. *llama* is the basic version of the model, using the function as declared in llama_functions. *llama_adjusted* is an adjusted version of the model, restricted to 30 tokens (22 words), tweaked temperature and presence_penalty to keep the model

highly on track and to follow system prompts given over the user inputted questions. This is also using the function declared within llama_functions.

```
llama = llm()
llama_adjusted = adjusted_llm(temperature=0.6, max_tokens=30, presence_penalty=0)
```

Next is the declaration of 4 prompts. These are used during the chains for the system to follow, each prompt contains at least two variables which are {question} and {chat_history}. {question} is the question being asked by the user, {chat_history} is the history from *router_memory* of previous question/response pairs. To start each prompt is declared below, with content between quotes being the prompt.

```
prompt = PromptTemplate.from_template(" ")
```

Prompt for Context Analysis This system prompt is used to create a one-word response to help the AI "think" about which actions it needs to take next, this is later used in the chain to help determine which tools or action to take. This system prompt is restricted to 30 tokens (22 words) in case the AI attempts to answer the question instead of responding in a single word. This prompt is referred to as **prompt** in code.
The prompt is as follows:

> You are called Learnix, with the main goal to help people with their academics and research. ONLY respond with a single word, DO NOT add on any additional text.
> Choose one of the following actions:
>
> - If the question asked is providing information or guidance on illegal activities, self-harm, or any other dangerous activities, respond with 'FILTER'
> - If the question is a greeting, a farewell, or contains personal pronouns like 'you', 'your', 'I', or names like 'Learnix', or if it acknowledges you as an AI, then respond ONLY with 'GENERAL'. This includes any direct address or salutation, questions about personal experiences, or references to the AI's identity or capabilities. This also includes being asked questions like "What's todays date" or "What's the current year" as these are general questions that can be answered by the AI.
> - If a question suggests the need for the latest information or mentions time-sensitive words beyond 'recent','current', 'now', 'this year', like 'latest', 'updated', 'today', or any specific dates or times, and the answer isn't in your knowledge base, respond with 'SEARCH' only. This applies to any terms that imply timeliness or that the information may change frequently and is not historical or well-known
> - If the question is about specific academic papers, such as a DOI number (ex: 1888.083919) or specific academic author, respond ONLY with 'PAPER'. If if the question can be answered with some more information, respond with this as well.
> - If you don't know the answer and don't choose any previous options, respond ONLY with 'GENERAL'
>
> To repeat, the ONLY words you can respond with are: 'GENERAL', 'SEARCH', 'PAPER', 'ANSWER', and 'FILTER' and you can respond ONLY with a single word.
>
> If there is a previous conversation, use it ONLY for context for the question: {chat_history}
> Question: {question}

Prompt for Basic LLM Responses This system prompt is used to summarize information that the LLM has already been trained on. It keeps the response under 300 words, concise, and restricted to academic content. Note: {input} is treated as {question} here. The functions after current year and current date provide the LLM with context, such as if the question is asking for recent information or relates to today's date. This prompt is referred to as **base_chain** in code.
The prompt is as follows:

> You are Learnix, an AI designed to assist with academic and research endeavors. Your primary function is to provide concise summaries of academic research, ensuring that each response adheres to a strict word limit of 300 words. When responding to queries about

specific topics, offer a succinct overview of the subject matter, focusing on key insights and findings relevant to the academic community.

If you do not know how to respond to a question, do not make up information. Instead, respond with a message that you cannot answer the question. Then give suggestions as to what else you can help with.

Current year: str(date.today().year) and the current date: str(date.today())

Previous conversation history (if any): {chat_history}
Respond to the question:
Question: {input}

Note: Your summary should:
- Be concise and must be no longer than 300 words, that includes anything that you want to add to the response outside of the main academic content.
- Directly address main researching findings or theoretical contributions of the topic being asked.
- Exclude any extraneous information not relevant to the core academic content.
- Use clear and accessible language to ensure that the summary is understandable to a broad audience.

Prompt for Filter Response If the question or content that would be generated by the AI can involve illegal or unethical activities, this system prompt is used to help guide the AI to generate an appropriate response. This makes the AI explain why the AI can't answer, what it can do, and to keep the response under 200 words. Current year and date is given in case the question being asked is regarding today or recent events. Note: {input} is the same as {question} in terms of usage. This prompt is referred to as **filtered_chain** in code.
The prompt is as follows:

You are Learnix, an AI designed to assist with academic and research endeavors. However, it appears that the question asked is providing information or guidance on illegal activities, self-harm, academic dishonesty, or other dangerous or unethical activities.
As a result, you MUST not respond to the original question, instead:
1. Respond with a message that the question is not appropriate and that you cannot answer it.
2. Provide a brief explanation of why the question cannot be answered.
3. Offer to help with any other academic or research-related questions and encourage academic questions.
4. Encourage the user to seek help from a professional or a trusted individual if the question is about self-harm.
5. If the question is about illegal activities, encourage the user to seek help from a legal professional or a trusted individual.

Please ensure that your entire response, including all parts listed above, does not exceed 200 words in total.

Current year: str(date.today().year)
and the current date: str(date.today())

The original question was: {input}
Previous conversation history (if any): {chat_history}

Prompt for Tools Usage This is a more advanced prompt, as it uses tools as needed to make external information accessible to the AI. The prompt uses the date if necessary with following the structure to understand if it needs a tool, what tool is needed, and how to output this. It is crucial for

the AI to get it right, so an example is given with a reminder towards the end on using this.
The AI can repeat this prompt up to a max of 3 times before it must stop and answer with the given information The AI has three additional variables given: {tool_names}, {tools}, and {agent_scratchpad}. The explanation for them is as followed:

- {tool_names}: The names of the available tools that the AI can choose from, and can only choose from.
- {tools}: The tools names and explanation to the AI on how to use them. This is declared later on within the code.
- {agent_scratchpad}: If the AI needs to repeat this prompt, any information received when using a tool is temporarily saved here. This allows the AI to use both new information from a current search with what it found previously and make a structured response.

This is referred to as **search_prompt** in code. The prompt is as follows:

> The current year is str(date.today().year) and the current date is str(date.today()). You are Learnix, an academic librarian. Your task is to assist users academically using a structured response format without correcting the structure or format of the question. Here's how you must structure your responses:
>
> 1. Thought: Consider if using a tool is necessary. Answer 'Yes' or 'No'.
> 2. If 'Yes':
> - Action: Specify which tool you will use. Choose from [{tool_names}]. Use the tool's exact name without any punctuation or additional text.
> - Action Input: Give the input you want to the tool, make sure to follow what the tool expects for Action Input.
> - Observation: Describe in detail the outcome of using the tool. Include specific information, data, or insights gained from the tool's output.
> 3. If 'No' or after using tools:
> - Final Answer: Using the detailed observations from the tool(s), provide a comprehensive and accurate answer to the question. Ensure the final answer directly utilizes the information gathered during the Observation step.
>
> It is crucial to follow this format strictly. For example:
>
> Thought: Do I need to use a tool? Yes
> Action: Wikipedia
> Action Input: 'Quantum Computing Basics'
> Observation: Wikipedia provided a detailed overview of quantum computing principles.
> Thought: Do I need to use a tool? No
> Final Answer: Quantum computing is a field of computing focused on developing computer technology based on the principles of quantum theory...
>
> You have access to the following tools:
>
> {tools}
>
> Do not include any additional text outside of Thought, Action, Action Input, Observation, or Final Answer.
> Your response must strictly adhere to the provided structure. Begin!
>
> Question: {input}
> Thought:{agent_scratchpad}

The next step within the code is the declaration of the tools needed for the AI. There are three tools the AI can use and are declared: DuckDuckGo, ArXiv, and Wikipedia. *Search* is used with DuckDuckGo's API wrapper to limit it to only showing the three most recent results per use. *arxivsearch* and *wikipedia* are used with ArXiv and Wikipedia's APIs respectively, both pulling the three most recent search results by default.

```
search = DuckDuckGoSearchResults(api_wrapper=DuckDuckGoSearchAPIWrapper(max_results=3))
arxivsearch = ArxivAPIWrapper()
wikipedia = WikipediaAPIWrapper()
```

After is setting it up as a tool chain so the AI can read and understand how to use the tools. This includes the name of the tool, description, and what function is called on when the AI asks to use the tool in question. This is declared as *paper_tools* and is given to the AI as such.
For arXiv (And beginning of the *paper_tools* declaration):

```
paper_tools = [
    Tool(
        name="Arxiv",
        func=arxivsearch.run,
        description="""Searches for the 3 most recent papers on the topic. In this case, it
            is used to search for academic papers on the web. Information from this tool is
            always accurate, but make sure to summarize the information. When using the
            information from the tool, make sure to tell the user information from arxiv
            only shows the 3 most recent papers from the author, on the topic, or similar
            to DOI number that is given. Action Input must be the exact search query for
            the paper. Example: 'History of Artificial Intelligence'"""
    ),
```

For Wikipedia tool declaration:

```
        Tool(
        name="Wikipedia",
        func=wikipedia.run,
        description="""Used to search for information in a more granular way, meaning alot
            more details on general topics. This can be used for authors, places, events,
            topics, and more. When using the information from the tool, make sure to tell
            the user that the information from wikipedia is often accurate but should be
        verified alongside other sources. Action Input must be the exact search query for
            the topic. Example: 'Evolution of Quantum Computing' """
    ),
```

And for DuckDuckGo declaration and end of *paper_tools* declaration:

```
        Tool(
        name="DuckDuckGo",
        func=search.run,
        description="""
        Use DuckDuckGo Search Tool for single, focused searches in academic research. It
            retrieves factual data from Instant Answers and top search results,
            synthesizing it into a coherent response.

        **Guidelines:**
        - Input a clear, concise search query.
        - Tool performs one search per query, avoiding loops.
        - Extracts and processes relevant information for a direct response.

        **Example:**
        - Input: 'Quantum Computing advancements'
        - The tool searches, then provides a synthesized response based on authoritative
            sources. Designed for efficient, loop-free academic research using DuckDuckGo.
        """
    )
]
```

Agents are then made after this, which links together the prompts, tools (if any are needed), and which AI model to use. The agent in use is made with create_react_agent from LangChain, which is a function that executes the AI which can be repeated multiple times succeeding and has data carry over between usage. AgentExecutor is what allows the agent to be run within code and gives specific parameters to adhere to, such as tool's once more, verbose output (limited to debugging scenarios), return_intermediate_steps meaning to return data between repeats, how many times to repeat in a row before stopping, and if the AI should handle issues when reading the prompt

```
online_agent = create_react_agent(llm=llama, tools=paper_tools, prompt=search_prompt)
research_executor = AgentExecutor(agent=online_agent,
                                  tools=paper_tools,
                                  verbose=debug,
                                  return_intermediate_steps=True,
                                  max_iterations=3,
                                  handle_parsing_errors=True)
```

Next is the creation of *router_chain*. This is the 'Context Analysis' part of the AI, which links together *prompt* as seen above, which AI model to use which is the adjusted model restricted to 30 tokens (22 word) and be parsed with *StrOutputParser* which will put it in a way Python can use.

```
router_chain = prompt | llama_adjusted | StrOutputParser()
```

Then the declaration for *chain_decision* and, unlike *save_response_to_server*, this is a definition within this function limited to only being used when needed for *router_chain*. *output* in this case is the single word response made from Context Analysis. Fuse is declared as nonlocal to update the fuse variable at the start of Llama Complete in case an error happens or triggers the filtered function. Then a try chain starts.

```
def chain_decision(output):
    nonlocal fuse
    try:
```

The first part of the try chain is to compare the output against 'GENERAL' or 'ANSWER'. If this is the case, the console logs that it will be using the base model and will add the users' chat history (being the last 4 question/response pairs) to the initial question. From here the prompt is invoked via *base_chain.invoke* with the response saved into *temp_dict* and returned to the user.

```
if output["action"] == "GENERAL" or output["action"] == "ANSWER":
    print('Using base llama2 - Model Generating...')
    output["chat_history"] = router_memory.load_memory_variables({})
    temp_dict = {'input': output.get("input"), 'output': base_chain.invoke(output)}
    return temp_dict
```

The second and third parts of the try chain compare output to 'SEARCH' or 'PAPER', both of which use research_executor (search_prompt with agent executing it) that will use tools and return the response directly to the user. Note: *research_executor* already has the history of the user due to the AgentExecutor declaration made previously. The reason this is split into two is so the AI has an idea of which tool to consider using first.

```
elif output["action"] == "SEARCH":
    print("Model using tools - Model Searching & Summarizing... (Search)")
    return research_executor
elif output["action"] == "PAPER":
    print("Model using tools - Model Searching & Summarizing... (Paper)")
    return research_executor
```

The fourth part of the try chain is to compare the output to 'FILTER'. In this case, it means that the filter was tripped and will print to console this too. It will grab the users' history and add it to the *filtered_chain* response, returning the response to the user. Note: Fuse is set to true in this instance due to the filter being tripped and not wanting to save the response at all.

```
elif output["action"] == "FILTER":
    print("Azure Saftey / Llama Filter Tripped - Model Generating...")
```

```
    output["chat_history"] = router_memory.load_memory_variables({})
    fuse = True
    temp_dict = {'input': output.get("input"), 'output': filtered_chain.invoke(output)}
    return temp_dict
```

The last part of the try chain is a fallback in case the comparison fails for some reason. An alert is given to the console noting a possibility of a hallucination and this fallback being triggered. User history is added on to the response, and the *base_chain* (base AI model) is used to generate an answer.

```
else:
    print("ALERT: AI could not make a decision, falling back to general response\nResponse
        can be hallucinated")
    output["chat_history"] = router_memory.load_memory_variables({})
    temp_dict = {'input': output.get("input"), 'output': base_chain.invoke(output)}
    return temp_dict
```

There are two except cases for this try chain, dealing with two error types. One being timeouts, the other being HTTP errors with requests not succeeding correctly. The first, listed below, handles when a timeout occurs when a response isn't generated within 50 seconds. If this is the case, it will trigger the Fuse to not save the response and alert the user of the error and what they could do.

```
except TimeoutError:
    fuse = True
    return {"input": output.get("input"), "output": "The response took too long to
        generate, please try again. If the problem persists, please try another question or
        notify the developers."}
```

The next except deals with HTTP Errors, which is when a request does not complete correctly or faces an error. There are two error types that are generated this late into the chain: Error 424 and Error 500.
Error 424 at this stage indicates that Azure caught an inappropriate response being generated by the AI and throws this error, and in this case, the Fuse is tripped and an explanation of the safety error is sent to the user.
Error 500 at this stage indicates that Azure could not generate a response for some reason, typically when Azure's endpoint is given too many responses at once and the server is overwhelmed. In this case, this is not solvable by the application or Flask server, with an explanation sent to the user to wait and what might be happening at the time.

```
except HTTPError as err:
    if err.code == [424, 500]:
        if err.code == 424:
            fuse = True
            return {"input": output.get("input"), "output": "Hi there, your prompt has
                tripped the content safety filter and unfortunately I cannot answer your
                question. Please know that I am here to help with any questions that depend
                on academic research and learning. If you have any other questions, feel
                free to ask!"}
        elif err.code == 500:
            fuse = True
            return {"input": output.get("input"), "output": "There was a problem with the AI
                trying to create your request as it seems like it might be overwhelmed.
                Please try again at a later time. If the problem persists, please alert the
                developers or check your internet connection."}
```

After this, the chain is declared and prepped so that when the user inputs a question, it can invoke the try:except chain above. This does occur afterwords, as the declaration for *chain_decision* needs to be declared before this can be made. It uses a RunnableMap from LangChain to help map out which actions need to be taken before calling the chain, in this case parsing the input for the keywords necessary to invoke the if statement

```
chain = RunnableMap({
    "action": router_chain,
    "input": lambda x: x["question"]
```

```
}) | chain_decision
```

Next is the declaration of a try:except statement to start the RunnableMap. This initially formats the question and adds on the chat history for use within *chain_decision*.

```
try:
    response = chain.invoke(
        {
            "question": question,
            "chat_history": router_memory.load_memory_variables({})
        }
    )
```

If the chain fails for some reason, meaning a failure with connecting to the AI model, it will catch the exception and trigger the Fuse to not save the response. In this case, the user is alerted about the error and given an explanation as to what to try to resolve the issue.

```
except Exception as e:
    fuse = True
    print("Error: ", e)
    response = {"input": question, "output": "There was a problem attempting to generate a
        response, please wait "
    "and try again at a later time. If the problem persists, "
    "please check your internet connection."}
```

Fuse is then checked if false or not, if it is false then it is logged to console that the response is not saved at all. If it is true, then *save_response_to_server* is triggered with the user's id, question asked, and the response the AI generated.

```
if fuse is False:
    save_response_to_server(userid, question, response.get("output"))
else:
    print("Response was not saved due to an function error of the AI")
```

Lastly, the response is returned to the user once *chain_decision* has been successfully triggered with a response generated by the AI.

```
return response.get("output")
```

1.3.2 Services

This consists of seven definitions written in Python, with all seven of them located within *llama_functions.py*. In production, *llm* and *adjusted_llm* are used in a couple of ways within models. The rest of the Services were used early on in testing, debugging, or are unused at this stage of the application.

LLM In the definition for *LLM*, it consists of retrieving the API key from *config.py* located in the root directory of the project, and the URL needed to connect to the Azure Endpoint. Then it is used within the *model* declaration using AzureMLChatOnlineEndpoint, which is a function of LangChain to format the request into what Azure's Endpoint expects. In *model* it uses the URL and API key, but also information about the type of Endpoint (Serverless in this case), how to format the content (Using LLamaChatContentFormatter due to the model in use being Llama2), and which keywords need to be passed in. The keywords consist of temperature being 0.6 to focus more on the system prompt, and the max tokens being 600 (450 words) for use within the LLM.

```
def llm():
    azure_key = ConfigAzure.azure_key
    url = 'https://servername.ai.azure.com/v1/chat/completions'
    model = AzureMLChatOnlineEndpoint(
        endpoint_url=url,
        endpoint_api_type=AzureMLEndpointApiType.serverless,
        endpoint_api_key=azure_key,
        content_formatter=LlamaChatContentFormatter(),
```

```
        model_kwargs={"temperature": 0.6,
                      "max_tokens": 600},
    )
return model
```

Adjusted LLM This is similar to LLM as above, but with a few changes. The model takes in 3 parameters that are then passed to *model* as keywords. Temperature determines how closely it should stay relative to the system prompt or user prompt, max tokens indicates how long Llama2 can generate a response, and presence penalty (between -2.0 and 2.0) determines how likely the model will talk about new subjects.

```
def adjusted_llm(temperature: float, max_tokens: int, presence_penalty: float):
    azure_key = ConfigAzure.azure_key
    url = 'https://servername.ai.azure.com/v1/chat/completions'
    model = AzureMLChatOnlineEndpoint(
        endpoint_url=url,
        endpoint_api_type=AzureMLEndpointApiType.serverless,
        endpoint_api_key=azure_key,
        content_formatter=LlamaChatContentFormatter(),
        model_kwargs={"temperature": temperature,
                      "max_tokens": max_tokens,
                      "presence_penalty": presence_penalty},
    )
    return model
```

Prompt Template Prompt Template is a basic prompt to be used with *conversation_with_memory* and *adjusted_conversation*. This prompt consists of SystemMessage, which is the system prompt for the AI to follow, MessagesPlaceholder, which holds the chat_history of the conversation, and HumanMessagePromptTemplate, which takes in the user question and formats it for the AI to read. Note: The definition itself does not take in these variables as LangChain handles this during execution.

```
def prompt_template():
    prompt = ChatPromptTemplate.from_messages(
        [
            SystemMessage(
                content="""You are Learnix, with the main goal to help people with their
                    academics and research.\n
                If there was a previous conversation, here is the history: {chat_history}"""
            ),
            MessagesPlaceholder(
                variable_name="chat_history"
            ),
            HumanMessagePromptTemplate.from_template(
                "{user_input}"
            ),
        ]
    )
    return prompt
```

Prompt Template No History Prompt Template No History is similar to Prompt Template except it does not have any place for holding user history. This is useful to test out simple questions with an AI to determine if the AI is available / how the AI interacts with specific prompts. Prompt consists of SystemMessage, which is the system prompt for the AI to follow, and HumanMessagePromptTemplate, which takes in the user question and formats it for the AI to read. Note: The definition itself does not take in these variables as LangChain handles this during execution.

```
def prompt_template_no_history():
    prompt = ChatPromptTemplate.from_messages(
        [
```

```
            SystemMessage(
                content="""You are Learnix, with the main goal to help people with their
                    academics and research.\n
                If there was a previous conversation, here is the history: {chat_history}"""
            ),
            HumanMessagePromptTemplate.from_template(
                "{user_input}"
            ),
        ]
    )
return prompt
```

Conversation With Memory Conversation with Memory takes in two parameters: *memory_key*, which is a key to help keep track of the specific conversation memory, and *question*, which is the question the user wants to be answered. First the AI model is made with the *llm* function, followed by grabbing *prompt_template*. Memory is declared with ConversationBufferWindowMemory using *memory_key* and will return messages made in the conversation into memory. After of which the question and history is given to the AI using LLMChain from LangChain, with the response returned.

```
def conversation_with_memory(memory_key: str, question: str):
    llama = llm()
    template = prompt_template()
    memory = ConversationBufferWindowMemory(memory_key=memory_key, return_messages=True,
        k=5)
    chat = LLMChain(llm=llama, prompt=template, verbose=False, memory=memory)
    response = chat.predict(user_input=question)
    return response
```

Conversation Conversation is similar to Conversation with Memory, the main difference is the lack of memory for the user's questions asked. The function declares the AI model using *llm*, and pulls in the prompt without history via *prompt_template_no_history*. Then the question from the parameters and gives it to LLMChain for the AI to answer, with the response being returned.

```
def conversation(question: str):
    llama = llm()
    template = prompt_template_no_history()
    chat = LLMChain(llm=llama, prompt=template, verbose=False)
    response = chat.predict(user_input=question)
    return response
```

Adjusted Conversation Adjusted Conversation is similar to Conversation, except allows parameters for the AI to have its temperature, max tokens, and presence penalty adjusted. It first declares the AI by using the temperature, max tokens, and presence penalty to help create it. Then it pulls in the template without history via *prompt_template_no_history*, and uses the prompt in conjunction with the AI declared to ask the question. The response given from the AI is then returned.

```
def adjusted_conversation(question: str, temperature: float, max_tokens: int,
     presence_penalty: float):
    llama = adjusted_llm(temperature, max_tokens, presence_penalty)
    template = prompt_template_no_history()
    chat = LLMChain(llm=llama, prompt=template, verbose=False)
    response = chat.predict(user_input=question)
    return response
```

1.3.3 Routes

This consists of four routes that are used to help direct requests recieved from the Flask server to the correct functions. Each function can be accessed via an HTTP request using the path of *http://server.com/api/llama/route*, where *route* is replaced by the route name given in each explanation.

Adjusted Conversation This is the route to access *adjusted_conversation* from within *llama_functions.py*, and can be requested via POST with the root of *http://server.com/api/llama/adjusted_conversation*. This request requires data to be sent as JSON containing a question to be asked, with temperature, max tokens, and presence penalty for the AI to be. Then the function *adjusted_conversation* is invoked and the answer returned as JSON.

```
@llama_bp.route('/adjusted_conversation', methods=['POST'])
def adjusted_conversation():
    data = request.json
    question = data['question']
    temperature = data['temperature']
    max_tokens = data['max_tokens']
    presence_penalty = data['presence_penalty']
    response = llama.adjusted_conversation(question, temperature, max_tokens,
        presence_penalty)
    return jsonify({'response': response}), 200
```

Conversation This is the route to access *conversation* from within *llama_functions.py*, and can be requested via POST with the root of *http://server.com/api/llama/conversation*. This request requires data to be sent as JSON containing a question to be asked. Then the function *conversation* is invoked, and the answer returned as JSON.

```
@llama_bp.route('/conversation', methods=['POST'])
def conversation():
    data = request.json
    question = data['question']
    response = llama.conversation(question)
    return jsonify({'response': response}), 200
```

Conversation With Memory This is the route to access *conversation_with_memory* from within *llama_functions.py*, and can be requested via POST with the root of *http://server.com/api/llama/memory_and_conv*. This request requires data to be sent as JSON containing a memory key in order to keep track of message history, and the question to be asked. Then the function *conversation_with_memory* is invoked, and the answer returned as JSON.

```
@llama_bp.route('/memory_and_conv', methods=['POST'])
def conversation_with_memory():
    data = request.json
    memory_key = data['memory_key']
    question = data['question']
    response = llama.conversation_with_memory(memory_key, question)
    return jsonify({'response': response}), 200
```

Llama Complete This is the route to access *llama_complete* from within *llama_complete.py* within Models, and can be requested via POST with the root of *http://server.com/api/llama/llama_complete*. This request requires data to be sent as JSON containing the question to be asked, and the userid of the user who is asking the question. Then the funciton *llama_complete* is invoked, and the answer returned as JSON.

```
@llama_bp.route('/llama_complete', methods=['POST'])
def llama_complete():
    data = request.json
    question = data['question']
    userid = data['userid']
    message = llama_advanced.llama_complete(question, userid)
    return jsonify({'response': message}), 200
```

2 Test Cases

2.1 Test Case Sign up Screen

Description: This test case ensures that the functionality of the login screen is working properly. The login screen is responsible for verifying user's credentials and allowing access to the system's functionalities. It also needs to handle situations when the users enter incorrect credentials.

Associated Requirements:

1. The application must provide a proper user interface for login screen.
2. The login screen should only give access if the credential matches the information in database, if not they can't gain access.
3. There should be a sign-up option for the users who don't have an account.

Status: Successful
Notes: The login screen displays correctly, and all the button works. The application is connected to the server correctly, and the user's credentials are properly verified.

2.2 Test Case Sign up Screen

Description: This test case ensures that the functionality of the sign-up screen. The sign-up screen is responsible for allowing users to create an account by entering their email address and password, so the users can successfully register.

Associated Requirements:

1. The application must provide a proper user interface for sign-up screen.
2. The sign-up screen should properly take inputs from the user.
3. After completing the sign up the user should automatically be directed to the main page.

Status: Successful

Notes: The sign-up screen displays correctly, the button works, and it automatically displays the main page after finishing the sign-up. The application is linked to the server correctly, and the user's credentials are properly stored in the database.

2.3 Test Case Main Screen

Description: This test case ensures that the functionality of the main screen. The main page is responsible for allowing users to view and interact with a variety of features, including asking questions, accessing the stored last five questions, and logging out. It assures that all key features are reachable and operational and that users can navigate the home page with ease.

Associated Requirements:

1. The application must display after log in.
2. Users should be able to ask questions.
3. After asking questions it should store the five recent questions and the users should be able to access it.
4. Users should be able to log out.

Status: Successful

Notes: The main screen displays correctly. The logout button works, the latest 5 search histories are stored, and it can be accessed. The application is connected to the server correctly, and the ai gives proper answers to questions.

2.4 Test Case AI Functionality

Description: This test case ensures that the AI can answer questions accurately and use various tools available to the AI as seen in *llama_complete.py*. The AI functionality is crucial to be fully functional to act as an academic chatbot with up-to-date, accurate information, and safety features to be accessible. It ensures that tools that are available to the AI are used properly with no issues when testing

Associated Requirements:

1. Launch Flask on localhost:5000
2. Open *LLamaCompleteTesting.py* located in *Capstone_Project_One/ai-testing*
3. Run *LlamaCompleteTesting.py*
4. Responses should answer each of the 14 questions without errors

Status: Successful

Notes: Latency is not something that should be tested or noted at all during this test. During this test, all 14 questions were answered appropriately, with questions requiring tools giving factual (and up-to-date) information when necessary. Chat history retrieval questions access history correctly, and tools will often cite what sources were used / tool used.