Viva Preparation Guide - AI-Powered Search Assistant

This document provides a comprehensive overview of the project, libraries used, and key concepts to help you prepare for your viva examination.

Table of Contents

Project Overview

Project Name: AI-Powered Search Assistant (Perplexity Clone)

Description: A web application that provides intelligent search capabilities using AI to generate optimized search queries, fetch information from multiple sources, and synthesize comprehensive answers. The application includes user authentication, credit-based usage system, and search history management.

Main Purpose:

- Provide AI-enhanced search functionality
- Generate multiple optimized search queries from user questions
- Scrape and synthesize information from multiple web sources
- Deliver comprehensive, well-structured answers with citations

---

Technology Stack

Frontend

- **HTML5** - Structure and markup

- **CSS3** - Styling with CSS variables for theming
- **JavaScript (ES6+)** - Client-side logic and interactivity
- **Marked.js** (CDN) - Markdown parsing for rendering AI responses
- **Supabase JS Client** (CDN) - Authentication and database operations

Backend

- **Python 3** - Server-side programming
- **Flask** - Web framework for API endpoints
- **Flask-CORS** - Cross-Origin Resource Sharing support

APIs and Services

- **Google Custom Search API** - Web search functionality
- **Google Gemini AI** - AI model for query generation and answer synthesis
- **Supabase** - Backend-as-a-Service for authentication and database

Data Processing

- **BeautifulSoup4** - HTML parsing and web scraping
- **lxml** - XML/HTML parser for BeautifulSoup

□ Libraries and Frameworks

1. Flask

Purpose: Web framework for building REST API endpoints

Why Used:

- Lightweight and easy to use
- Perfect for building simple REST APIs
- Good for rapid development

Key Usage:

```
from flask import Flask, request, jsonify
app = Flask(__name__)
@app.route('/search', methods=['POST'])
```

What to Know:

- Flask is a micro web framework
- Uses decorators for routing (`@app.route()`)

- Handles HTTP requests and responses
- Supports JSON data exchange

---

2. Flask-CORS

Purpose: Enables Cross-Origin Resource Sharing (CORS)

Why Used:

- Frontend (HTML) and backend (Flask) run on different origins
- Allows frontend to make API requests to Flask server
- Prevents CORS errors in browser

Key Usage:

```
from flask_cors import CORS
CORS(app)
```

What to Know:

- CORS is a security feature in browsers
- Needed when frontend and backend are on different ports/domains
- Without CORS, browser blocks cross-origin requests

---

3. BeautifulSoup4

Purpose: HTML parsing and web scraping

Why Used:

- Extract text content from web pages
- Parse HTML structure
- Clean and process web content

Key Usage:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_page, 'lxml')
main_content = soup.find('main')
page_text = main_content.get_text(separator=' ', strip=True)
```

What to Know:

- Parses HTML/XML documents

- Provides methods like `find()`, `find_all()`, `get_text()`
- `lxml` is the parser backend (faster than default)
- Used to extract readable text from web pages

---

4. Google API Python Client

Purpose: Interact with Google Custom Search API

Why Used:

- Access Google's search results programmatically
- Get search results with titles, links, and snippets
- Official Google library for API access

Key Usage:

```
from googleapiclient.discovery import build
service = build("customsearch", "v1", developerKey=api_key)
response = service.cse().list(q=query, cx=cx_id, num=num_results).execute()
```

What to Know:

- Google Custom Search API provides programmatic search access
- Requires API key and Custom Search Engine ID
- Returns structured JSON results
- Limited free quota (100 queries/day)

---

5. Google Generative AI (Gemini)

Purpose: AI model for generating search queries and synthesizing answers

Why Used:

- Generate optimized search queries from user questions
- Synthesize comprehensive answers from multiple sources
- Natural language understanding and generation

Key Usage:

```
import google.generativeai as genai
genai.configure(api_key=MY_API_KEY)
model = genai.GenerativeModel("gemini-2.5-flash")
response = model.generate_content(prompt, generation_config={...})
```

What to Know:

- Gemini is Google's AI model (similar to GPT)
- Used for two main tasks:
    1. **Query Generation:** Convert user question into 3-4 optimized search queries
    2. **Answer Synthesis:** Combine information from multiple sources into coherent answer
- Supports temperature, max tokens, and other generation parameters
- Returns text responses that can be formatted as markdown

---

6. Supabase

Purpose: Backend-as-a-Service for authentication and database

Why Used:

- User authentication (sign up, login, logout)
- Database for storing user credits
- Real-time capabilities
- Easy integration with JavaScript

Key Usage:

```
// Authentication
supabaseClient.auth.signUp({ email, password })
supabaseClient.auth.signInWithPassword({ email, password })

// Database
supabaseClient.from('user_credits').select('credits')
supabaseClient.from('user_credits').update({ credits: newValue })
```

What to Know:

- Supabase is an open-source Firebase alternative
- Provides:
    - Authentication (email/password, OAuth, etc.)
    - PostgreSQL database
    - Real-time subscriptions
    - Storage
- Uses Row Level Security (RLS) for data protection
- Client-side SDK for easy frontend integration

---

7. Marked.js

Purpose: Markdown to HTML converter

Why Used:

- Convert AI-generated markdown responses to HTML
- Display formatted text (bold, lists, headings, etc.)
- Better readability of AI responses

Key Usage:

```
// Loaded from CDN
if (typeof marked !== 'undefined') {
    renderedContent = marked.parse(data.summary);
}
```

What to Know:

- Markdown is a lightweight markup language
- AI generates responses in markdown format
- Marked.js converts markdown to HTML for display
- Supports headings, lists, bold, italic, code blocks, etc.

□ Project Architecture

System Flow

```
User Input → Frontend (HTML/JS)
    ↓
Flask Backend (/search endpoint)
    ↓
1. Generate Search Queries (Gemini AI)
    ↓
2. Search Google (Custom Search API)
    ↓
3. Scrape Web Pages (BeautifulSoup)
    ↓
4. Synthesize Answer (Gemini AI)
    ↓
Return JSON Response → Frontend
    ↓
Display Formatted Answer
```

File Structure

```
perplexityClone/
├── index.html          # Frontend UI
├── styles.css          # All CSS styles
├── script.js           # All JavaScript logic
├── app.py              # Flask backend server
├── requirements.txt    # Python dependencies
├── supabase_setup.sql  # Database schema
└── VIVA_PREPARATION.md # This file
```

□ Key Features

1. Multi-Query Search System

- **What:** Generates 3-4 optimized search queries from a single user question
- **Why:** Improves search coverage and finds more relevant information
- **How:** Uses Gemini AI to analyze the question and generate diverse queries

2. Web Scraping

- **What:** Extracts text content from search result pages
- **Why:** Gets actual content, not just snippets
- **How:** Uses BeautifulSoup to parse HTML and extract main content

3. Answer Synthesis

- **What:** Combines information from multiple sources into one comprehensive answer
- **Why:** Provides complete, well-structured answers instead of raw search results
- **How:** Uses Gemini AI to analyze all scraped content and generate markdown-formatted answer

4. User Authentication

- **What:** Sign up, login, logout functionality
- **Why:** Track usage, manage credits, personalize experience
- **How:** Supabase authentication with email/password

5. Credit System

- **What:** Users have credits that are deducted per search
- **Why:** Control usage, potential monetization
- **How:** Supabase database stores user credits, updated after each search

6. Search History

- **What:** Stores previous searches locally
- **Why:** Allow users to revisit previous searches
- **How:** Browser localStorage stores search history

7. Follow-up Questions

- **What:** Ask follow-up questions related to previous searches
- **Why:** Enable conversational search experience
- **How:** Passes previous context to AI for better follow-up answers

---

□ Implementation Details

Backend (app.py)

1. Query Generation Function

```
def generate_search_queries(user_query):
    # Uses Gemini AI to generate 3-4 optimized queries
    # Handles errors with fallback query variations
```

Key Points:

- Takes user question as input
- Prompts Gemini to generate multiple search queries
- Cleans and validates the generated queries
- Returns list of 3-4 queries

2. Google Search Function

```
def search_google_api(query, api_key, cx_id, num_results=1):
    # Uses Google Custom Search API
    # Returns list of search results with links, titles, snippets
```

Key Points:

- Requires Google API key and Custom Search Engine ID
- Returns structured results (link, title, snippet)
- Handles API errors gracefully

3. Web Scraping Function

```
def fetch_page_content(url):
    # Uses BeautifulSoup to extract text from web pages
    # Limits content to 8000 characters per page
```

Key Points:

- Fetches HTML from URLs
- Parses with BeautifulSoup
- Extracts main content (looks for `<main>` or `<article>` tags)
- Limits content size to manage token usage

4. Answer Synthesis Function

```
def synthesize_answer(user_query, all_content, previous_context=None):
    # Uses Gemini AI to create comprehensive answer
    # Supports follow-up questions with context
```

Key Points:

- Combines all scraped content
- Generates markdown-formatted answer
- Can include previous context for follow-ups
- Uses specific generation parameters (temperature, max tokens)

5. Flask Route

```
@app.route('/search', methods=['POST'])
def search():
    # Main endpoint that orchestrates the entire search process
```

Workflow:

1. Receives user query from frontend
2. Generates search queries
3. Searches Google for each query
4. Scrapes content from result pages
5. Synthesizes comprehensive answer
6. Returns JSON response with answer, sources, and queries used

---

Frontend (index.html, script.js, styles.css)

1. Supabase Initialization

```
function initSupabase() {
    // Initializes Supabase client
    // Checks for existing session (auto-login)
}
```

2. Authentication Flow

- **Sign Up:** Creates account, initializes 10 credits
- **Login:** Authenticates user, loads credits
- **Logout:** Signs out, clears user data
- **Auto-login:** Checks for existing session on page load

3. Search Flow

```
async function performSearch(query, isFollowup) {
    // 1. Check authentication
    // 2. Check credits
    // 3. Deduct credit
    // 4. Show loading animation
    // 5. Send request to Flask backend
    // 6. Display results
    // 7. Save to history
}
```

4. History Management

- Stores searches in localStorage
- Displays in sidebar
- Allows viewing previous searches
- Supports follow-up questions from history

5. UI Components

- **Search Bar:** Top bar for initial search, bottom bar for follow-ups
- **Chat Interface:** Displays user questions and AI answers
- **Sidebar:** Search history with view/follow-up options
- **Login Modal:** Sign up and login forms
- **Loading Animation:** Shows search progress steps

□ Database Schema

Supabase Tables

1. `user_credits` Table

```
CREATE TABLE user_credits (
    user_id UUID PRIMARY KEY REFERENCES auth.users(id),
    credits INTEGER DEFAULT 10
);
```

Purpose: Stores credit balance for each user

Fields:

- `user_id`: Foreign key to Supabase auth.users
- `credits`: Integer count of available credits

Operations:

- Insert: When user signs up (10 credits)
- Select: Get current credit balance
- Update: Deduct credit after search

2. Authentication (Supabase Built-in)

- `auth.users` table managed by Supabase
- Stores email, password hash, user metadata

□ API Integration

1. Google Custom Search API

Endpoint: `https://www.googleapis.com/customsearch/v1`

Authentication: API Key

Parameters:

- `q`: Search query
- `cx`: Custom Search Engine ID
- `num`: Number of results

Response Format:

```
{
  "items": [
    {
      "link": "https://example.com",
      "title": "Page Title",
      "snippet": "Description..."
    }
  ]
}
```

Limitations:

- Free tier: 100 queries per day
- Requires Custom Search Engine setup

---

2. Google Gemini API

Model: `gemini-2.5-flash`

Two Main Uses:

a) Query Generation:

- Input: User question
- Output: 3-4 optimized search queries
- Temperature: 0.8 (more creative)

b) Answer Synthesis:

- Input: User question + scraped content
- Output: Comprehensive markdown answer
- Temperature: 0.7 (balanced)
- Max tokens: 4096

Generation Config:

```
generation_config={
    "temperature": 0.7,
    "top_p": 0.9,
    "top_k": 40,
    "max_output_tokens": 4096,
}
```

---

3. Supabase API

Authentication:

- `signUp()`: Create new user
- `signInWithPassword()`: Login
- `signOut()`: Logout
- `getSession()`: Get current session

Database:

- `from('table').select()`: Read data
- `from('table').insert()`: Create data
- `from('table').update()`: Update data
- `from('table').delete()`: Delete data

□ Common Viva Questions

1. Why did you choose Flask over Django?

Answer: Flask is lightweight and perfect for building simple REST APIs. Django is more feature-rich but heavier. For this project, we only need API endpoints, so Flask is more appropriate.

2. Why use Gemini AI instead of ChatGPT?

Answer: Gemini is Google's AI model with good performance and reasonable pricing. It integrates well with Google services and provides good quality responses for both query generation and answer synthesis.

3. How does the multi-query system work?

Answer: Instead of searching with the user's exact question, we use AI to generate 3-4 optimized search queries. This improves coverage by searching from different angles, leading to more comprehensive results.

4. Why scrape web pages instead of using snippets?

Answer: Snippets are limited and may not contain complete information. Scraping actual page content gives us more detailed information to synthesize better answers.

5. How do you handle errors in web scraping?

Answer: We use try-catch blocks, set timeouts (10 seconds), and gracefully handle failures. If a page can't be scraped, we continue with other sources.

6. What is the credit system for?

Answer: The credit system controls usage and allows for potential monetization. Each search costs 1 credit. New users get 10 free credits.

7. Why use Supabase instead of building your own auth?

Answer: Supabase provides secure, production-ready authentication with minimal code. Building custom auth is complex and error-prone. Supabase handles security best practices automatically.

8. How does the follow-up question feature work?

Answer: When a user asks a follow-up, we pass the previous question, answer, and sources as context to the AI. This helps the AI understand the conversation context and provide relevant follow-up answers.

9. Why store search history in localStorage?

Answer: localStorage is simple and doesn't require backend storage. It persists across sessions and is sufficient for this feature. For production, we might move to database storage.

10. What are the limitations of this system?

Answer:

- Google Custom Search API: 100 queries/day free limit
- Web scraping: Some sites block scrapers
- Token limits: AI models have token limits
- Rate limiting: Need to handle API rate limits

11. How would you improve this project?

Answer:

- Add caching to reduce API calls
- Implement rate limiting
- Add more AI models as fallback
- Improve error handling
- Add user feedback mechanism
- Implement payment system for credits
- Add search result ranking/quality scoring

12. Explain the search workflow step by step.

Answer:

1. User enters question
2. Frontend checks authentication and credits
3. Frontend sends POST request to Flask backend
4. Backend uses Gemini to generate 3-4 search queries
5. Backend searches Google for each query
6. Backend scrapes content from result pages
7. Backend uses Gemini to synthesize answer from all content
8. Backend returns JSON with answer, sources, and queries
9. Frontend displays formatted answer
10. Frontend saves to history and deducts credit

13. What security measures are implemented?

Answer:

- Supabase handles secure authentication (password hashing, JWT tokens)
- Input validation on frontend and backend
- HTML escaping to prevent XSS attacks
- CORS configuration for API security
- API keys stored securely (should use environment variables in production)

14. Why use CSS variables?

Answer: CSS variables allow easy theming and consistent styling. We can change colors globally by updating variable values. This makes it easy to implement dark mode (currently disabled but structure is there).

15. How does the loading animation work?

Answer: We show 4 steps (generating queries, searching, scraping, synthesizing) that update every 2 seconds. This gives users feedback about the search progress, even though we can't track exact backend progress in real-time.

□ Key Concepts to Remember

1. **REST API:** Flask provides REST endpoints that return JSON
2. **Asynchronous Operations:** JavaScript uses async/await for API calls
3. **Promise-based:** Supabase and fetch API return Promises
4. **Markdown:** AI generates markdown, converted to HTML for display
5. **CORS:** Needed when frontend and backend are on different origins
6. **Token Limits:** AI models have maximum token limits for input/output
7. **Rate Limiting:** APIs have usage limits (Google: 100/day)
8. **Web Scraping Ethics:** Should respect robots.txt and rate limits
9. **Authentication Flow:** Sign up → Create user → Initialize credits → Login
10. **State Management:** Frontend manages user state, credits, history

---

□ Study Tips

1. **Understand the Flow:** Be able to explain the complete search workflow
2. **Know Each Library:** Understand what each library does and why it's used
3. **Code Walkthrough:** Be ready to explain any part of the code
4. **Error Handling:** Know how errors are handled at each step
5. **Improvements:** Think about how you would improve the project
6. **Alternatives:** Know why you chose these technologies over alternatives
7. **Limitations:** Be honest about project limitations and how to address them

---

□ Additional Resources

- **Flask Documentation:** https://flask.palletsprojects.com/ (https://flask.palletsprojects.com/)
- **Supabase Documentation:** https://supabase.com/docs (https://supabase.com/docs)
- **Google Gemini API:** https://ai.google.dev/ (https://ai.google.dev/)
- **BeautifulSoup Documentation:** https://www.crummy.com/software/BeautifulSoup/ (https://www.crummy.com/software/BeautifulSoup/)
- **JavaScript Fetch API:** https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

Good luck with your viva! □