High Level Design (HLD)

Job Recommender System

Revision Number: 1.0

Last date of revision: 15/12/2023

# Document Version Control

| Date Issued | Version | Description | Author |
| --- | --- | --- | --- |
| 15/12/2023 | 1 | Initial HLD – v1.0 | Abraham Audu |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[Document Version Control 2](#_heading=h.gjdgxs)

[**Abstract 4**](#_heading=)

[**1. Introduction 5**](#_heading=)

[1.1 Why this High-Level Design Document? 5](#_heading=)

[1.2 Scope 5](#_heading=h.d1qeboog8101)

[**2. General Description 5**](#_heading=h.ggkmsajlo83h)

[2.1 Product Perspective 5](#_heading=h.47rs17phkyud)

[2.2 Problem Statement 5](#_heading=h.2m4jh7xphb00)

[2.3 Proposed Solution 5](#_heading=h.a24luzxqjj4y)

[2.4 Further Improvements 6](#_heading=h.9tipoxtfyl22)

[2.5 Technical Requirement 6](#_heading=h.6e6nbndd0z0)

[2.6 Data Requirements 6](#_heading=h.u8e5flsgmxuv)

[2.7 Tools Used 6](#_heading=h.eeubfnt9ay1j)

[2.8 Constraints 6](#_heading=h.uq8kg0k4hxoi)

[2.9 Assumptions 6](#_heading=h.jy1d8hqituqs)

[**3. Design Details 7**](#_heading=h.4o154eianxnu)

[3.1a Development Process Flow 7](#_heading=h.2izcvvqfv1k2)

[3.1b Deployment Process Flow 7](#_heading=h.gegq5e7jn42o)

[3.2 Event Log 7](#_heading=h.7vujpp5x3o0f)

[3.3 Error Handling 8](#_heading=h.754hode35qq)

[**4. Performance 8**](#_heading=h.83j129w5xneh)

[4.1 Reusability 8](#_heading=h.k9trcoggc87)

[4.2 Application Compatibility 8](#_heading=h.va0xe28dsx4t)

[4.3 Resource Utilisation 8](#_heading=h.yci7e8mbhjs5)

[4.4 Deployment 8](#_heading=h.rwu1iljkevt8)

[**5. Conclusion 8**](#_heading=h.j1a2hkradlcn)

# Abstract

Over the years, the job search market has expanded rapidly, and in recent times, most public job offerings find their way to online job websites and job boards. With the growing number of jobs and users on these platforms, it has become very important to make it easier for users to not only navigate through the thousands of available jobs, but also navigate through the jobs that will match their interests, skills and experience. In addition, the job search market consists of so many separate job search websites that it becomes a chore trying to find the right jobs. This project covers the implementation of a job search web application which collects jobs from multiple job websites and is able to recommend the best fitting jobs to users according to their profile information, recent activity on the app and the particular search query they input.

# Introduction

## Why this High-Level Design Document?

The purpose of this High-Level Design Document is to add necessary detail to this Job Recommender System project to serve as a suitable background for the coding aspect of this project. This document also captures conceptual design propositions to guide code requirements and interaction between modules.

This document will:

* Present design aspects and define them in detail
* Describe the user interface being implemented
* Describe the development hardware and software
* Include design features and the architecture of the project
* List and describe the non-functional attributes such as:
  + Security
  + Reliability
  + Maintainability
  + Portability
  + Reusability
  + Application compatibility
  + Resource utilisation
  + Serviceability

## 1.2 Scope

This document covers the structure of the system, including application architecture, application flow and technology architecture. Non-technical to mildly technical terms have been used in order to facilitate easy understanding of the contents of this document.

# 2. General Description

## 2.1 Product Perspective

The Job Recommender System solution is a deep learning-based recommender system to maximise the value derived by users of job search platforms by presenting contextually relevant search results to each user based on their unique needs.

## 2.2 Problem Statement

To create a recommender system which is able to use user profile information, work experience and recent activity to recommend the most relevant jobs when they make a search on the job search platform.

## 2.3 Proposed Solution

The solution proposed here is a web app which allows the user to create an account on the platform, update their profile and preferences, and then make searches. On the backend, the app will be able to scrape jobs from multiple job platforms and store them on a database, after which the jobs will then be vectorised and stored on a vector database to enable contextual search i.e document-based recommender system.

## 2.4 Further Improvements

When the user base grows substantially, a custom collaborative filtering model can be trained on the user data to predict user preferences based on the activity of other similar users.

## 2.5 Technical Requirement

The primary requirement is that the user is able to search for jobs and receive hits which fit their profile details, preferences and recent activity on the platform.

## 2.6 Data Requirements

The jobs data used on the platform will be scraped from job search platforms and job boards such as LinkedIn, Indeed and Jobberman. The data will consist of relevant fields such as job title, description, location, etc.

The data collected will also reference the source of the data, to ensure that all search results point back to the original source, as a means to ensure fairness in conduct. In addition, all jobs collected will be assigned unique identifiers to ensure that there are no duplicates on the database.

Finally, data collected will have an expiry date, to ensure that the database does not get clogged with outdated information.

## 2.7 Tools Used

Python and packages such as Selenium and BeautifulSoup were used in the web scraping phase. Other tools used are:

* Visual Studio Code as IDE
* Docker for building and managing containers
* Cassandra for the database
* Chroma for the vector database
* Sentence Transformers for the vector embedding model
* FastAPI for backend development
* Streamlit for frontend development
* Git/GitHub for version control and remote code backup
* PyTest for unit testing

## 2.8 Constraints

Lack of existing user interaction data makes document-based recommendation the most viable option as at this time.

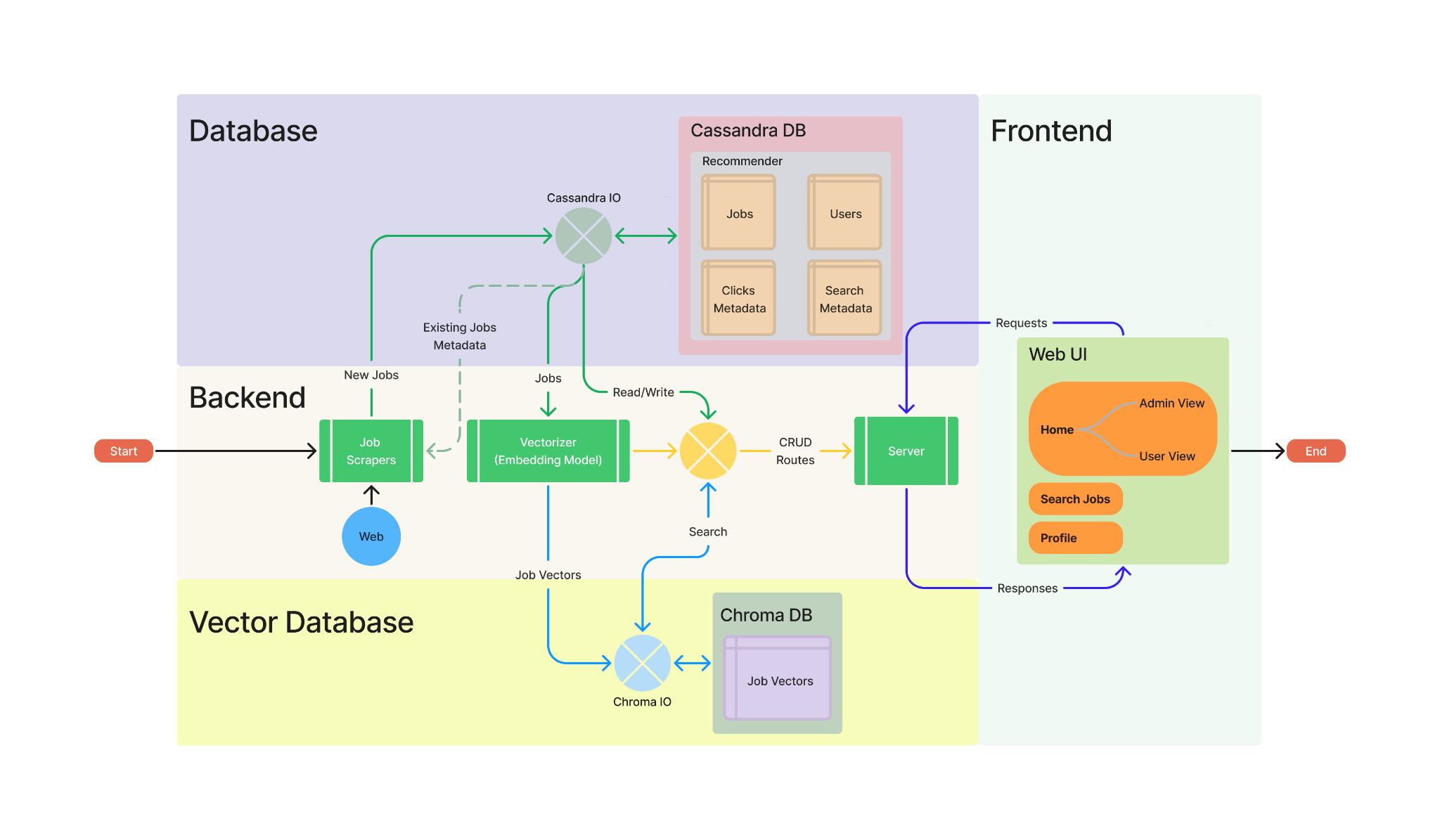
## 2.9 Assumptions

* The objective of the project remains as described in section 2.2
* Deep learning approach is required to achieve desired result
* The users will follow the profile setup instructions on the web app

# 3. Design Details

## 3.1 Process Flow

The diagram below illustrates the workflow of the Job Recommender System. Each shaded area represents a standalone docker container running as a microservice.



## 

## 3.2 Event Log

The solution developed has logging implemented at all levels from the data scraping pipeline, backend server process and frontend interface to monitor activity at different logging levels. The loggers for the different layers log information to separate files.

## 3.3 Error Handling

Exception handlers have been implemented in sections of the code base which have dynamic behaviour so as to catch errors and perform specific action to handle the error and keep the process running.

For fatal errors in the scraping pipeline and backend, the error messages are logged, and for errors in the frontend, errors are logged and an appropriate error message is relayed to the user interface for appropriate action to be taken.

# 4. Performance

The recommender system during testing was able to return search results within tens of milliseconds, which is considerably fast, as the lag is barely noticeable.

## 4.1 Reusability

Large portions of the code have been made reusable across the project. For example, all the scrapers follow an abstract base class template and only the methods requiring unique implementation for each website were modified. In addition, the input-output classes to read and write from the traditional and vector databases were reused across the project, as well as many other instances where reusability was prioritised.

## 4.2 Application Compatibility

The application, though developed on Ubuntu, has been designed to run from Docker containers which makes it not only platform agnostic, but also configuration agnostic and easier to scale.

## 4.3 Resource Utilisation

During testing with docker, the application was given access to 8 CPU cores (Intel i7-12650) and required 7.5GB of Docker vRAM to function smoothly. In addition, the total size of all the images came to 10GB of storage.

## 4.4 Deployment

To demonstrate the functionality of the solution built, it has been deployed locally using Docker with four containers: Cassandra, Chroma, Backend server and Frontend containers.

### 4.4.1 Local Deployment Details

Code Repository: <https://github.com/theabrahamaudu/Multi-Platfrom_Job_Recommender>

Refer to *README.md* on the code base for *Quick Start* steps.

### 4.4.2 Cloud Deployment

Due to the app requiring RAM >=7.5GB, no free hosting service can support it. Running a paid instance will cost around ~$32 to run on AWS EC2 t2.large for two weeks, around which this project might be evaluated, and I’m not able to fund it at this time.

# 5. Conclusion

The Credit Card Default Predictor app will take a credit card owner’s information and financial history and determine the probability that the credit card owner will default on their monthly card payment.