



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Facultat d'Informàtica de Barcelona



# CCBDA Final Project

Cloud jobs

MAY 29, 2022

Daniel Arias

David Gili

Jordi Bosch

Francesco Aristei

Samy Chouiti

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Requirements . . . . .	1
<b>2</b>	<b>Working methodology</b>	<b>2</b>
2.1	Organization . . . . .	2
2.2	Gantt diagram . . . . .	3
2.3	Task assignments . . . . .	3
2.4	Twelve-Factor Methodology . . . . .	4
2.4.1	Code Base . . . . .	4
2.4.2	Dependencies . . . . .	4
2.4.3	Configuration . . . . .	4
2.4.4	Backing Services . . . . .	4
2.4.5	Build, Release, Run . . . . .	4
2.4.6	Processes . . . . .	4
2.4.7	Port binding . . . . .	5
2.4.8	Concurrency . . . . .	5
2.4.9	Disposability . . . . .	5
2.4.10	Parity . . . . .	5
2.4.11	Logs . . . . .	5
2.4.12	Admin processes . . . . .	5
2.5	Additional tools . . . . .	6
<b>3</b>	<b>Technologies used</b>	<b>7</b>
3.1	Frontend and Backend . . . . .	7
3.1.1	Frontend . . . . .	7
3.1.2	Backend . . . . .	10
3.1.3	Difficulties and time spent . . . . .	13
3.2	Scraper . . . . .	13
3.3	Optical Character Recognition (OCR) . . . . .	15
3.3.1	AWS Textract . . . . .	15
3.4	CV and jobs matching . . . . .	17
3.4.1	Named Entity Recognition (NER) . . . . .	17
3.4.2	Issues . . . . .	18
3.5	Databases . . . . .	19
3.5.1	PostgreSQL database . . . . .	19
3.5.2	NoSQL Database . . . . .	20
<b>4</b>	<b>Architecture</b>	<b>23</b>
<b>5</b>	<b>Conclusion</b>	<b>26</b>

# 1 Introduction

The goal of this project is to create a platform that allows users to find their dream job by searching for the most relevant offers on the internet and then performing a matching of the requirements with the user's skills.

Most job listing websites require the user to complete a long on-boarding process after registering, in which information about the user, like the type of jobs he is interested, in are collected. Our key idea is to eliminate this process by allowing the user to upload their CV and then extracting the relevant information from it. Finally we perform a matching of the jobs with the user data displaying the top jobs that best suit the user's abilities.

## 1.1 Requirements

After having decided the project idea, we defined a list of requirements to be met in order to consider the project successful. We tried to define the specifications in a way that each of them were as atomic and concrete as possible. This way, all the specifications could then be translated into a list of tasks that would help organize the project.

- Obtain and store the job information from more than one data source.
- Built a scheduler that periodically runs the job-fetching task to keep the data as real-time as possible.
- Allow user to upload his CV and automatically parse relevant keywords from its content.
- Return the jobs that match the user qualities better. This is done by matching the job listing information with data extracted from the user's CV.
- Allow the user to save the most suitable jobs and to visualize them in his/her profile page after having logged-in.
- Website specifications:
  - Page that displays the job listings to the user.
  - Filter jobs by title, skills.
  - Save the jobs that the user has marked as interesting.
  - Button that allows the user to upload CV.
  - Second page that displays the jobs most suited with the skills extracted from the user CV.
  - Third page used for user login and authentication.

## 2 Working methodology

### 2.1 Organization

As we only had around 3 weeks to complete the project and the documentation, it was crucial to select a fitting organization and scheduling system. In order to maximize the usefulness of Thursday's meetings with the professor, we used a combination of the agile and Kanban methodologies.

The major advantage of following an agile approach is that we could subdivide the work into one-week sprints. After the week had passed, we would have a meeting that would work as a feedback loop. Each member, would present the tasks he had done up to that point. After that, we would discuss what could be improved and we would organize the tasks to be done for the next week

Each sprint will be internally organized using the Kanban approach. There are many websites to organize projects this way, but we decided to use [Notion](#) which provides different project management tools, as for example an editable Gantt diagram for the tasks.

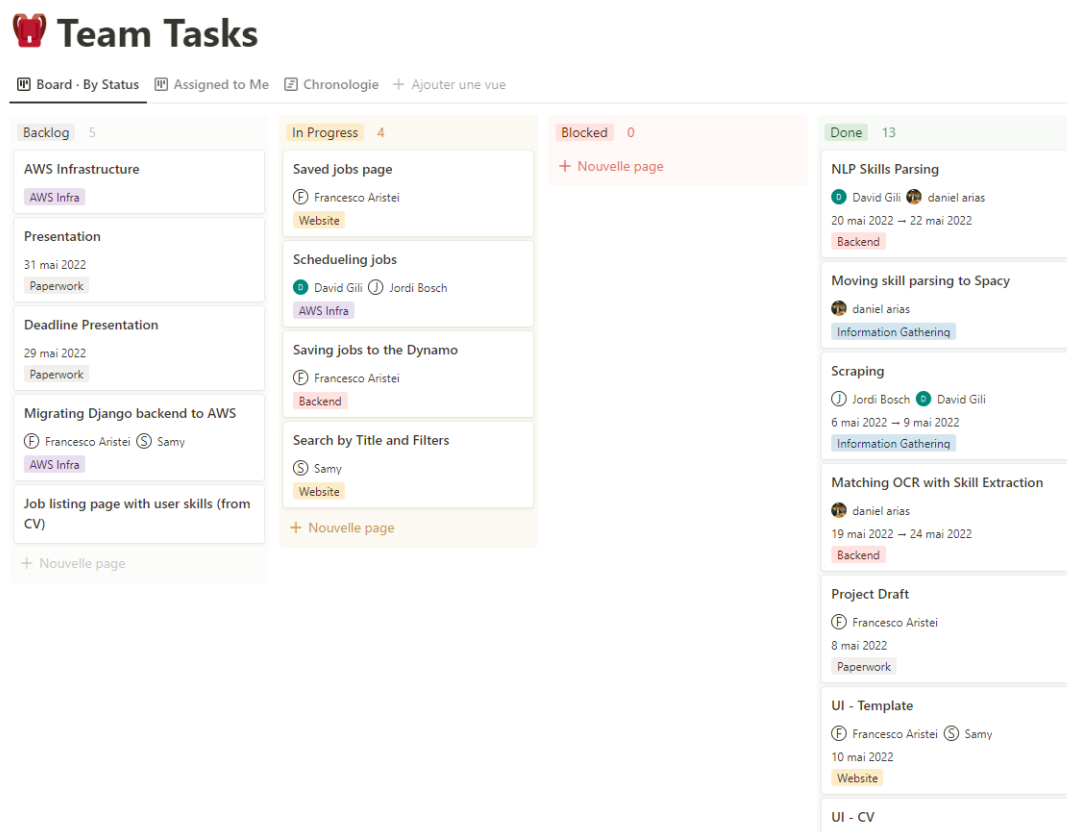


Figure 1: Tasks in Notion

## 2.2 Gantt diagram

When we first organized the project, we created a Gantt diagram to organize the tasks and ensure we would finish on time. This diagram was updated several times when new tasks appeared or when others suffered delays or blocks. Figure 2, show the final Gantt diagram were one can visualize the flow of tasks.

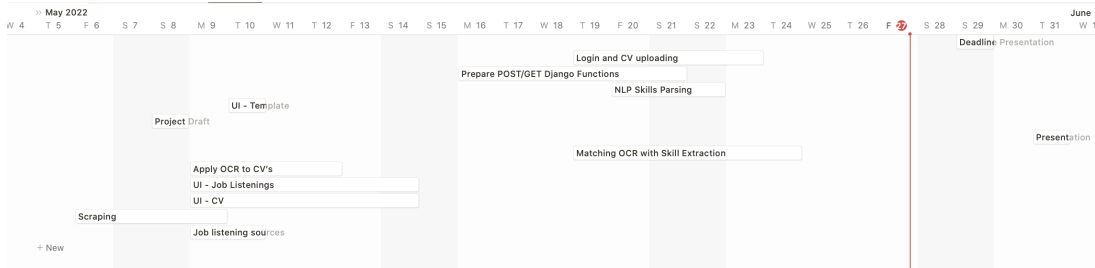


Figure 2: Gantt diagram

## 2.3 Task assignments

During the project, our methodology was to divide the tasks. Depending on the volume of work, we would assign one or two members to complete each task. In general, at the beginning of the project, the tasks were more atomic and could be done in parallel. However, to finish the tasks related to building the infrastructure and deploying it, nearly all the members needed to be involved. Table 1 shows the list of tasks and the members assigned to each of them.

	Jordi Bosch	Daniel Arias	David Gili	Francesco Aristei	Samy Chouiti
Scraper	✗		✗		
NER		✗	✗		
ORC		✗			
Frontend				✗	✗
Backend				✗	✗
Job scheduler	✗				
Documentation	✗	✗	✗	✗	✗
Presentation	✗	✗	✗	✗	✗

Table 1: Task distribution

## **2.4 Twelve-Factor Methodology**

Twelve-factor app is a methodology for building distributed applications that run in the cloud and are delivered as a service.

### **2.4.1 Code Base**

In this project, we used GitHub as a version control tool. It allows us to easily share the code between team members and also allows us to recover past versions of the code in case of a fatal mistake and pinpoint when some bug was introduced to the codebase. Specifically, we organized the work in branches. Once one task was finished, we reviewed the code and merged it into the master.

### **2.4.2 Dependencies**

For the explicitly declare and isolation of all dependencies, we used separate python environments for each part of the project and freeze all the libraries in a requirements file.

### **2.4.3 Configuration**

For the project configuration variables, we did not store any of them as constants in code. Instead, we design the app to consume them via environment variables.

### **2.4.4 Backing Services**

We treated back-end services as attached resources to be accessed with a URL or other locator stored in the configuration. The front-end consumes their services through REST API calls.

### **2.4.5 Build, Release, Run**

In order to be compliant with the separation between build and run stages, we create different branches from each main feature and the integration takes places in the main branch tagging it.

### **2.4.6 Processes**

We execute the app as one or more stateless processes. The data that must be persistent is stored in a stateful backing service such as Dynamo DB.

### **2.4.7 Port binding**

As a web app based project, we export HTTP as a service through port binding and listens to the requests coming from that port.

### **2.4.8 Concurrency**

The deployment of the different components of the application are separate and independent processes. These will allow upscaling and downscaling of components independently.

### **2.4.9 Disposability**

In order to be able to address the increase of load in the usage of the web page, as we deploy it in *AWS elastic beanstalk* it will quickly add more stateless processes as desirable.

### **2.4.10 Parity**

As web application, in order to be compliant in having the development, staging, and production as similar as possible we create a Linux virtual machine with the most similarity features as the instances that will run the production development.

### **2.4.11 Logs**

Logs emitted by an application provide visibility of his behavior. However, in cloud environments, you cannot reliably predict where your application is going to run. This makes it difficult to get visibility of those, we will need to treat application logging as a stream. Treating application logs as a stream would make it easier for other services to aggregate and archive log output for centralized viewing.

It could be achieved by specifying a custom logging configuration dictionary in the *settings.py*. Otherwise Django uses Python's built-in logging module. However, it was not part of the scope of the project, this would be one of the features to be added in future release.

### **2.4.12 Admin processes**

For admin tasks we run them as one-off processes, we have single endpoints that are only available for admin users.

## 2.5 Additional tools

To write the documentation, we used Overleaf, which is an online LaTeX editor which enables real-time collaboration between members. Additionally, we used Meet to perform the weekly online meetings.



## 3 Technologies used

### 3.1 Frontend and Backend

One of the very first part of such application is the Frontend and Backend, in order to integrate all others parts of the projects such as Scraping, NLP computations or user data management.

#### 3.1.1 Frontend

In order to conceive the UI of the web application, we used the Bootstrap framework, in its version 5. The main advantage of using such a framework is to be able quickly deliver ready-to-use responsive UI's.

We opted for a simple UI using the least amount of pages but to rather use multiple view of the same pages. The website in itself is made of:

- Main page: displaying latest scrapped jobs, with or without filtering by title of jobs or skills required
- Login page: to make able an user to login to use specific features
- Upload page: to upload the CV in order to extract skills for the job matching
- Profile page: to display jobs saved by the user

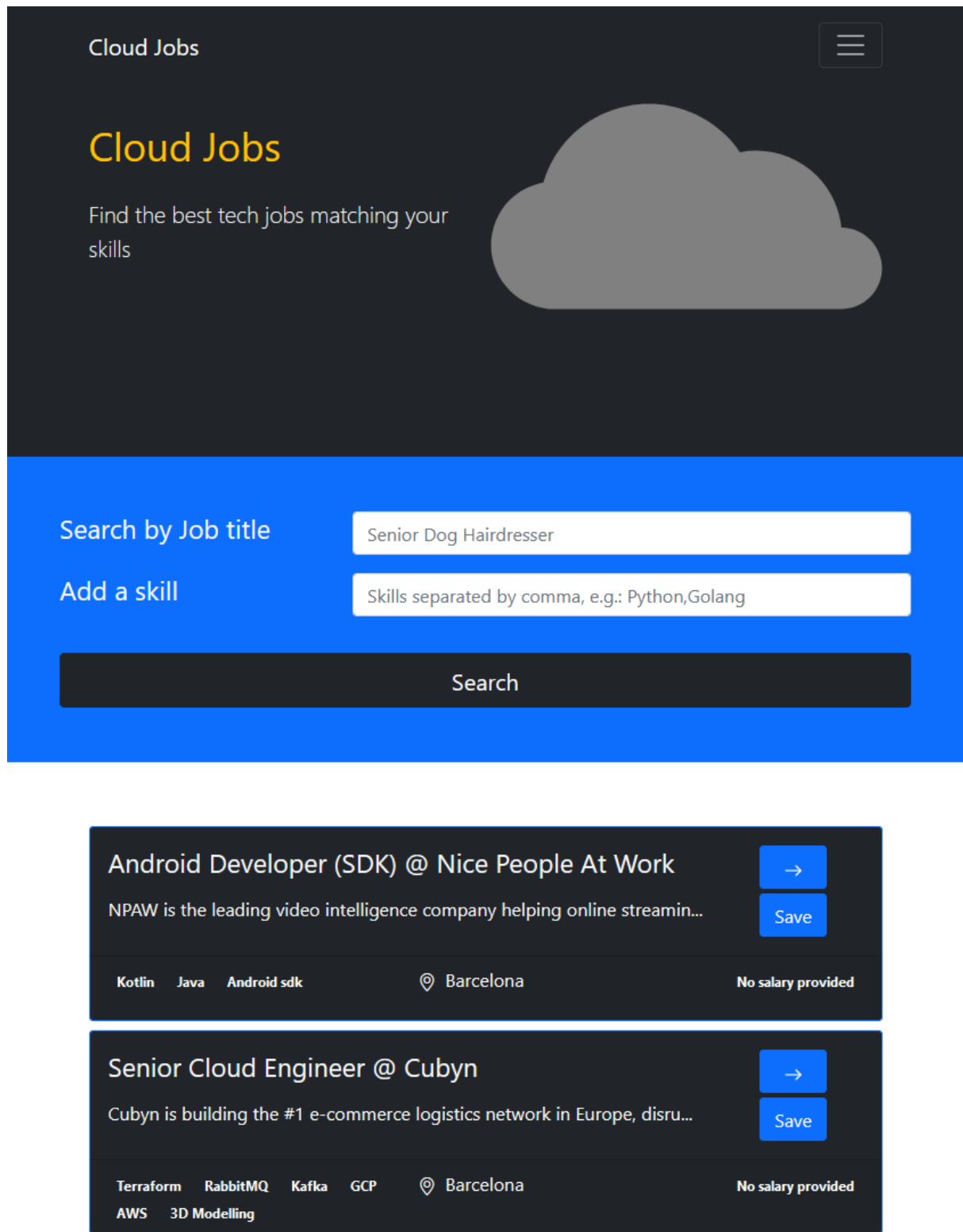


Figure 3: Main page on a small screen (e.g. mobile)

The UI contains more views, being variants of the pages presented above, which are:

- Index: Main page with unfiltered job listings
- Search: Main page with filtered job listings, by title and/or by skills

- Login: Login page, with login form
- Register: Login page, with registering form
- Upload: Upload page
- Profile: Profile page, which is a Main page variant with only saved jobs
- Save: Main page (without reload as using an AJAX call inside the HTML page)

In Figure 4, we can see the wireframe from the main page.

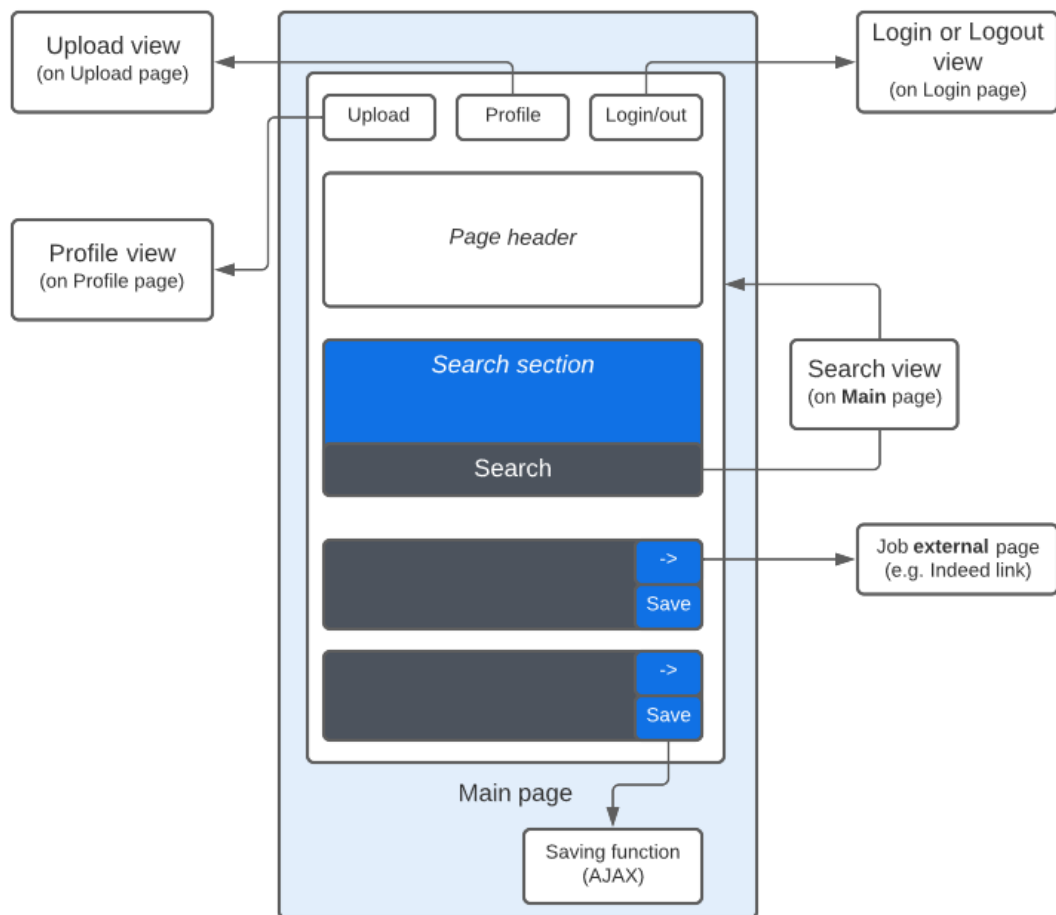


Figure 4: Wireframe of the main UI

### 3.1.2 Backend

#### 3.1.2.1 Introduction

For the backend, we used the Django framework as being simple and powerful, even for non-backend experts. As it is a MVC-based <sup>1</sup> framework, we used several views for the frontend above, but different models.

#### 3.1.2.2 Views of the application

We will describe the specifics of each views of the application in this part.

##### Index view

Basic view of the application. As is, displays the 5 most recently scrapped jobs. A job listing is in itself composed of:

- Job title
- Company
- Description of the position
- Skills required for the position (e.g. Python, AWS)
- Salary (if provided)
- Location (if provided)
- Link to the original listing (behind the arrow button)

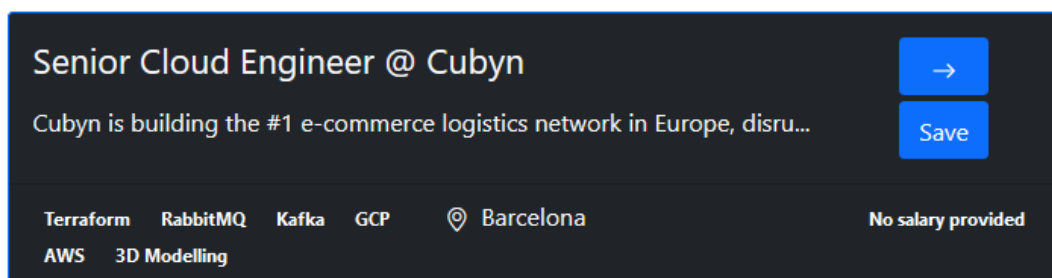


Figure 5: A job listing

##### Search view

The search view handles the feature of the same name that can be found on the main page. It is a non-case-sensitive search by job title and/or by multiple skills (exclusive search) in *Jobs*.

---

<sup>1</sup>Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. Source: MVC Wikipedia page

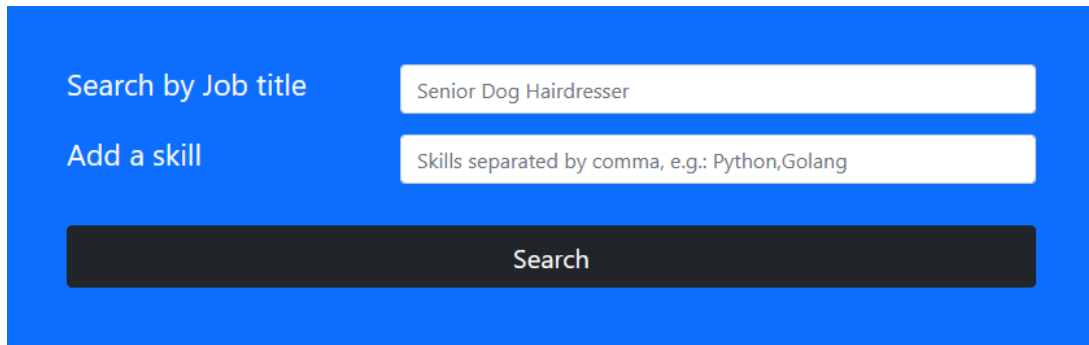

 A blue rectangular box containing a search interface. On the left, the text "Search by Job title" is positioned above a white input field containing "Senior Dog Hairdresser". Below this, the text "Add a skill" is positioned above another white input field containing "Skills separated by comma, e.g.: Python,Golang". At the bottom of the box is a wide, dark grey button with the word "Search" in white text.

Figure 6: Search section

### Upload view

Based on the Upload page, handles the upload of a PDF CV and calls the `Textract_CV` module (which itself calls NLP AWS services, as described later in the documentation).

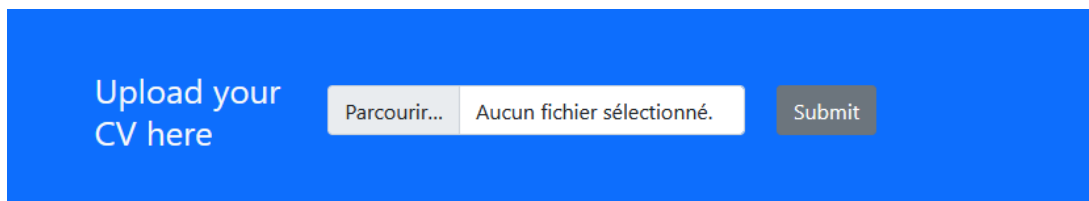

 A blue rectangular box containing an upload interface. On the left, the text "Upload your CV here" is displayed. To its right is a file selection area with a grey button labeled "Parcourir...", a white box containing the text "Aucun fichier sélectionné.", and a dark grey button labeled "Submit".

Figure 7: Upload section (on upload page)

### Login and logout views

It receive the user information as a POST request. After that it performs the log-in, checking the validity of the typed credentials. If the user types the correct information, is redirected to the main page, otherwise an error message appears.

### Register view

Handles the POST request the user makes when typing the credentials for the new account. It relies on the methods provided by the auth API.

### Save view

The save view is triggered once the user saves a specific job posting displayed in the main page. The view calls the `save_job()` method in the `models.py` which, using the `boto3` library, put a new item in the DynamoDB connected with the application. So that, when the user accesses his/her profile, the same jobs are retrieved from the DynamoDB, using `boto3`, and displayed. Allowing to keep track of the offers they would like to inspect better.

#### 3.1.2.3 Login features

To handle the user session, which comprises the login and registering part, we relied on the Authentication system provided by Django, specifically, we used the auth functions. The first tasks to perform for a new user, is the registration. As specified above, to handle this mechanism, we relied on the auth functions. Once the user has typed the credentials for the new account, the `User` object is called, which creates a new user in the PostgreSQL

database used to store such information. User objects are the core of the authentication system. They typically represent the people interacting with your site. Only one class of user exists in Django's authentication framework, i.e., 'superusers' or admin 'staff' users are just user objects with special attributes set, not different classes of user objects. The primary attributes of the default user are:

- Username
- Password
- Email
- First\_name
- Last\_name

After the registration has been completed, the user can easily log in and out of the website, using his/her credentials. This functionalities are possible thanks to the login mechanism provided by Django. The login procedure has been handled in the views, simply calling the `auth.login(username, password)` function provided by the `django.auth` API. Specifically the authenticated user is attached to the current session. The login function, takes an `HttpRequest` object and a `User` object then it saves the user's ID in the session, using Django's session framework. Therefore, the user's ID and the backend that was used for authentication are saved in the user's session. This allows the same authentication backend to fetch the user's details on a future request. Regarding the logout, a simple button is provided. Once clicked, the logout view is triggered, which, thanks to the `auth.logout()` method, logs the user out. Specifically, when you call `logout()`, the session data for the current request is completely cleaned out. All existing data is removed. This allows to prevent another person from using the same web browser to log in and have access to the previous user's session data.

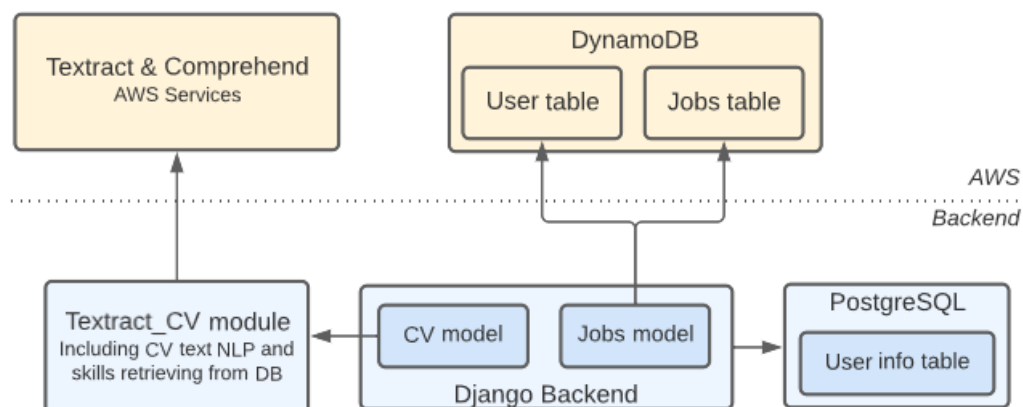


Figure 8: Backend interactions <sup>2</sup>

### 3.1.3 Difficulties and time spent

As not being previously trained to conceive the backend and the frontend, this project was a real opportunity for developing skills to do, in a limited range of time. However, the trial and error process might have increased the amount of work needed to achieve this part of the project, compared to what an educated developer could have provided.

More specifically, our surfaced knowledge of the fullstack technologies made the bug correction harder (e.g. fixing Javascript bugs without knowing about Javascript language or making the use of AJAX calls).

We (Francesco and Samy) estimate to **40** hours the time spent on coding, setting up and deploying the Frontend and Backend of the application, without taking into account the time spent for realizing the present documentation or presentation.

## 3.2 Scraper

One of the most relevant steps of the project was how to obtain the necessary data to meet the requirements. At first, we considered obtaining the information from an open databases or a publicly accessible API. The main problem with this first approach was the fact that the data would be probably old. Moreover, data would already be curated, excluding some potentially valuable information for our project.

As solution, we all realized that our website would benefit greatly from real-time data. Therefore, we decided that the best approach would have been to implement several scrapers to obtain data from different web listing websites. As we wanted data from multiple sources, we developed scrapers for [Indeed](#) and [Jobfluent](#).

Some of the problems we encountered when trying to scrape the data are the following:

- **Anti scrape measures:** Originally we tried to scrape the data from other popular job listing web pages like LinkedIn or remoteOk. This anti scrape measures can be rate limiting or based on the headers of the request (they can identify if you are a bot and return an error response).
- **Data fields:** Different websites display different information about the job listings. Therefore, we had to select some fields we would like to have preserve from each site and then perform a data unification process. In cases where some of the fields were not present in a job posting, we would substitute it with a null value.

On a more technical note, to scrape the information we used Scrapy, which is a web-crawler written in Python. After obtaining the jobs, we have built a pipeline that applies some preprocessing and then stores them in a DynamoDB table. After that, we just deployed the code using an AWS Lambda function **and used a scheduler that calls the function every day.**

Following, the reasons (pros and cons) of Why we used a lambda function for the job scraping task:

- Cons: Lambdas timeout; after a given time, if they have not finished the execution, they will timeout and fail to end the process. In the case of AWS the timeout is 900 seconds (15 minutes). Since our scraper in local was taking about 30 seconds, we could fit perfectly into the timeout, but if we had done a more powerful scraper or were looking into more websites, eventually this could not have been an option.
- Pro: Lambdas are the tiniest unit of on demand cloud processing power. Billing is extremely cheap because we only reserve machine power for the limited time that we the function is executing.
- Pro: Lambdas can be executed using cron jobs. This works perfect for us since we only need to create a cron job that executes our endpoint every day. We have decided to call the endpoint every day and not more because job boards don't change on a minutes/hours scale but on a days scale. Every day we would find 20 or so new cloud jobs in our boards, and we thought it was a good threshold.

Scheduler works by sending an periodically an event from CloudWatch to lambda. To set it up, we will use Amazon EventBridge, which is an module that introduces more functionality to the CloudWatch event APIS.

As we can see in the image, we can define some rules and the resources into which they apply. In our case we defined a rule that executes every day over our lambda function. We can see how we can introduce the cron job in the second image.

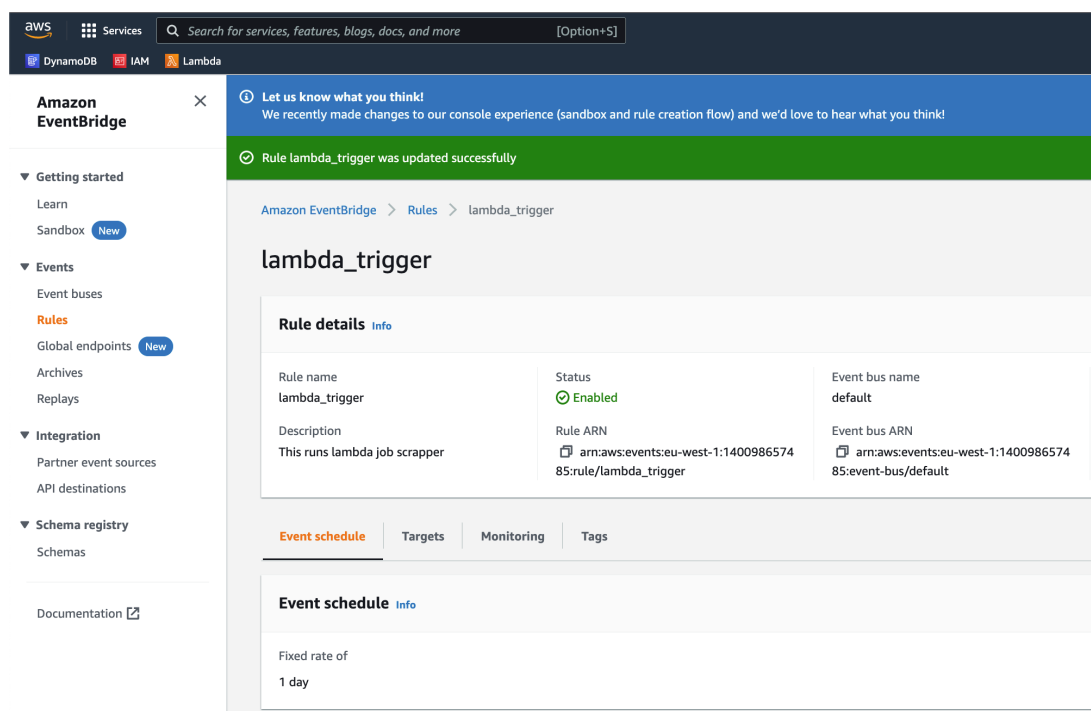


Figure 9: Trigger over lambda function



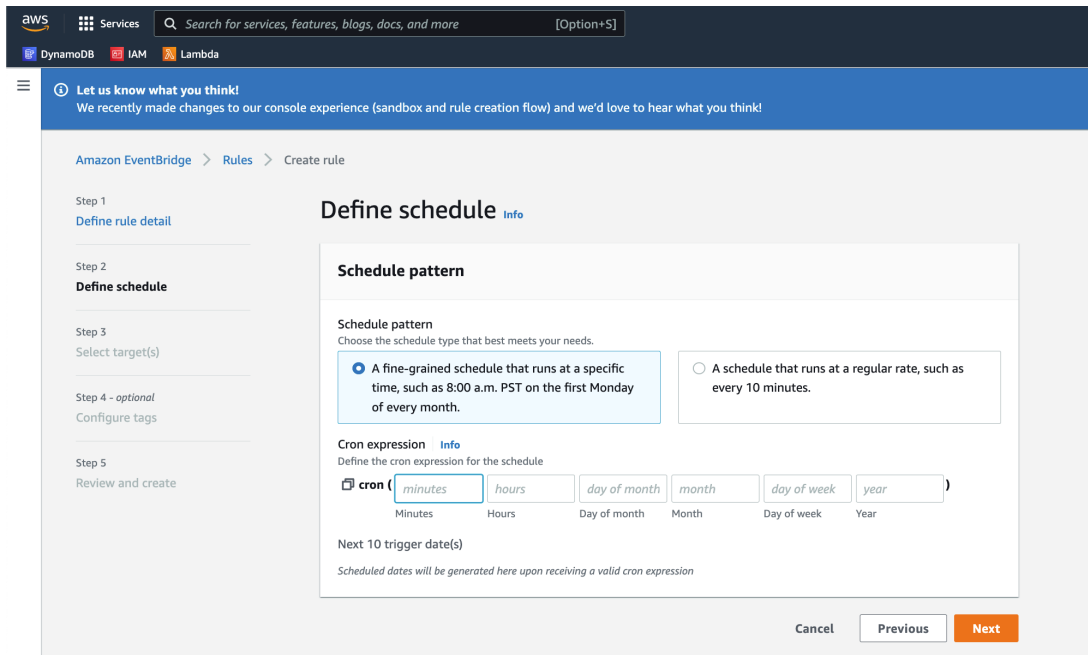


Figure 10: Cron job definition EventBridge

### 3.3 Optical Character Recognition (OCR)

Another requirement of our project was to read and extract the text from a CV uploaded by the user. To accomplish this goal, we considered two main solutions:

- **PDF reader:** We could use some Python library like PyPDF2 to extract the text from a pdf file. This solution is really fast and if the document is correctly saved as a pdf, the text will be extracted without problems.
- **AWS textract:** It is an OCR service provided by AWS that can extract text from files. This approach is undoubtedly more costly both in time and money. Nevertheless, the one advantage it has is that it allows us to extract the text from a more diverse set of files, such as pdf or even an image of a CV.

As CV files could be uploaded as images or PDF documents by the user, we need to extract the information to get the features needed to match the employee skills with the skills required by the jobs. We choose AWS Textract as it is a machine learning (ML) service that automatically extracts text, handwriting, and data from scanned documents. Textract uses ML to read and process any type of document, accurately extracting text, handwriting, tables, and other data without manual effort. This will allow us to quickly automate document processing.

#### 3.3.1 AWS Textract

AWS textract provides several alternatives to accomplish the goal of retrieving information from a document. Because of this, one of the main objective was to select the best strategy that accomplish the best results.

The following strategies are predefined models that were developed to tackle the most common text extraction use cases.

- Optical character recognition
- Handwriting recognition
- Form extraction
- Identity documents
- Table extraction
- Bounding boxes

Having into account the context of our problem where CV nowadays can be generated by a lot of web pages with several designs, we selected the *Optical character recognition*, *Form extraction* and *Bounding boxes* options.

### Optical character recognition

It provides synchronous and asynchronous operations that return only the text detected in a document such as the lines and words and their relationships.

### Form extraction

With it we can detect key-value pairs in document's images automatically and retain the context. For instance, in a CV, the field "First Name" will be the key and "David" his value. This makes it easier to import the extracted data into the database. Additionally, it will significantly improve the detection of the entities.

### Bounding boxes

For this one, all the extracted data is returned within box frames that enclose each piece of the text, such as a word, a line, a table, or individual cells within a table. This could be helpful if we would like to display and spot the skills extracted from the document of the user as displayed in the following image.

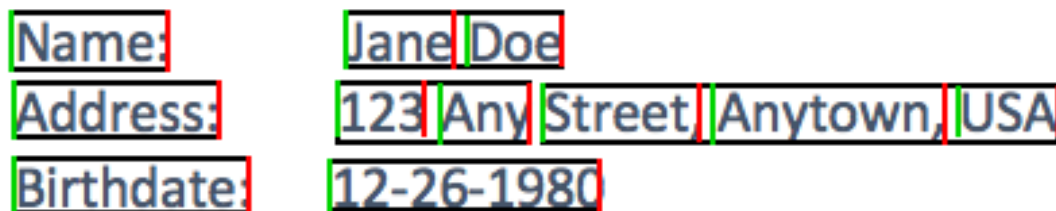


Figure 11: Bounding Boxes

On the whole, after analyzing the results with each one of the methods from the textract we decided to detect text synchronously, using the *DetectDocumentText* API operation. As it retrieves most of the lines and words from the documents we text with. The key-value pairs from the *Form extraction* could be used in a further step for storing user data that could allow us to create knowledge graphs and represent a network of users.

### 3.4 CV and jobs matching

The main distinct functionality of our website is being able to match the data from the user's CV with the data from job listings in order to offer to the user a curated list of jobs adequate for his skills. The main problem is that the contents of the CVs and jobs are just plain text and not tabular data. Thus, it is much harder to find a simple solution to match one with the other. We considered the following options:

- **Compare similarity of texts:** We could do the matching by just finding those jobs most similar to the content of the user's CV. This could be done using several similarity measures and preprocessing steps like TF-IDF or Doc2vec.
- **Compare keywords of both texts:** Instead of comparing the texts directly, we could first extract the relevant list of keywords from both texts and then simply compare those lists.

We realized that the best and simplest way to do this matching is to compare keywords of both texts. Nevertheless, this extraction is rather complex and could be tackled in different ways:

- **Predefined keyword list:** One option, we initially considered, was to have a fixed list of keywords and use regex functions to see if they appeared in the text. Nevertheless, we thought this option was rather limited, because we could only pick up those keywords that we had already predefined.
- **Named Entity Recognition (NER):** We thought the best approach would be to use a name-entity-recognition tool to obtain a list of relevant terms.

#### 3.4.1 Named Entity Recognition (NER)

Named-entity recognition (NER) is a subtask of data extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories like person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc.

Ousted **WeWork** founder **Adam Neumann** lists his **Manhattan** penthouse for **\$37.5 million**  
[organization] [person] [location] [monetary value]

Figure 12: Example of NER

We consider the use of Named-entity recognition (NER) as we are dealing with text analysis. In particular, we would like to extract information general information such as the locations, **companies**, **dates** and **skills** and more specific information like **programming languages**, **databases**, etc. To perform the name-entity-recognition, we considered the following solutions:

- **Spacy:** A Python library that performs a fast statistical based entity recognition. The biggest advantage is that it is free, but after testing it, we realized that it did not pick up most entities.
- **AWS Comprehend:** Service provided by AWS that uses trained machine learning models to perform entity recognition. From what we tested, it seems to perform the task quite accurately. The only disadvantage is that there is a small monetary cost for each call we perform.

We performed several tests to determine which technology was most adequate for the requirements of our project. In particular, we created a simple experiment which we would pass a big piece of text to both Spacy and AWS Comprehend. From this text, we extracted by hand the text entries related to computer science that were most useful for our needs. After that, we would obtain the list of entities from both services and check how many of the elements in our predefined set they included. The results showed that AWS Comprehend performed clearly better than Spacy at the task, as displayed in table 2.

Technology	Accuracy
Spacy	32 %
AWS Comprehend	65 %

Table 2: Comparison between Spacy and AWS Comprehend.

In the end, we decided it would be best to use AWS Comprehend, not only because of the better accuracy but also because after all the tests we had done, we only used a small amount of the free-tier version.

Each time a job is parsed or a CV is uploaded, we will perform a call to this service to obtain the entities present in its content. This service will return as response a list containing the entities, a category and a confidence score.

### 3.4.2 Issues

The main issue we encountered is that all the NER tools we tried could not classify words into more specific computer science categories like database or programming languages. For example, as shown in table 3 AWS will only classify words into 8 different labels.

Type	Description
Commercial_item	A branded product.
Date	A full date, day, month or time.
Event	An event, such as a festival, concert, election, etc.
Location	A specific location, such as a country, city, lake, building, etc.
Organization	Large organizations, such as a government, company, sports team.
Person	Individuals, groups of people, nicknames, fictional characters.
Quantity	A quantified amount, such as currency, percentages, numbers, bytes.
Other	Entities that don't fit into any of the other entity categories.

Table 3: AWS Comprehend types of entities

In order to correctly obtain and classify these more specific words, we considered these two solutions:

- **Build a custom NER model:** AWS Comprehend allow the user to build upon its models and [customize the type of entities](#) to be recognized. To do so, one just needs to have a large enough amount of annotated data to train the model.
- **Fix some keywords:** Another option would be to use some predefined lists of *programming Languages*, *databases*, *cloud*, *devOps* and *dataScience* terms and check if these words appear in the content.

In the end, although the first solution would be the most desirable, we do not have the type nor the data required to do it, which would include an extra step to preprocess the data and add the specific label for training the classification model to predict with better accuracy the entity, and therefore we chose the second solution.

## 3.5 Databases

A key step for the project was deciding how we were going to store the data. In particular, we needed to store the data of the jobs, users and CV. After a detailed analysis in which we studied which database paradigm would fit our requirements the best, we decided to use both relational (PostgreSQL) and NoSQL (DynamoDB) databases.

### 3.5.1 PostgreSQL database

As explained in the introductory section of the report, one of our main objective was to allow the users to create a personal profile, in order to save their preferred jobs together with the CV, to apply the skills matching. Therefore, we needed a secure login process, so that the user could access the website, without having his/her information compromised. To do so, we decided to rely on the API exposed by Django, specifically, we used the Django authentication system. Such system, has is key component in the User object, which, with few API calls allows to perform things like restricting the access, registering user profiles and more importantly, handling the user session. In this way we didn't have to implement a login system from scratch, with all the difficulties this imply. By default, Django provides a SQLite database, to store the user data (like the encrypted password, the username etc), however, One of the main drawbacks of the SQLite system is its lack of multi-user capabilities which can be found in full-fledged RDBMS systems like MySQL and PostgreSQL. This translates to a lack of granular access control, a friendly user management system, and security capabilities beyond encrypting the database file itself. This is a major drawback when designing multi-user applications. Therefore, we decided to rely on a PostgreSQL database instance to store all the user's data. This can be achieved with reduced effort simply modifying the settings.py file, specifying the DB engine to use with the needed parameters:

```

{
if 'RDS_DB_NAME' in os.environ:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql_psycopg2',
            'NAME': os.environ['RDS_DB_NAME'],
            'USER': os.environ['RDS_USERNAME'],
            'PASSWORD': os.environ['RDS_PASSWORD'],
            'HOST': os.environ['RDS_HOSTNAME'],
            'PORT': os.environ['RDS_PORT'],
        }
    }
else:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql',
            'NAME': 'cloudjobs_db',
            'USER': 'francesco',
            'PASSWORD': '',
            'HOST': 'localhost'
        }
    }
}

```

Listing 1: settings.py code to connect with Database

From now on, Django will store all the required information needed to handle the registering and login of the users in the PostgreSQL database specified. Specifically, different tables are created, each of which has the purpose of handling a different aspect of the session. From a code point of view, nothing has changed, the calls to the Django.auth functions are performed in the same way as if the SQLite database was still the main storing option. Once the process of connecting locally the DB instance to the app is completed, the next step needs to be addressed. Specifically, the purpose of the project is to deploy the app on AWS, therefore, also the PostgreSQL database, has to be deployed. AWS Beanstalk, which we used to integrate all the different technologies used in one single production environment, allows the integration of an RDS Database. The process is simple and clean, the EC2 instance in which the app will be deployed, exposes a number of environment variables with the needed details on how to connect to the Postgres server. Therefore, the only real change to the code we need to perform, is to add these variables in the settings.py file, in the section used to connect with the database.

### 3.5.2 NoSQL Database

We quickly realized that to store the jobs, the best option would have been to use a non-relational database. We arrived at this conclusion because our job object contains several lists, like the skills or the entities. If we had used a SQL database, these items would have needed to be stored in separate tables and thus several joins would have been

required to retrieve the information about jobs. By using a NoSQL table, we could keep the data denormalized. Moreover, this allowed us to decouple completely the user credentials from any other data. Therefore, we decided to rely on DynamoDB, a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale.

Figure 2 shows an example of how a job object look like. All the fields contain simple string values, except for the skills which are just a list of strings and the entities which contains a list of objects. Among all the fields, we decided to use the link field as primary key for our DynamoDB table because it is the only one that should be unique and can indentify an entry.

```
{
  "link": "https://www.indeed.com/rc/clk?jk=7275eb4391c9a0eb&fccid=9519b78d91690a4e&vjs=3",
  "title": "Entry Level Software Developer",
  "company": "Eagle Creek Software Services",
  "location": "Brookings, SD",
  "salary": "Estimated $57.6K - $73K a year",
  "description": "Position: Full-Stack Software Development Training. A Full-Time paid
  ↪ opportunity! ...",
  "content": " We are looking for people who want to learn Java, .Net and Salesforce
  ↪ development ...",
  "entities": [
    {"Score": 0.998830258846283, "Type": "ORGANIZATION", "Text": "Eagle Creek"},
    {"Score": 0.9247828125953674, "Type": "TITLE", "Text": "Java"}
  ]
}
```

Listing 2: Example of job object

Furthermore, we also needed to store information about the user. This included the attributes required for user authentication, the skills and information extracted from the CV, and the list of jobs the user had saved. Similarly, we thought it was best to keep the data as denormalized as possible, and therefore, we decided to also store it in DynamoDB.

```
{
  "id": ""
  "CV": {
    "entities": []
    "skills": []
  }
  "savedJobs": []
}
```

Listing 3: Example of user object

In order to retrieve and save the Jobs, from a code point of view, we used Boto3, an AWS SDK for Python to create, configure, and manage different AWS services apart

from DynamoDB, such as Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3). The SDK provides an object-oriented API as well as low-level access to AWS services. The boto3 calls have been performed directly inside the `model.py`, where the tables to be created in the DynamoDB are defined. This calls are triggered in the views, when the user search for a specific job topic, or decides to save an offer he/she finds interesting.



## 4 Architecture

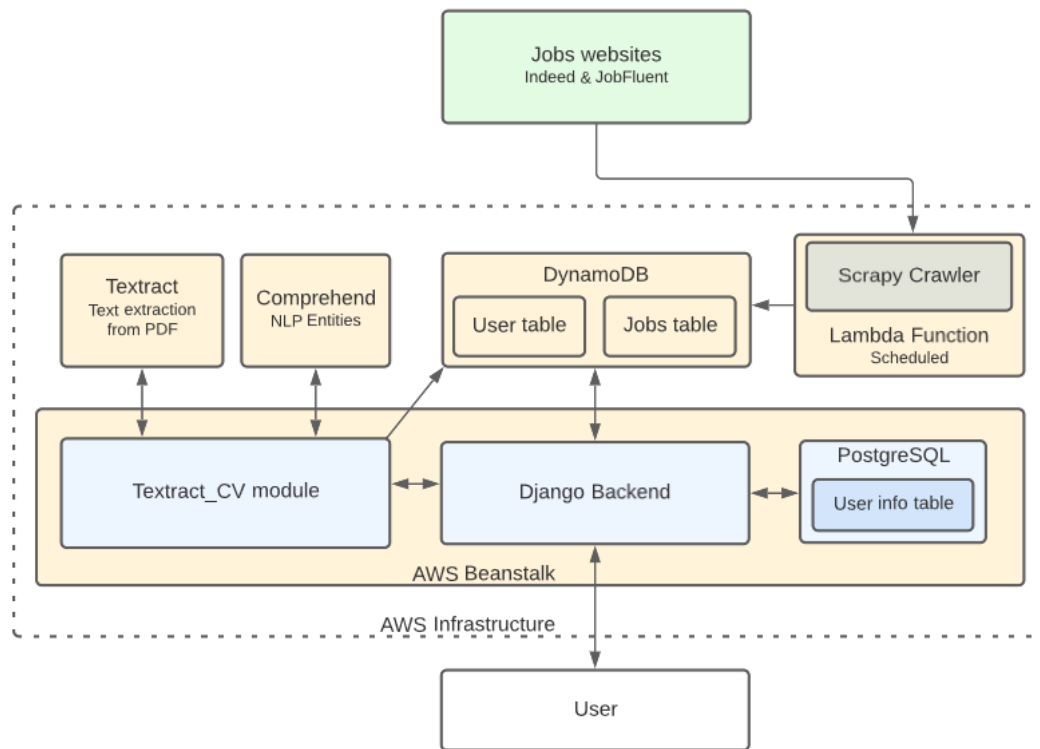


Figure 13: Architecture

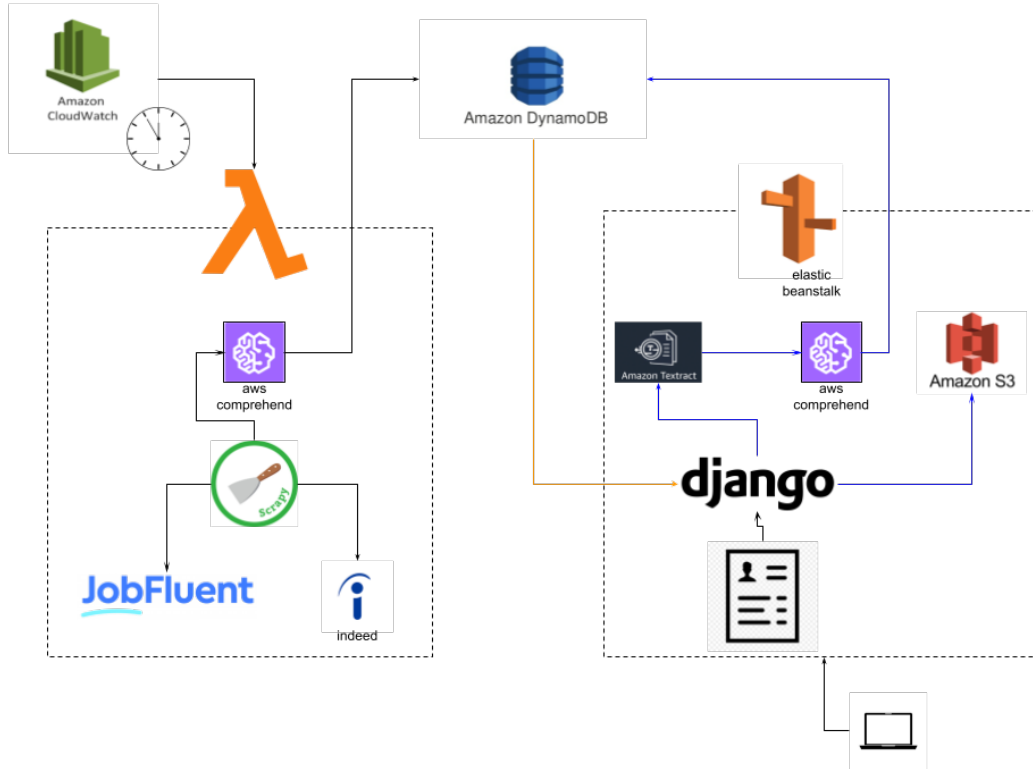


Figure 14: Architecture with logos

As displayed in fig. 14 we have a our project designed in two main flows. The first one is in charge of collecting the data from the job posts, scraping the web pages. The second one is related to collecting the job scraped, displaying them to the user and allowing the user to upload his/her information to execute the skills matching storing all the information in the *DynamoDB*.

The first flow is composed by the scraper (Scrapy) of the web pages *jobfluent* and *indeed*. This scraper is packaged as an *AWS Lambda* function executed via cron job with *AWS CloudWatch* events. Each one of the jobs retrieved go through the *AWS Comprehend* that will store the entities extracted in the *DynamoDB*.

The second part is the one that handles the user interaction with the web page. For this one we deploy our code in an Elastic Beanstalk taking into account the possibility of up scaling the web application. In this case, everything starts with the user registering in the website. Once he/she entered the needed information (username, password and email) a new user is created, storing all the data in the PostgreSQL instance deployed in the Beanstalk. After this, the user proceed with the log-in, which, once completed, allows the visualization of the main page, with all the extracted jobs. Specifically, with the main page creation, a python method is triggered, which contains the boto3 calls to retrieve the jobs stored in the scraping phase in the *DynamoDB*. The user can specify in a provided search bar, the main topic of the jobs he/she is looking for (Data Scientist, Software Engineer etc) displaying the best match between the topic searched and the jobs stored.

Finally, the web page, provides save button, which when clicked, saves the job offer in the user profile. Specifically, the job information are stored in the second *DynamoDB*,

together with the user id, to link such information with the user logged in. A profile page has been provided, in order for the user to visualize his/her saved job, retrieved from the same DynamoDB where they were saved before. The user has also the possibility of uploading his/her CV, which is then stored in an S3 bucket from where it will be retrieved and analyzed with the *AWS Textract* passing through the *AWS Comprehend*. The result entities are persisted in the *DynamoDB*.

The architecture design was chosen taking into account that these main flows mentioned above should be decoupled in order to avoid a single point of failure and considering that this will allow us to up scale in a better way each one of them without compromising the other. Also, the maintainability will be easier due to each flow independence.

One of the future improvements for this architecture would be to dissociate the entities detection from the *AWS Comprehend* given that the code used is the same in both main flows. This means that each time the functionality changes or is improved in one side it has to be updated in the other, potentially leading to an error where one of the parts is outdated. This could be fixed if we package the *AWS Comprehend* functionality in another endpoint to be executed by each one of the main flows.

## 5 Conclusion

Overall, we consider this project has been quite challenging because we lacked experience in some key areas like Frontend and UI design in general. Moreover, the limited time to complete the project has forced us to be quite diligent with task planning and working methodology. Despite this, we believe the finished product is a good prototype that meets all the requirements we wanted to accomplish.

With regard to the technologies, we ended up using several services like AWS Textract or AWS Comprehend that were not seen during the course. Moreover, we stored data in not only non-relational databases like DynamoDB but also SQL base databases like PostgreSQL. Furthermore, we expanded our knowledge of several topics already covered in class, for example, although scrappers were seen in one assignment, we manage to deploy the code to AWS lambda and used a scheduler to run it periodically.

There are several things we would have to improve given more time, specially some infrastructure automation, a custom NER model or a better CI/CD. Nevertheless, it was a team decision to prioritize handing in a functional product.

Although we learned a lot of new technologies, perhaps the biggest learning has been the ability to conceptualize an idea and then bring it to light not only in a local environment but in the cloud using the tools available in such a short time-span.