CLC – Final Project

Scalable Personality-Based Career Selection Web Application with Full-Stack Technologies: NextJs, Flask, PostgreSQL, Docker, Kubernetes, EKS AWS, and Locust.

Professor: Prof. Emiliano Casalicchio

Student:

1. Name: Arman
   Surname: Feili
   Matricola: 2101835
   Email: feili.2101835@studenti.uniroma1.it

Description of the Personality Website Project:

This project aimed to develop a scalable website for analyzing users' personalities using the MBTI and Big Five classifications, leveraging pre-trained models from Hugging Face. The website begins by asking users to complete 60 multiple-choice questions, followed by 26 open-ended questions. After the user finishes answering, the website generates a grammatically correct text from the user's responses.

This text is then analyzed to evaluate the user's personality using both the MBTI and Big Five classifications.

The MBTI (Myers-Briggs Type Indicator) categorizes individuals into 16 distinct personality types based on four key dichotomies:
- Introversion (I) vs. Extraversion (E)
- Sensing (S) vs. Intuition (N)
- Thinking (T) vs. Feeling (F)
- Judging (J) vs. Perceiving (P)

These categories help to understand how people perceive the world and make decisions. For the MBTI analysis, we used the "JanSt/albert-base-v2_mbti-classification" model from Hugging Face. The MBTI is beneficial as it provides insights into personal preferences and cognitive styles, which can enhance self-awareness, improve communication, and foster better relationships in both personal and professional settings.

The Big Five Personality Traits, also known as the Five Factor Model (FFM), assesses personality based on five major dimensions:
- Openness (inventive/curious vs. consistent/cautious)
- Conscientiousness (efficient/organized vs. easy-going/careless)
- Extraversion (outgoing/energetic vs. solitary/reserved)
- Agreeableness (friendly/compassionate vs. challenging/detached)
- Neuroticism (sensitive/nervous vs. resilient/confident)

These dimensions provide a comprehensive understanding of an individual's personality. For the Big Five analysis, we used the "Minej/bert-base-personality" model from Hugging Face. The Big Five is beneficial because it is a robust, empirically-based framework that predicts a range of important life outcomes, such as job performance, mental health, and interpersonal relationships. It offers a nuanced view of personality that can guide personal development and career choices.

The website displays the probability of each user's MBTI personality type in a pie chart and shows the probability of each user's Big Five personality traits in a bar chart. After determining the user's personality type, the website suggests suitable careers based on their personality. These career suggestions are ranked by average salary in the United States and presented in detailed tables. All generated data, including the user's text, MBTI and Big Five results, and suggested careers, are stored in a PostgreSQL database in a table named "personality_records".

Technical Implementation of the Personality Website Project:

To create a scalable project, we developed a full-stack web application using Next.js for the frontend, Flask for the backend, and PostgreSQL for database management. Next.js, a framework based on React.js, allows for server-side rendering and static site generation, enabling fast loading times and improved SEO. Flask, a lightweight Python framework, is ideal for building web applications and APIs. PostgreSQL, a powerful open-source relational database, efficiently handles our data storage needs.

We used Docker to containerize each part of the application into individual apps: FlaskApp, NextApp, and DB. Containerizing with Docker provides several benefits, including application isolation, consistent environments across different stages of development, and simplified deployment processes.

To ensure scalability and high availability, we used Kubernetes to manage and orchestrate our Docker containers. We created replicas of our Docker apps, setting a minimum of three replicas for each. This ensures that multiple instances of each app are running, providing load balancing and fault tolerance.

We then migrated our application to Amazon EKS (Elastic Kubernetes Service). EKS is a managed Kubernetes service that simplifies running Kubernetes on AWS without the need to handle the control plane or nodes manually. It provides scalability, security, and integration with other AWS services.

Within EKS, we created an EKS cluster named "eks-cluster-1". We assigned the user "eks-user" with comprehensive permissions, including 'AdministratorAccess', 'AmazonEC2ContainerRegistryFullAccess', and 'AmazonEKSClusterPolicy'. Additionally, we defined several IAM roles and assigned them to "eks-cluster-1" to manage various aspects of the service:

- AWSServiceRoleForAmazonEKS
- AWSServiceRoleForAmazonEKSNodegroup
- AWSServiceRoleForAmazonGuardDuty
- AWSServiceRoleForAmazonGuardDutyMalwareProtection
- AWSServiceRoleForAmazonPrometheusScraper
- AWSServiceRoleForAutoScaling
- AWSServiceRoleForCostOptimizationHub
- AWSServiceRoleForElasticLoadBalancing
- AWSServiceRoleForSupport
- AWSServiceRoleForTrustedAdvisor
- eksctl-eks-cluster-1-cluster-ServiceRole
- eksctl-eks-cluster-1-nodegroup-ng--NodeInstanceRole
- eksctl-personality-cluster-cluster-ServiceRole
- eksctl-personality-cluster-nodegro-NodeInstanceRole

The "eks-cluster-1" was responsible for creating EC2 instances, for which we selected the "m5.xlarge" instance type. The m5.xlarge instances are general-purpose instances that provide a balance of compute, memory, and networking resources. Each m5.xlarge instance has the following configuration:
- 4 vCPUs
- 16 GiB of RAM
- High-frequency Intel Xeon Platinum 8000 series processors

These instances host the EKS pods we defined. Each instance is responsible for managing the load and ensuring the application runs smoothly across all pods.

To achieve horizontal scalability, we configured the Horizontal Pod Autoscaler (HPA) to automatically add pods to the FlaskApp based on the available resources in AWS instances. This setup allows our application to scale dynamically and handle increased traffic efficiently.

For testing scalability, we used the Locust testing library to simulate user activity. We wrote a test script to send POST requests to our main API endpoint "/get_career_recommendations", which triggers all backend methods, retrieves responses, and saves records to the PostgreSQL database. This comprehensive test ensures all three Docker apps function correctly under load.

Finally, the application is live and accessible at the following URL:
http://ad0b4043cf0c74a9d99733004f9ad7bd-66423231.us-east-1.elb.amazonaws.com/

The Locust testing interface is available at:
http://a97b3a6e06cec478b87ac8e494c4b948-978139299.us-east-1.elb.amazonaws.com:8089/

Here is the directory of our project:

```
.
├── Kubernetes
│   ├── db-deployment.yaml
│   ├── flask-deployment.yaml
│   ├── locust-configmap.yaml
│   ├── locust-deployment.yaml
│   ├── next-deployment.yaml
│   ├── persistent-volume-claim.yaml
│   └── persistent-volume.yaml
├── backend
│   ├── api.py
│   ├── app.py
│   ├── datasets
│   │   ├── 5-big-traits-career.xlsx
│   │   └── MBTI-career.xlsx
│   ├── flask.dockerfile
│   ├── migrations
│   │   ├── README
│   │   ├── alembic.ini
│   │   ├── env.py
│   │   ├── script.py.mako
│   │   └── versions
│   ├── models.py
│   └── requirements.txt
├── compose.yml
├── frontend
│   ├── README.md
│   ├── next-env.d.ts
│   ├── next.config.js
│   ├── next.dockerfile
│   ├── package-lock.json
│   ├── package.json
│   ├── postcss.config.js
│   ├── public
│   │   ├── favicon.ico
│   │   ├── flasklogo.svg
│   │   ├── next.svg
│   │   └── vercel.svg
│   ├── src
│   │   ├── components
│   │   │   ├── Questionary.tsx
│   │   │   └── questionsAndOptions.tsx
│   │   ├── pages
│   │   │   ├── _app.tsx
│   │   │   ├── _document.tsx
│   │   │   └── index.tsx
│   │   └── styles
│   │       └── globals.css
│   ├── tailwind.config.ts
│   └── tsconfig.json
└── testing
    ├── locustfile.py
    └── questionsAndOptions.json

13 directories, 37 files
```

Issues and Challenges Faced

During the development of this project, we encountered several significant obstacles and challenges:

1. AWS Student Account Limitations:
Since the AWS student account had limitations on creating users for EKS, we had to register for AWS and use the free tier facilities provided for new users. Despite this, we have been charged over 100 euros so far, which presented an unexpected financial challenge.

2. Integrating Multiple Frameworks:
One of the primary challenges was ensuring that all the frameworks and technologies (Next.js, Flask, PostgreSQL, Docker, Kubernetes, EKS AWS, Locust) worked together seamlessly. Each component had to be configured correctly to avoid conflicts and ensure smooth operation.

3. Configuring Frameworks in Different Phases:
We had to configure each framework multiple times:
- During the development phase to ensure local functionality.
- During the Dockerization phase to containerize the applications correctly.
- During the migration to Kubernetes to manage and orchestrate the containers.
- During the migration to EKS AWS to leverage managed Kubernetes services.

4. AWS Configuration:
Configuring AWS services, including IAM roles, permissions, and networking settings, was crucial to ensure the system worked efficiently and securely. This required detailed attention to AWS best practices and thorough testing.

5. Optimizing YAML Configurations:
We had to update the YAML configuration files numerous times to determine the optimal settings for server configurations, the number of pods, and the amount of resources allocated to each pod. This was essential for achieving auto-scaling and maintaining system performance under varying loads.

6. Datasets for Career Recommendations:
Creating and providing datasets that map careers to MBTI and Big Five personality types was another challenge. This involved extensive research and data collection to ensure the recommendations were accurate and meaningful.

7. Generating Test Data:
We needed to provide a JSON file with randomized answers to the questions to test the application thoroughly. This was necessary to simulate different user inputs and ensure the system handled various scenarios correctly.

8. Managing Dependencies:
Ensuring compatibility between numerous Python and JavaScript packages was critical. We had to carefully manage dependencies to avoid version conflicts and ensure all packages worked together as intended.

Configuring Amazon Elastic Kubernetes Service (EKS) of AWS

To set up and deploy our project on Amazon Elastic Kubernetes Service (EKS), we performed several key steps. Below is an explanation of the commands used and their results.

1. Configure AWS CLI for EKS User

```bash
% aws configure --profile eks-user
```

This command configures the AWS CLI for the specified user profile (`eks-user`). The user is prompted to provide the following details:

```
- AWS Access Key ID: `AKIATCKAPF43PSJ4TVX6`
- AWS Secret Access Key: `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`
- Default region name: `us-east-1`
- Default output format: `json`
```

This step ensures that all subsequent AWS CLI commands are authenticated and authorized using the `eks-user` profile with the provided credentials.

2. Create the EKS Cluster using `eksctl`
~~~~
```bash
% eksctl create cluster --name eks-cluster-1 --region us-east-1 --nodes 3 --node-type m5.xlarge -
-profile eks-user
```
~~~~

This command creates an EKS cluster named `eks-cluster-1` in the `us-east-1` region. It sets up 3 nodes of type `m5.xlarge`. The `--profile eks-user` option specifies the AWS profile to use for authentication. This step sets up the infrastructure necessary to run Kubernetes workloads.

3. Configure `kubectl`

~~~~
```bash
aws eks --region us-east-1 update-kubeconfig --name eks-cluster-1 --profile eks-user
```
~~~~

This command configures `kubectl` to use the newly created EKS cluster (`eks-cluster-1`). It updates the Kubernetes configuration file with the cluster details, allowing `kubectl` to interact with the EKS cluster.

4. Create Namespace

~~~~
```bash
kubectl create namespace my-app
```
~~~~

This command creates a new namespace called `my-app` in the Kubernetes cluster. Namespaces are used to organize and manage resources within the cluster.

5. Deploy Persistent Volumes

~~~~
```bash
kubectl apply -f Kubernetes/persistent-volume.yaml
kubectl apply -f Kubernetes/persistent-volume-claim.yaml
```
~~~~

These commands deploy the persistent volumes and persistent volume claims defined in the respective YAML files. Persistent volumes provide durable storage for the applications running in the cluster.

6. Deploy Database, FlaskApp, NextApp

~~~~
```bash
kubectl apply -f Kubernetes/db-deployment.yaml
kubectl apply -f Kubernetes/flask-deployment.yaml
kubectl apply -f Kubernetes/next-deployment.yaml
```
~~~~

These commands deploy the database, Flask application, and Next.js application using the configurations specified in the YAML files. Each command creates the necessary pods, services, and other Kubernetes resources for each application component.

7. Verify Deployments

~~~~
```bash
kubectl get pods -n my-app
kubectl get svc -n my-app
```
~~~~

These commands list the pods and services in the `my-app` namespace, allowing us to verify that the deployments are running correctly.

Setting Environment Variables:

Backend Configuration

In `/backend/.env`:

```plaintext
DATABASE_URL=postgresql://postgres:postgres@db-service:5432/postgres
```

This environment variable sets the database connection URL for the Flask application. It specifies that the database is hosted at the `db-service` endpoint, using the PostgreSQL protocol.

Frontend Configuration

In `/frontend/.env.local`:

```plaintext
NEXT_PUBLIC_API_URL=http://a11043d4ebd004509aa9a413be69311b-689262011.us-east-
1.elb.amazonaws.com:4000
```

This environment variable sets the API URL for the Next.js application. It points to the public endpoint of the Flask application service.

Testing Connectivity

```bash
wget -qO- http://a11043d4ebd004509aa9a413be69311b-689262011.us-east-
1.elb.amazonaws.com:4000/health
```

This command sends a GET request to the Flask application's health check endpoint to verify that the service is up and running. The expected response is `OK`, indicating that the Flask application is functioning correctly.

Results from Running the Project on EKS AWS

After deploying the project on Amazon EKS, I used several `kubectl` commands to check the status and configuration of our deployments, pods, services, and nodes. Below are the details and explanations of the results:

1. Deployments:

```bash
kubectl get deployments --namespace=my-app
```

This command lists the deployments in the specified namespace (`my-app`).

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
db	3/3	3	3	43h
flaskapp	3/3	1	3	43h
locust-master	1/1	1	1	19h
locust-worker	2/2	2	2	19h
nextapp	3/3	3	3	43h

- **db:** The database deployment is running 3 replicas, all of which are up-to-date and available.

- **flaskapp:** The Flask application is also running 3 replicas. However, the `UP-TO-DATE` column shows only 1, indicating that a recent update might still be propagating.
- **locust-master and locust-worker:** These deployments are running as expected, with the master having 1 replica and the worker having 2 replicas.
- **nextapp:** The Next.js application has 3 replicas, all running smoothly.

2. **Pods:**

```bash
kubectl get pods --namespace=my-app
```

This command lists all the pods in the `my-app` namespace.

NAME	READY	STATUS	RESTARTS	AGE
db-744bb8d59f-gwk5d	1/1	Running	1 (26h ago)	26h
db-744bb8d59f-p9jxd	1/1	Running	1 (26h ago)	26h
db-744bb8d59f-pqvmc	1/1	Running	1 (26h ago)	26h
flaskapp-5b44d489cc-dzl6r	1/1	Running	3 (145m ago)	26h
flaskapp-5b44d489cc-nkhr2	1/1	Running	3 (145m ago)	26h
flaskapp-5b44d489cc-z2ktv	1/1	Running	3 (143m ago)	26h
flaskapp-5b4cd95d77-ghnd6	0/1	Running	17 (77s ago)	154m
locust-master-85c46f77d4-gglwc	1/1	Running	0	19h
locust-worker-767cc68cf5-bdnp6	1/1	Running	0	19h
locust-worker-767cc68cf5-k2xl7	1/1	Running	0	19h
nextapp-798769797c-lb29z	1/1	Running	0	26h
nextapp-798769797c-q8znc	1/1	Running	0	26h
nextapp-798769797c-tmm9s	1/1	Running	0	26h

- Most pods are in the `Running` status, indicating they are functioning correctly.
- The `flaskapp-5b4cd95d77-ghnd6` pod shows multiple restarts due to insufficient resources.

3. Services:

```bash
kubectl get services --namespace=my-app
```

This command lists all services in the `my-app` namespace.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
db-service	ClusterIP	10.100.48.227	<none>	5432/TCP	43h
flaskapp-service	LoadBalancer	10.100.217.16	a11043d4ebd004509aa9a413be69311b-689262011.us-east-1.elb.amazonaws.com	4000:31170/TCP	43h
locust-master-service	LoadBalancer	10.100.251.116	a97b3a6e06cec478b87ac8e494c4b948-978139299.us-east-1.elb.amazonaws.com	8089:32555/TCP,5557:30706/TCP	19h
nextapp-service	LoadBalancer	10.100.82.46	ad0b4043cf0c74a9d99733004f9ad7bd-66423231.us-east-1.elb.amazonaws.com	80:30301/TCP	43h

- db-service: A ClusterIP service for internal database communication.
- flaskapp-service: A LoadBalancer service exposing the Flask app to the internet.
- locust-master-service: A LoadBalancer service for the Locust master, allowing access for load testing.
- nextapp-service: A LoadBalancer service exposing the Next.js frontend to the internet.

4. Nodes:

```bash
kubectl get nodes
```
This command lists all nodes in the Kubernetes cluster.

NAME	STATUS	ROLES	AGE	VERSION
ip-192-168-19-213.ec2.internal	Ready	<none>	43h	v1.30.0-eks-036c24b
ip-192-168-34-219.ec2.internal	Ready	<none>	43h	v1.30.0-eks-036c24b
ip-192-168-57-160.ec2.internal	Ready	<none>	43h	v1.30.0-eks-036c24b

- All nodes are in the `Ready` state, indicating they are functioning correctly and can accept workloads.

- Each node has the version `v1.30.0-eks-036c24b`, which is the Kubernetes version managed by EKS.

The project is functioning smoothly overall. We tested the project using Locust and assigned 1000 users to simulate load. Once the number of users exceeded 700, EKS automatically scaled the Pods horizontally. This scaling action generated an additional pod for FlaskApp, increasing the total number of FlaskApp pods to 4.

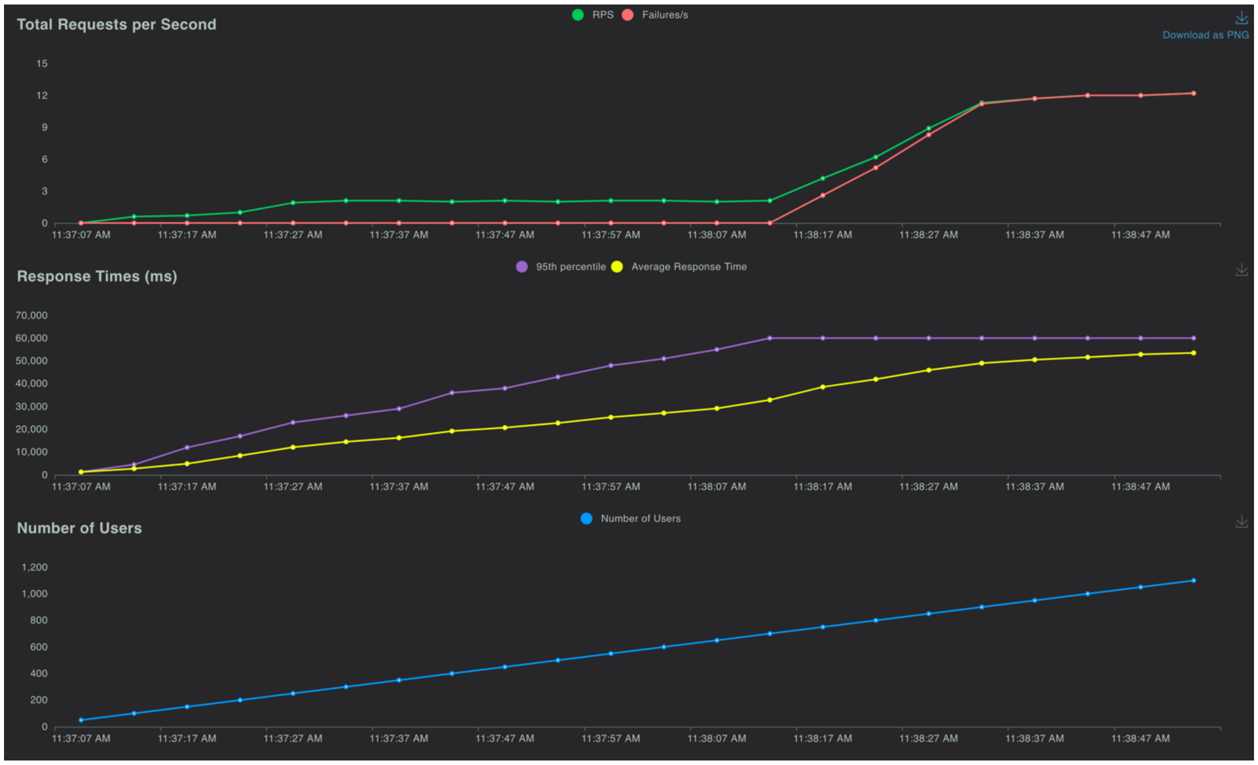
However, due to insufficient RAM and CPU resources in the "m5.xlarge" instances, the fourth pod repeatedly crashed and restarted. The "m5.xlarge" instances have the following configuration:
- 4 vCPUs
- 16 GiB of RAM

The fourth pod, `flaskapp-5b4cd95d77-ghnd6`, is shown to be created but is constantly restarting because the instances lack sufficient resources to support it.

To resolve this, we could define stronger instances, such as "m5.2xlarge," which offer more CPU and RAM. However, we did not opt for this solution due to the high hourly cost of each AWS instance.

By addressing these resource limitations, we can ensure that the project scales effectively to handle increased user load without encountering resource shortages.

Analysis of Locust Test Results



The Locust test results provide insights into the performance and scalability of the application under load. Below is an analysis of the key metrics and findings based on the provided screenshots.

The graphs provide a visual representation of the system's performance under load:

1. Total Requests per Second:

- The green line represents the Requests per Second (RPS).
- The red line represents the number of failures per second.
- Up to around 700 users, the RPS steadily increases, indicating that the system can handle the growing number of requests smoothly.
- Beyond 700 users, the failure rate starts to rise sharply, matching the RPS. This indicates that the system begins to struggle with the load, resulting in a significant number of failed requests.

2. Response Times (ms):

- The yellow line represents the average response time.
- The purple line represents the 95th percentile response time.
- Both response times show a steady increase as the number of users increases, indicating that the system is taking longer to process requests as the load grows.
- The steep increase in response times beyond 700 users highlights the system's struggle to maintain performance under high load.

3. Number of Users:

- The blue line shows the number of simulated users, which increases linearly throughout the test.
- The system maintains performance up to 700 users, after which performance degrades, as indicated by the increasing response times and failure rates.

Conclusion:

The system works fine and smoothly up to 700 users. Although the system is configured correctly and is scalable, with requests being sent to all FlaskApp pods based on the LoadBalancer configuration, due to insufficient resources and budget limitations, we could not scale the project vertically. We were able to scale the project horizontally by creating more pods, up to 10 for FlaskApp, but these new pods also require more resources.

There are some ways to improve this project:

- Using stronger instances like `m5.2xlarge`, which have more CPU and RAM, can provide the additional resources needed to handle increased loads more effectively.
- Adding more instances to the project to handle the increased load. This approach would require additional budget but is necessary for maintaining performance under high load.

By addressing these procedures, the system can be made more resilient and capable of handling higher loads. However, both increasing instance sizes and adding more instances will cause higher costs.