Automation and Computing
Machine Learning
Darie-Teofil Glanda
Cofster
2023-2024

Universitatea Politehnica din Timișoara

# COFSTER: ARCHITECTURAL FRAMEWORK FOR AN INTELLIGENT COFFEE SYSTEM USING CV AND NLP

**Candidate: Darie-Teofil Glanda**

**Scientific coordinator: Assoc. Prof. dr. Răzvan Bogdan**

Session: June 2024

Automation and Computing
Machine Learning
Darie-Teofil Glanda
Cofster
2023-2024

Universitatea Politehnica din Timișoara

# TABLE OF CONTENTS

Universitatea Politehnica din Timișoara

## List of tables



## List of figures

Automation and Computing
Machine Learning
Darie-Teofil Glanda
Cofster
2023-2024

Universitatea Politehnica din Timișoara

# ABSTRACT

Cofster, an innovative coffee-making system, aims to redefine the coffee creation experience by introducing a seamless and autonomous operation, eliminating the need for manual intervention. The primary focus of this project involves the integration of cutting-edge technologies and architectures into the coffee-making process, resulting in a sophisticated system that enhances user control and overall satisfaction.

At its core, Cofster features a mobile application built with the Flutter framework. The app allows users to place orders, make payments, create custom drinks (manually or via a LangChain agent), check order history, view active orders, collect gifts, and navigate using voice commands.

A key feature of Cofster is its use of Computer Vision and Natural Language Processing. State-of-the-art detectors, pre-trained on the MS COCO dataset and then further trained for the custom cup detection task, enable the system to autonomously identify custom cups and their sizes (small, medium, large), to initiate the coffee-making process. By using a region of interest (ROI) to detect the drip area, Cofster ensures precise cup placement. An algorithm was adopted to detect a rectangle with a specific area and color, to confirm the drip area is correctly aligned with the coffee cup. Cofster also includes a recommendation system with dual objectives: displaying the top k favorite drinks in the mobile app and sending a free gift upon account creation. The LangChain agent uses different versions of GPT and the RAG method to combine stored data with the LLM, continuously updating the system based on user preferences gathered through a response form for personalized drink choices.

Cofster's operational efficiency is achieved with a mix of on-premise servers using microservice architecture and a serverless approach on AWS Cloud, enhancing scalability and resource utilization.

For more details, regarding the implementation, please visit: https://github.com/glandaDarie/Cofster.

# ABSTRACT

Cofster, un sistem inovator de preparare a cafelei, își propune să definească experiența de creare a cafelei prin introducerea unei operațiuni autonome, eliminând necesitatea intervenției umane. Principalul obiectiv al acestui proiect implică integrarea tehnologiilor și arhitecturilor de ultimă generație în procesul de preparare a cafelei, rezultând un sistem sofisticat care îmbunătățește controlul utilizatorului și satisfacția generală.

În esență, Cofster dispune de o aplicație mobilă construită cu framework-ul Flutter. Aplicația permite utilizatorilor să plaseze comenzi, să efectueze plăți, să creeze băuturi personalizate (manual sau printr-un agent LangChain), să verifice istoricul comenzilor, să vizualizeze comenzile active, să colecteze cadouri și să navigheze folosind comenzi vocale.

O caracteristică importantă al framework-ului Cofster, este utilizarea de Computer Vision și Natural Language Processing. Detectoare de ultimă oră, pre antrenate pe setul de date MS COCO și apoi antrenate suplimentar pentru sarcina de detectare a ceștilor personalizate, permit sistemului să identifice autonom ceștile personalizate și dimensiunile acestora (mică, medie, mare) pentru a iniția procesul de preparare a cafelei. Prin utilizarea unei regiuni de interes (ROI), pentru a detecta zona de picurare, Cofster asigură o plasare precisă a ceștii. A fost adoptat un algoritm pentru a detecta un dreptunghi cu o anumită zonă și culoare, pentru a confirma că zona de picurare este corect aliniată cu ceașca de cafea. Cofster include, de asemenea, un sistem de recomandare, care are două obiective: afișarea băuturilor favorite în aplicația mobilă și trimiterea unui cadou gratuit la crearea contului. Agentul LangChain folosește diferite versiuni de GPT și metoda RAG pentru a combina datele stocate cu LLM, actualizând continuu sistemul pe baza preferințelor utilizatorilor colectate printr-un formular de răspuns pentru alegeri personalizate de băuturi.

Eficiența operațională a Cofster este realizată printr-un mix de servere locale folosind arhitectura microserviciilor și o abordare fără server pe AWS Cloud, îmbunătățind scalabilitatea și utilizarea resurselor.

Pentru mai multe detalii, referitoare la implementare, vă rog să vizitați https://github.com/glandaDarie/Cofster.

# 1. INTRODUCTION

## 1.1 CONTEXT

In today's rapidly advancing technological landscape, the primary aim is clear: automation of diverse human tasks. This paradigm promises to lift individuals from manual laborers to a more promminent role, as orchestrators, guiding operations rather than merely executing them. With the help of AI (Artificial Intelligence), this transformation becomes easier than ever to achieve. Tasks that were once deemed labor-intensive are now effortlessly accomplished. [1]

From smart homes to self-driving vehicles, the surge of automated systems, utilizing AI, redefine how we perceive and interact with technology. Among these innovations, the creation of a smart coffee system would open doors to a new era for coffee stores, eliminating the necessity for additional personnel, removing labor work, enhancing automation. [2]

In the scheme of those things, the development of Cofster, which is an architectural framework integrating cutting-edge technologies, heralds a new era for coffee stores. This framework enhances operational efficiency, but it also marks a new leap towards a technologically-driven future in the realm of coffee creation.

## 1.2 MOTIVATION OF THESIS

In today’s fast-paced world, where time is the most valuable asset, individuals find themselves unable to afford the luxury of spending time waiting for their coffee to be prepared. During this period, they are obligated to engage in other activities, in order to not lose focus, as well as time. Having said that, they would prefer to have a system that notifies them when the coffee is ready, in order for them to integrate the coffee collection into their busy schedule.

There is also an additional problem with coffee stores nowadays, is that they need to hire too much personnel. This arises from the fact that they do not use automation in any way shape or form, making the coffee making process labor-intensive, where the focus still lies on manual labor.

Therefore, I proposed a coffee making process, Cofster, to handle those two problems right there.

## 1.3 LIMITATIONS

The coffee recommender model has its limitations in the context that it can be further trained for more epochs. There can also be an implementation of an algorithm in the form of K-fold-cross-validation, in order to find the best hyperparameters for the given model. It could also benefit from a better Neural Network architecture, by adding more linear layers. This on the flip side, would cause both the training and the inference to be slower, making the forward and the backwards pass more resource intensive. There is also one more thing that can be enhanced, and this would be on the data collection part, where we could probably collect more real samples, from real people, instead of generating them with a Large Language Model.

Room for improvement can also be done on the voice to text converter pipeline, where I use the Levenshtein with padding algorithm for comparing the correct coffee drink with the one we receive from the voice to text converter. A better approach would be to use word embeddings, and calculate the similarity score of the actual word we are getting, with every coffee that I have in store, instead of using the approach that I went with initially.

On the detection pipeline, although not many, there are still some limitations there as well. When we try to detect the drip area, we don't use Deep Neural Networks, like when trying to detect the cup, our implementation simply relies on detecting rectangles with a certain area size and color, by applying thresholding. This can lead to issues later on when we try to use our system with other coffee machines that don't strictly have the hardware implementation like our own one.

On the hardware side, a significant limitation stems from using a camera connected to our server via a cable. Additionally, our coffee machine prototype currently has a simple architecture, restricting its capability to produce only cold and not hot coffee.

# 2. STATE-OF-THE-ART

In a multitude of studies, the integration of advanced detector models, particularly those belonging to the YOLO (You Only Look Once) family, has become a primary approach for conducting object detection tasks on edge devices, where inference processes are executed locally on the device [3], [4], [5], [6]. For example, in the study referenced as [4], the authors developed and proposed a real-time dog detection model. This model is capable of sending email notifications by utilizing Embedded Machine Learning (Embedded ML) techniques, and it performs inference operations on a Raspberry Pi, which is a widely-used, low-cost edge computing device. This particular study stands in contrast to my approach, which employs a microservice architecture to decouple the task of cup detection from the rest of the backend system. By doing so, it allows for the utilization of a Graphics Processing Unit (GPU) for model inference, thereby enhancing performance and efficiency.

Generally, older versions of the YOLO model are preferred in various implementations due to their proven reliability and stability over time. For instance, in the paper denoted as [7], the authors implemented the YOLOv5 model to develop an automatic detection system for defects in coffee beans. Similarly, the YOLOv5 model was employed in the study cited as [8], where it was used for the purposes of image classification and recognition within the domain of traditional Chinese medicine. Another example can be found in the paper referenced as [9], where the YOLOv3 architecture was utilized alongside the darknet framework to accomplish underwater object detection tasks. In my own work, however, I have adopted the latest iterations of the YOLO model family, specifically YOLOv8 and YOLOv9, to undertake the task of custom cup detection, leveraging their advanced capabilities and improved performance metrics.

When constructing a model designed to predict users' top k favorite drinks based on responses to a set of questionnaire items, with each question corresponding to a specific column in a database, previous implementations have typically favored the Boosting strategy. For instance, in the study cited as [10], the authors utilized the XGBoost algorithm to predict outcomes in a video game, which is a task bearing significant similarity to the predictive model I am developing. In a departure from this conventional approach, my methodology emphasizes the creation of a Multiple Layer Perceptron (MLP), thereby exploring a relatively unconventional approach for accomplishing this task.

To facilitate the viewing of the camera's perspective, I devised a sophisticated pipeline that centers around the use of OpenCV, a robust open-source computer vision and machine learning software library. This pipeline is integrated with a Kafka producer, which features a single partition and a single replica. Additionally, the pipeline includes a Kafka consumer, WebSocket, and a client developed using the React framework. Furthermore, these services have been containerized to ensure seamless deployment and

scalability. Kafka was chosen for its ability to decouple services and facilitate horizontal scaling, thereby enabling the addition of multiple clients without encountering scalability issues. A similar methodological approach was employed in the study cited as [11], where a more sophisticated pipeline was designed to address the challenge of maintaining social distancing during the COVID-19 pandemic. Their methodology incorporated the use of OpenCV in conjunction with a YOLOv3 model, which was deployed on a Kubernetes platform with the assistance of KubeFlow. For data processing, Apache Spark was utilized, and Kafka was employed to manage backend-to-backend communication. The processed data was subsequently routed to analytics using an API Gateway configured as a WebSocket, and the resultant analytical report was transmitted via email to the respective client.

In terms of utilizing Retrieval-Augmented Generation (RAG) techniques, a recent implementation showcased the use of retrieval-augmented models to generate responses based on multiple documents. However, these models tend to disregard the overarching topic and focus solely on the local context of the conversation [12]. In contrast, my pipeline employs a different strategy by using only a single document. This is achieved by deploying a Langchain agent that integrates information from a Large Language Model (LLM) and our PostgreSQL data store, ensuring a more focused and relevant response generation.

# 3. USED TECHNOLOGIES

In my coffee-making system, I leverage a diverse array of cutting-edge technologies within my software architecture. Now, I would like to delve into the specifics of those technologies that I make use of, and the rationale behind my approach.

- *Flutter* - framework that powers the frontend of my coffee-making system. By using Flutter, I streamline development by maintaining a single codebase for both Android and iOS, eliminating the need to build and manage separate codebases for each platform.
- *Dart* - represents the programming language I used for coding the client side Flutter application.
- *Stripe* - used to power online and in-person payment processing and financial solutions for businesses of all sizes [13]. In my coffee-making system, Stripe seamlessly handles payment processing whenever an order for coffee is placed.
- *AWS* - made use of their services in order to store the users of my application, and their personal information. I also make use of this, in order to handle other things, like the availability of all the drinks, all their information and their ratings from the users. The services that I use are: Lambda, API Gateway, S3 Bucket, DynamoDB and IAM. All the lambda functions are written in Javascript, Node16, while the API Gateways is a RESTful one. S3 is used for storage of the user's photo, and the noSQL DynamoDB database is used for storage of the things that I said earlier.
- *GCP* - used to show off your current location and the trajectory to our partnership companies.
- *Firebase* - used for achieving full duplex communication between the frontend and the backend when sending the coffee order (coffee_creation microservice)
- *Flask* - this framework was employed to create the majority of the microservices available. The following ones are: coffee_recommender, ingredients_updater, questionnaire_llm_updater, user_file_prompt_updater.
- *FastAPI* - this framework was employed to create the coffee_creation microservice.
- *Numpy, Pandas, Scikit-Learn* - used for data preprocessing and feature engineering on the coffee recommender model. Numpy is also used when performing training and inference on the YOLO models.
- *Pytorch* - framework employed to train the coffee recommender model.
- OpenCV - framework that is employed for the custom cup and drip area detection.
- *YOLO models* - used YOLOv8 and YOLOv9 for the custom cup detection task.
- *LangChain* - employed with the RAG method to update the coffee recipe given the responses from the questionnaire.

- *PostgreSQL* - database that contains all the users' responses for the questionnaire (each user can respond to the questionnaire multiple times).
- *SQLAlchemy* - ORM (Object Relational Mapper) used with Python for PostgreSQL to use classes and objects instead of plain SQL queries.
- *Mosquito (MQTT)* - used to decouple the frontend, from the questionnaire drink updater.
- *Kafka* - used to decouple the coffee_order microservice, with the one that handles the websocket communication, in order to facilitate horizontal scaling, enabling the addition of multiple clients without any issues.
- *React* - used to display the frames that are processed from the cup detection microservice.
- *Docker* - deployed almost all the microservices to Docker, except for the coffee_creation and cup_detection microservices. The decision to use Docker was primarily motivated by the need for consistency across different environments and the ability to scale efficiently. Additionally, Docker makes it straightforward to scale our services by adding more container instances to handle increased demand. This can be achieved using orchestration tools like Kubernetes or Docker Compose, which allow us to manage and scale our containerized applications seamlessly.

# 4. ARCHITECTURE AND IMPLEMENTATION

In this chapter, we will delve into the architecture of my coffee creation system, Cofster, exploring the interactions between the various microservices that power this system. By examining each component in detail, we aim to uncover the inner workings of the system, highlighting the methodologies employed to achieve this robust and scalable solution. This chapter will also shed light on the design principles and patterns that underpin the architecture, ensuring a thorough understanding of both the high-level structure and the finer details of Cofsters' microservice ecosystem.

## 4.1 SYSTEM ARCHITECTURE

As illustrated in Figure 4.1, at the core of my system, lies a Flutter mobile application. This application serves multiple functions, the most crucial being enabling users to purchase their favorite coffee and notifying them when their order is ready. An important aspect of Cofster is that multiple clients can place orders, which are added to the order queue using the FIFO method.

The integration of Cloud was also added, where the clients communicate with the serverless Lambda Functions via API Gateways that follow the REST architecture. A DynamoDB database stores all user-related tables and available drinks, while an S3 bucket is used to store user images upon account creation.

Additionally, the mobile application interacts with a Neural Network for two main purposes: displaying the top K favorite drinks and providing users with their favorite drink as a free gift upon account creation. This is achieved through a custom questionnaire built into the Flutter application. Users answer the questions, and the responses are passed as input to the model. Preprocessing of this data is done using Numpy, Pandas, and Scikit-Learn, including normalization with a standard scaler.

The preprocessed data is then fed into the Neural Network, which loads trained parameters from a pickle file to output either the top K favorite drinks or the user's favorite drink. This model is exposed via an API using the Flask framework, allowing it to be called from the frontend. To ensure consistency across different environments, this microservice is containerized.

Regarding the dataset, data is collected from two primary sources: real responses from individuals via a Google Questionnaire and randomly generated responses from GPT-3.5. These responses are merged to create a richer dataset, which is then fed into the training pipeline to facilitate the training process.

For our payment service, we've opted for Stripe due to its seamless integration with our application. Once payment is completed, order data is transmitted from the client via Firebase to our cup detection microservice. After paying, a timer will start, and when the time is elapsed, it will trigger a rerender on the UI side, in order to display a popup, allowing you to rate the actual drink that you consumed from.

Our cup detection system utilizes two cutting-edge models: YOLOv8 and YOLOv9. These models are dynamically selected using a Factory Design Pattern, ensuring flexibility and adaptability.

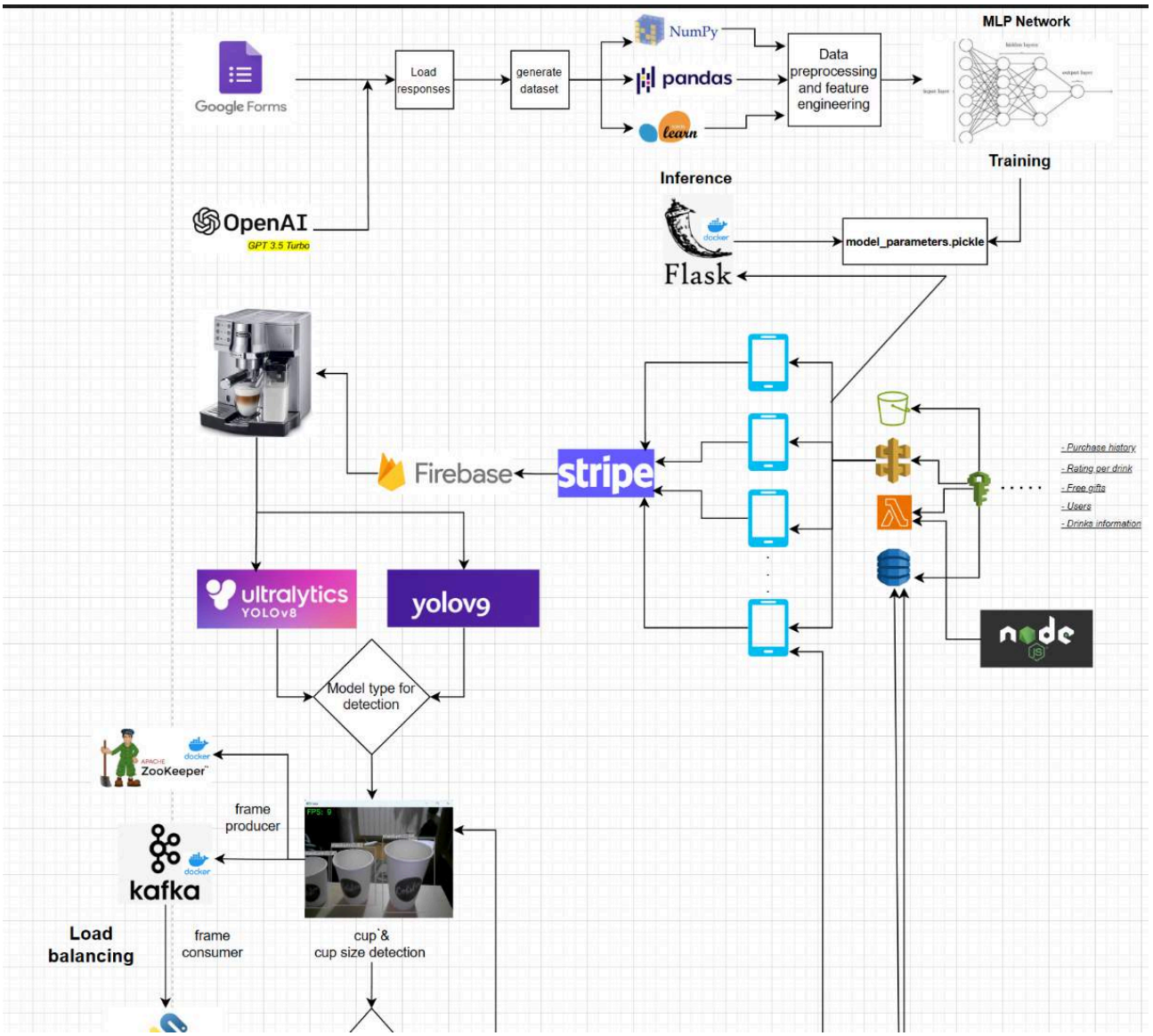


Figure 4.1 System Architecture part 1

As portrayed in Figure 4.2, Continuously monitoring the custom cup and the drip area, the cup detection microservice triggers the coffee creation microservice if both are detected for more than 10 seconds. It also redirects the client to a page where they can reply to a questionnaire in order to enhance their drink.

The coffee creation microservice, built using FastAPI and running on a Raspberry Pi 4, swiftly begins the process of creating the coffee. Once completed, the coffee creation microservice notifies the cup detection microservice, which, in turn, sends feedback to the user's mobile app via Firebase. This informs the user that their ordered coffee is now ready for collection.

For our drink enhancement pipeline, I've structured it around three microservices, all leveraging Flask and Docker for streamlined development and deployment. The update starts with the clients being subscribed to the MQTT Mosquitto topic, by answering the questions on the drink enhancement questionnaire, forwarding the information to the microservice that parses the data in a format that can be saved as an entry to a database entity, running on PostgreSQL. From the same entity, the latest k responses, from that given user, to that respective coffee, will be filtered, in order to update the coffee recipe. To update it, I make use of a Langchain agent, with the RAG method, using a LLM in the form of GPT (any model from GPT 3.5 to 4 family), as well as the PostgreSQL database that represents the chat history.

We also illustrate the flow of frames from a Kafka Producer to a Kafka Consumer. The Kafka Consumer facilitates bidirectional communication between Websockets and clients, enabling real-time updates. Despite currently having only one React application client, we've implemented this architecture with scalability in mind, making it easy to add additional clients in the future. All those services mentioned are containerized.

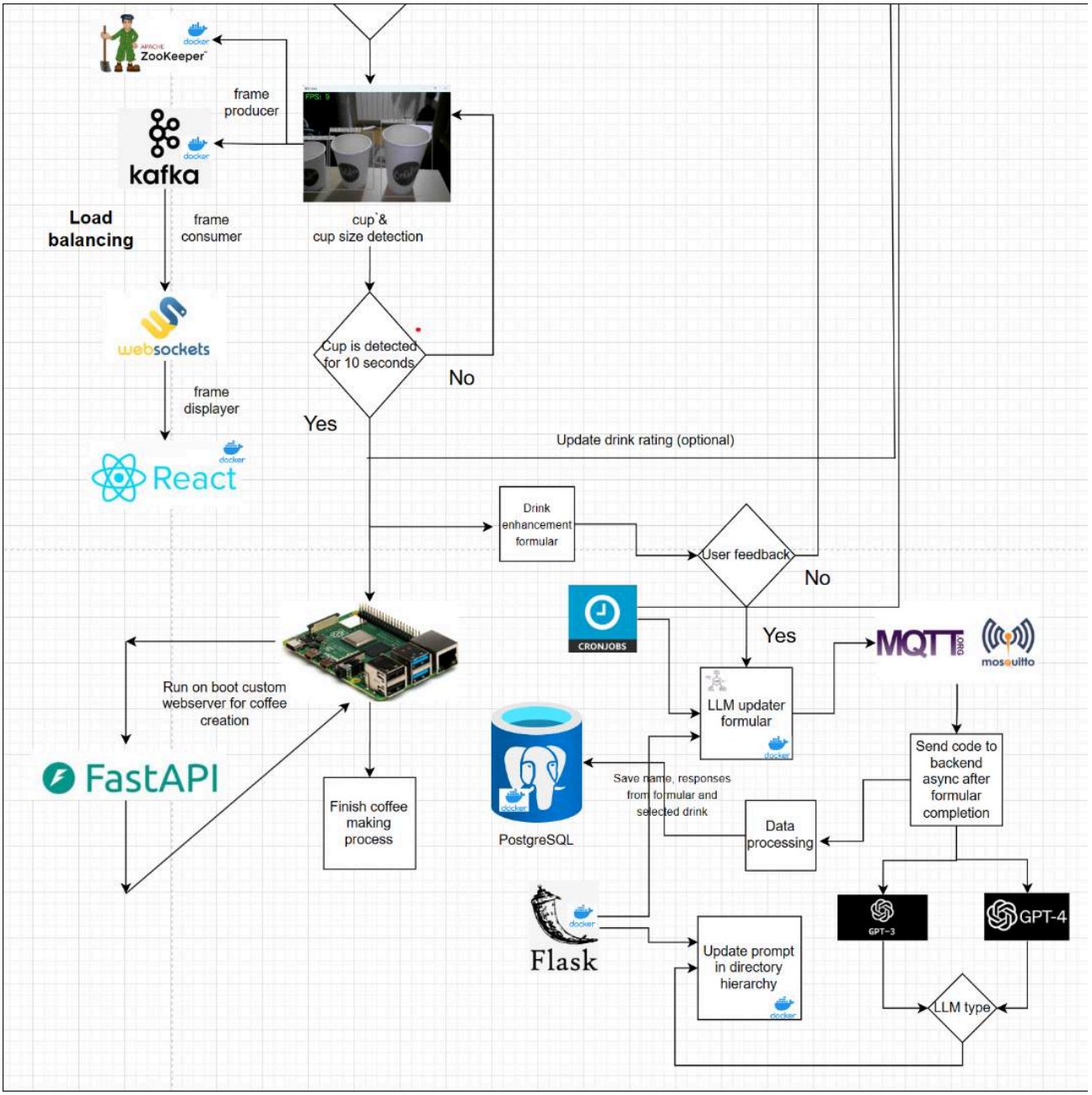


Figure 4.2 System Architecture part 2

## 4.2 HARDWARE ARCHITECTURE

Figure 4.3 illustrates the hardware architecture of my prototype coffee machine.

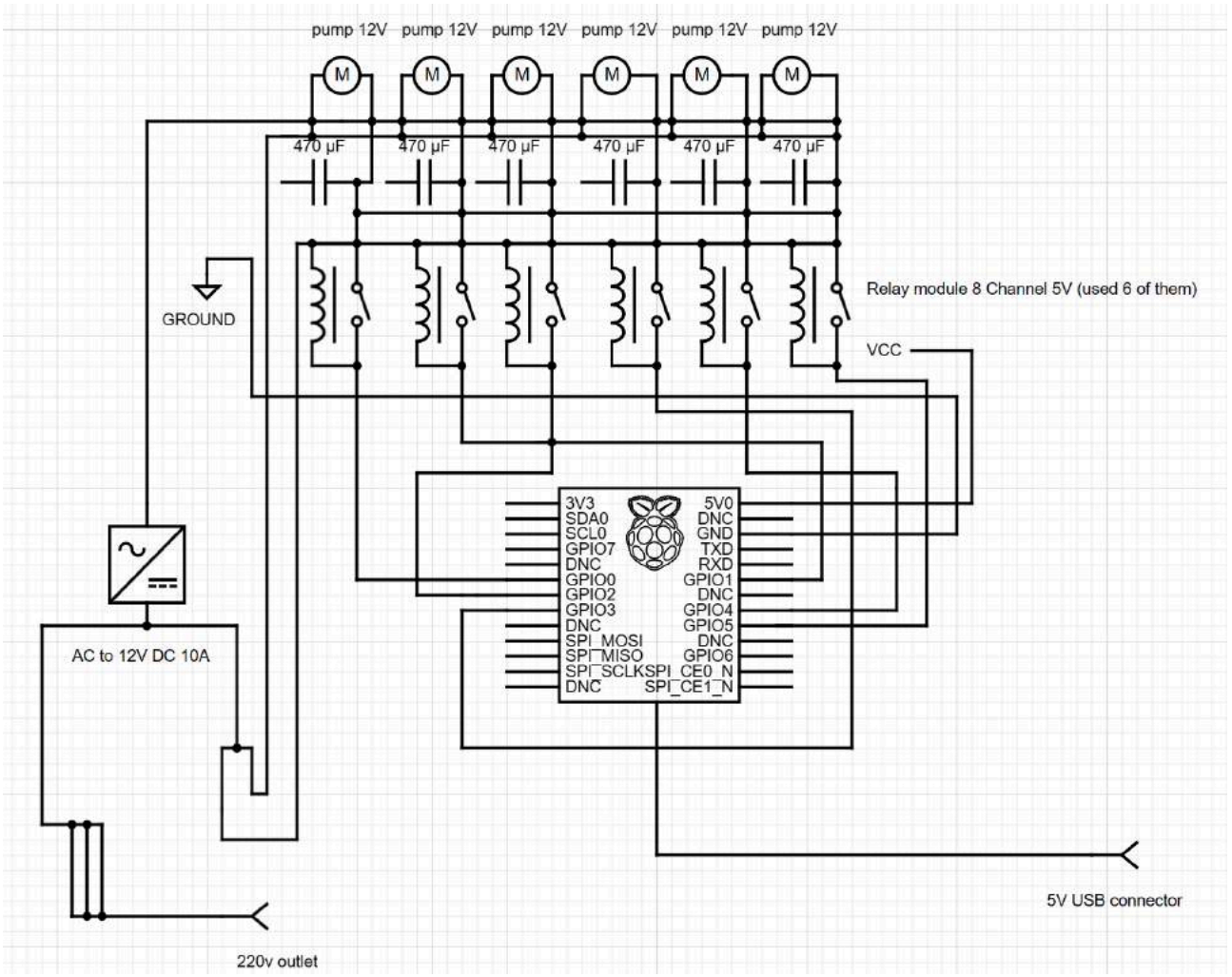


Figure 4.3 Hardware Architecture

The prototype coffee machine features a straightforward architecture, comprising a Raspberry Pi Model 4 with 8GB of RAM, a Redragon GW800 1080P Camera, an 8-channel 5V relay module (using only 6 inputs), six 470 µF capacitors, six 12V DC pumps, a small breadboard with LEDs, and an AC to 12V DC 10A converter.

The Raspberry Pi Model 4 orchestrates the coffee-making process. Each GPIO pin of the Raspberry Pi is connected to an input on the relay module, with only six inputs utilized for the six pumps, leaving the last two relay inputs unused. The converter is powered from a standard socket, with its ground pin connected to the left leg of each pump, wiring the pumps in parallel. The VCC from the converter connects to the relay's open port, and each switch is also connected in parallel. The common input from the relay module connects to the other leg of each pump.

Additionally, LEDs on the breadboard provide visual indicators for the start and finish of the coffee-making process.

### 4.2.1 RASPBERRY PI MODEL 4

As previously indicated, the component that has the most importance to our system is the Raspberry Pi. Without it, we could not create the actual coffee. Having in mind this, I want to say some words regarding the Raspberry Pi, and the actual reasoning of why we choose to use the Model 4 with 8 RAM.

The Raspberry Pi is a microprocessor created in England by the Raspberry Pi Foundation, which was formed in 2009. The purpose behind the development of this microprocessor was to promote the learning of simple electronics/computer science concepts in schools/universities. Its aim is also to provide a low-cost alternative to purchasing a traditional computer [14].

The performance of this microcontroller is significantly lower than that of a traditional PC, yet it can still provide all the features and capabilities. It is commonly used in applications such as image and video processing, as well as in Internet of Things (IoT) projects.

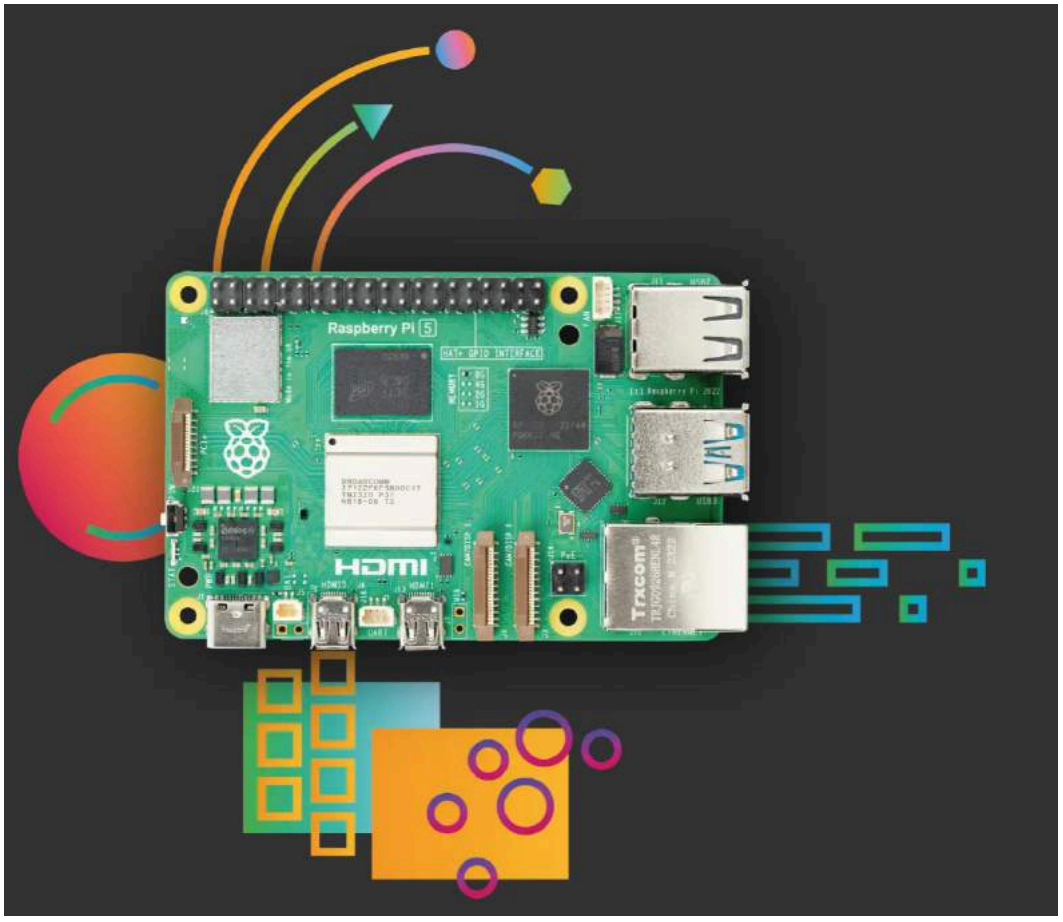


Figure 4.4 Raspberry Pi Model 5 [15]

Considering that for this project I used the model Raspberry Pi Model 4, I would like to make a comparison between this model, the newer Raspberry Pi 5 model.

| | **Raspberry Pi 4** | **Raspberry Pi 5** |
|---|---|---|
| **CPU** | Broadcom BCM2711<br>Quad-core<br>64-bit @ 1.8 GHz | Broadcom BCM2712<br>Quad-core<br>64-bit @ 2.4GHz |
| **GPU** | VideoCore VI @ 500 MHz<br>OpenGL ES 3.1, Vulkan 1.0 | VideoCore VII @ 800 MHz<br>OpenGL ES 3.1, Vulkan 1.2 |
| **Display output** | 2x Micro-HDMI<br>4K @ 60Hz (single display) | 2x Micro-HDMI<br>4K @ 60Hz (dual displays) |
| **Memory** | LPDDR4-3200 SDRAM<br>1GB, 2GB, 4GB or 8GB | LPDDR4X-4267 SDRAM<br>4GB or 8GB |
| **Network** | Wi-Fi, Bluetooth, Gigabit Ethernet<br>PoE enabled (with HAT) | Wi-Fi, Bluetooth, Gigabit Ethernet<br>PoE enabled (with HAT) |
| **USB ports** | 2x USB 2.0<br>2x USB 3.0 | 2x USB 2.0<br>2x USB 3.0 @ 5Gbps |
| **Power requirements** | 5V/3A via USB-C (15W) | 5V/5A via USB-C (27W) |
| **Storage** | Micro-SD card slot | Micro-SD card slot<br>M.2 SSD support (with HAT) |
| **Price** | $35 – $75<br>(depending on RAM amount) | $60 – $80<br>(depending on RAM amount) |

4.1 Raspberry Pi 5 vs Raspberry Pi 4 [16]

As shown in Table 4.1, the Raspberry Pi 5 outperforms the Raspberry Pi 4 in every aspect, but it comes at a higher cost. Therefore, we chose the Raspberry Pi 4 for our prototype, in order to save on money. For the coffee creation task, where cup detection is handled by a separate, isolated server, high computing power is unnecessary. The primary function is simply to create coffee; it does not require advanced processing capabilities, such as running the inference of the YOLO models for cup detection on the Pi.

### 4.2.2 FINAL HARDWARE IMPLEMENTATION

Figure 4.5 Coffee Machine Hardware Implementation

## 4.3 SOFTWARE ARCHITECTURE

In this chapter, I will provide a detailed overview of my software architecture, illustrating how the various components interact and communicate with each other.

In Figure 4.6, we see that Cofster has a Hybrid Cloud Architecture, we're I have servers both running on the cloud, as well as on premise. It has multiple microservices, in order to prioritize decoupling as much as possible, and to make the system more scalable.

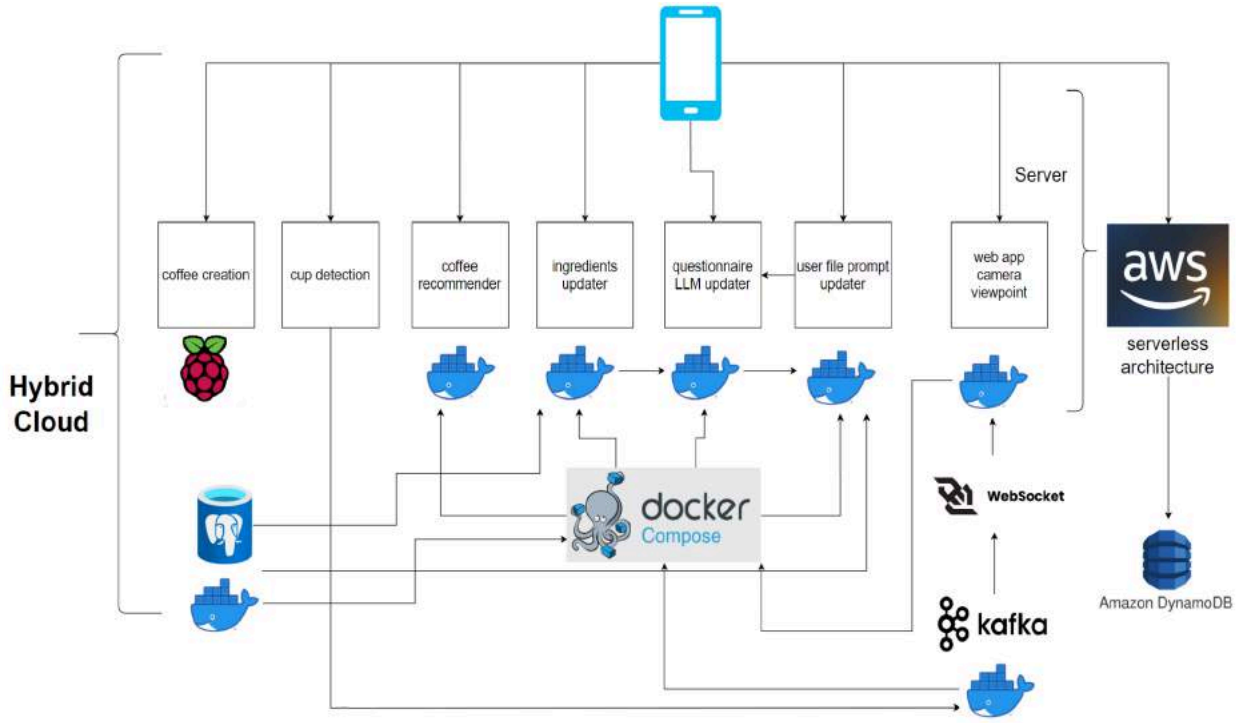


Figure 4.6 Software Architecture, Hybrid Cloud

Most microservices, excluding the coffee creation and cup detection microservices, are deployed as Docker containers to ensure portability, consistency, and scalability across various environments [17]. The coffee creation microservice is not deployed in a Docker container because it runs on a Raspberry Pi, where Docker would add unnecessary overhead. Similarly, the cup detection microservice is not containerized because it uses a GPU for YOLO model inference, achieving higher FPS in this manner. Linking a GPU to Docker would be overkill, so I decided to run it as a standalone process.

All the Docker containers are managed with the help of Docker Compose. With just a single command, "docker compose up," executed from the project's root directory, we start all the containers. This command streamlines the execution of all my containers, ensuring they are up and running. Further details regarding how to run all the services will be elaborated in the upcoming chapter.

### 4.3.1 SETTING UP THE ENVIRONMENT

In order to be up and running with the environment, you'll need to do the following things:

- Clone the repository using the command:

```
git clone https://github.com/glandaDarie/Cofster.git
```

- Run the frontend with the setup being specified in the *Frontend Setup* chapter.
- Run the backend with the setup being specified in the *Backend Setup* chapter.
- Purchase and connect the hardware components as specified in the *Hardware Architecture* chapter. While you have the option to re-engineer this setup, an edge device with internet connectivity is the only actual required hardware component that you need.

#### 4.3.1.1 FRONTEND SETUP

To run the client side, it's really simple, you'll just need to navigate (cd) to the root directory and then to the mobile_app_frontend directory, executing the following command:

```
flutter run
```

#### 4.3.1.2 BACKEND SETUP

To start running the backend, you'll need to do the following:

- Install Docker.
- Navigate to the microservices directory, you'll just need to cd into root/microservices.

- Build the Docker images for each microservice except the cup_detection one (if you want to do CUDA inferencing on the YOLO models, run it as a standalone process, and don't uncomment the code inside the Docker Compose) and the coffee_creation one, using the following command (you can also pull all of those images from Docker Hub and skip this step):

```
docker build -t "image-name" .
```

- Pull the rest of the images, from the Docker Hub website, using he command:

```
docker pull postgres && docker pull bitnami/zookeeper && docker pull bitnami/kafka && docker pull bitnami/spark && docker pull busybox
```

- After building the Docker images, return to the root directory.
- Start all microservices using docker compose, with the following command:

```
docker compose up
```

- For the coffee_creation microservice to work, you should have an edge device, preferably a Raspberry Pi (should work for any version of RPI).

  To run the server on boot, follow these steps:

  a) Navigate to the microservices/coffee_creation directory on your edge device.
  b) Execute the following shell script:

```
bash boostrap_loader_service.sh
```

Ensure you have a service created to run the process on boot. To create the service, follow these steps:

a) Navigate to the system service directory:

```
cd /etc/systemd/system/
```

b) Create the coffee making service unit file. An example file named bootstrap-coffee-making-service.service is provided in the microservices/coffee_creation directory.

- If you want to close the backend, power off the Raspberry Pi, as well as run the following command:

```
docker compose down
```

It's worth noting that with the APIs that I wrote on AWS, you can access resources seamlessly without needing to run any commands. Thanks to the serverless architecture of AWS Lambda, the servers are always up and running to handle requests.

#### 4.3.1.3 OPTIONAL: CUDA INFERENCE SETUP

If you are doing inference, you can also use your own GPU for the YOLO models. You should run the microservice as a standalone process, ignoring Docker. The following things are needed to accomplish GPU inference:

- Navigate to the microservices/cup_detection directory.
- Install the necessary Python dependencies using:

```
pip install -r requirements.txt
```

- Run the cup detection microservice:

```
python app.py
```

### 4.3.2 MOBILE APPLICATION

As mentioned in the previous chapters, the mobile application, built with the Flutter framework, has the primary scope to act as an intuitive interface for coffee ordering. For building it, I deployed a layered architecture, as you can see in Figure 4.7, similar to the Domain Driven Design one (DDD), but using a state manager as well, in the form of the Provider State Manager.

Regarding the utilization of this design approach, it was to decouple the components as much as possible. This design, more or less, is used on the backend side, with microservices [18], but because of the rise of Micro Frontends, which they appeared in 2016, this design approach is more and more adopted for the client side as well [19].

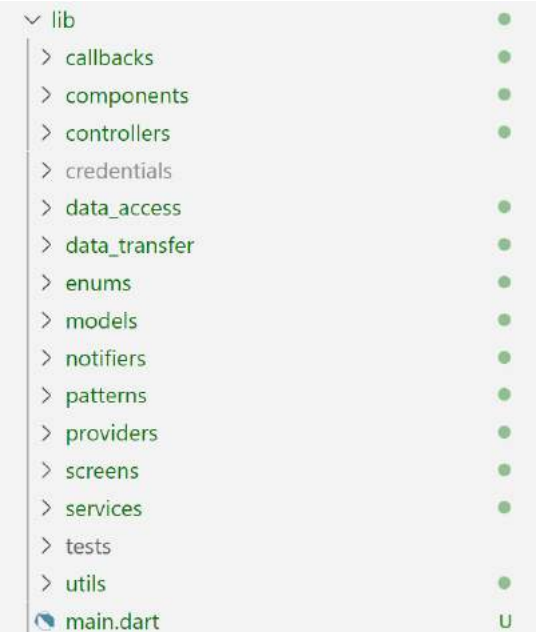


Figure 4.7 Frontend: Layered Architecture

The mobile application features several distinct screens, including the following:

- *Authentication Screen:* used for users to log in securely or create new accounts. Upon successful account creation, a confirmation code to the user's email is sent. Matching the confirmation code will give the user access to go to the next screen. To send the confirmation code, I have employed the mailer library, directly from Flutter

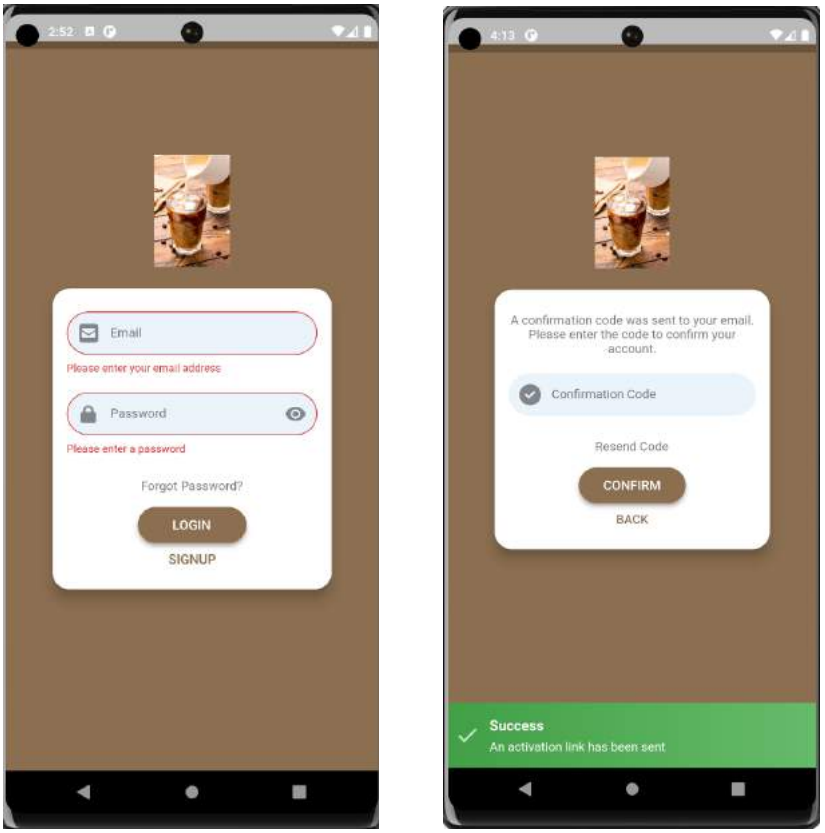


Figure 4.8 Authentication Screen

- *Photo Screen:* used for users to personalize their profile picture. If they choose not to, the Guest profile picture will appear on their account through the application

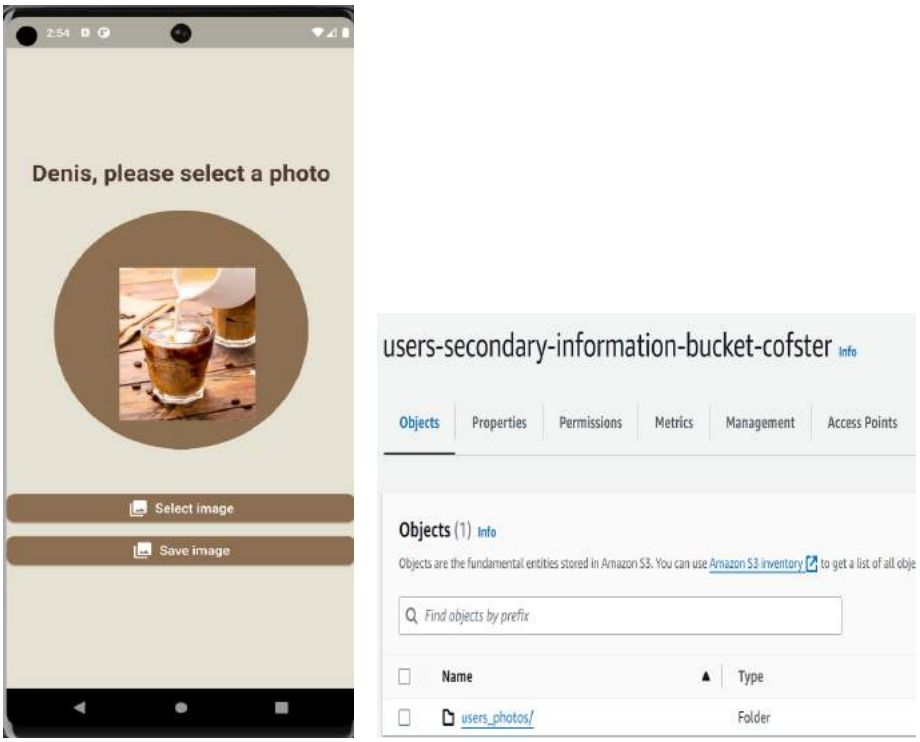


Figure 4.9 Authentication Screen

- *Coffee Questionnaire Screen:* used to fill out a form with 7 questions in order to receive a coffee gift upon on account creation, as well as a list of their favorite k drinks from a Neural Network

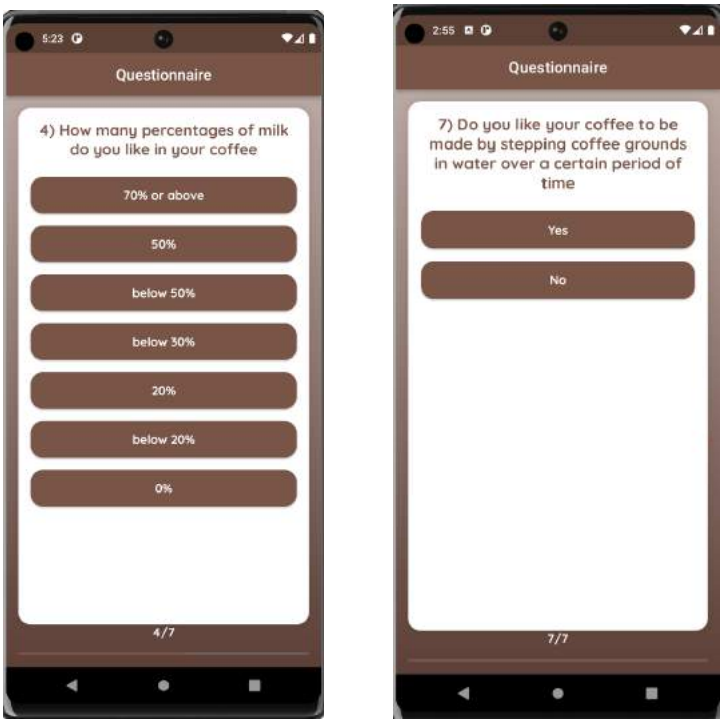


Figure 4.10 Questionnaire Screen

- *Main Screen:* used to display information like top k favorite drinks, additional user profile data, all liked drinks, shops that Cofster has partnership with, as well as all the gifts present. If you create a profile for the first time, you will get a free gift from the coffee recommendation model, which represents your favorite drink (information is gathered from the questionnaire). It also Includes a voice navigation feature for effortless access to preferred drinks. The navigation system makes use of the Levenshtein algorithm to calculate the word similarity between the ground truth one, and the predicted one, from the voice to text converter. More on this, in the next chapter.

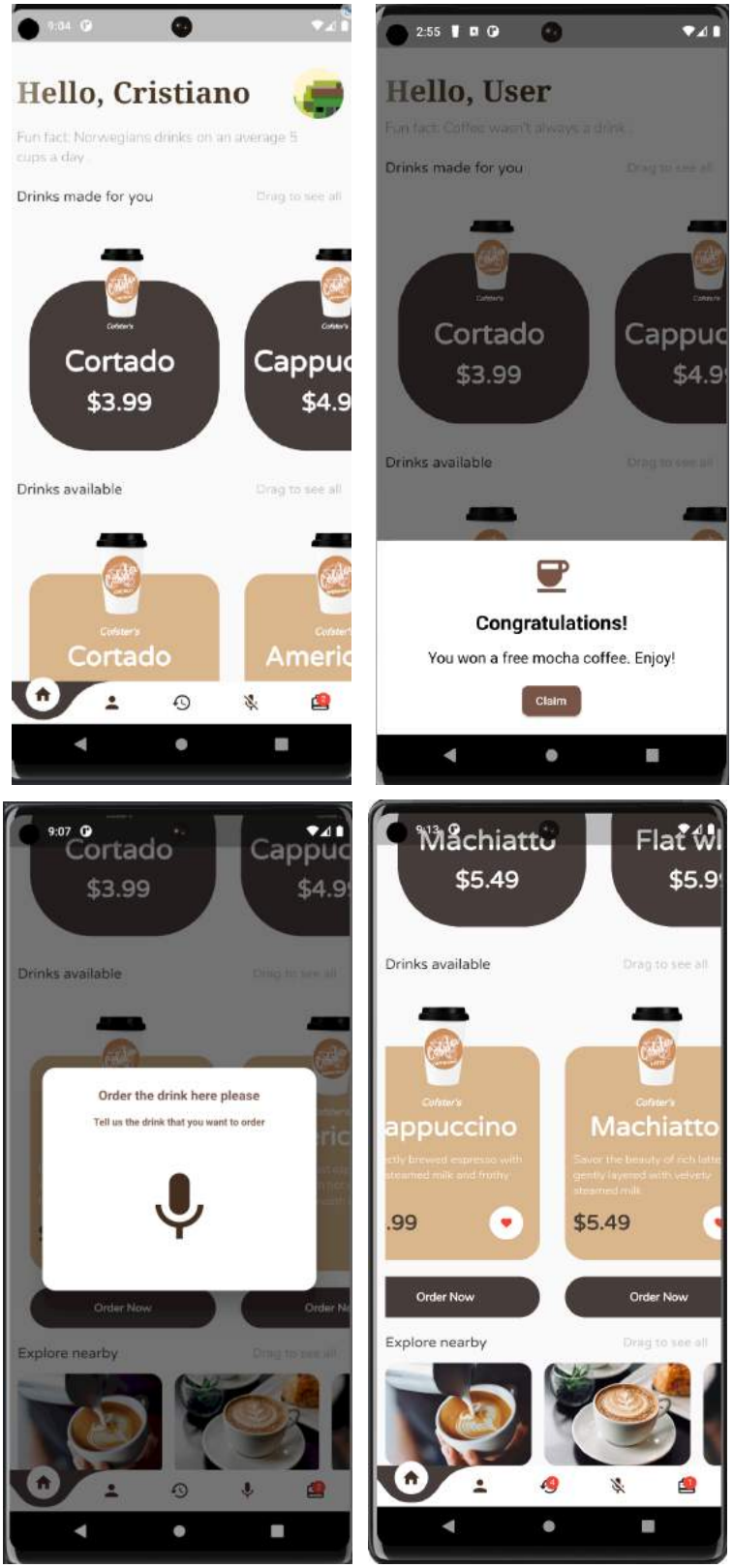


Figure 4.11 Main Screen

- *Details Screen:*
  Users customize beverages, with the option for autonomous drink creation based on preferences. On the UI, you can choose “Drink Made By”, either to be done using “Recipe ”or “LLM” (Large Language Model). There is also a feature where you can give a rating of how much you liked the drink, from 1 to 5, updating the coffee rating with your additional feedback, using an average mean formula for this. Payment is processed using the Stripe Payment Service and after that, the system will be in the cup detection phase.

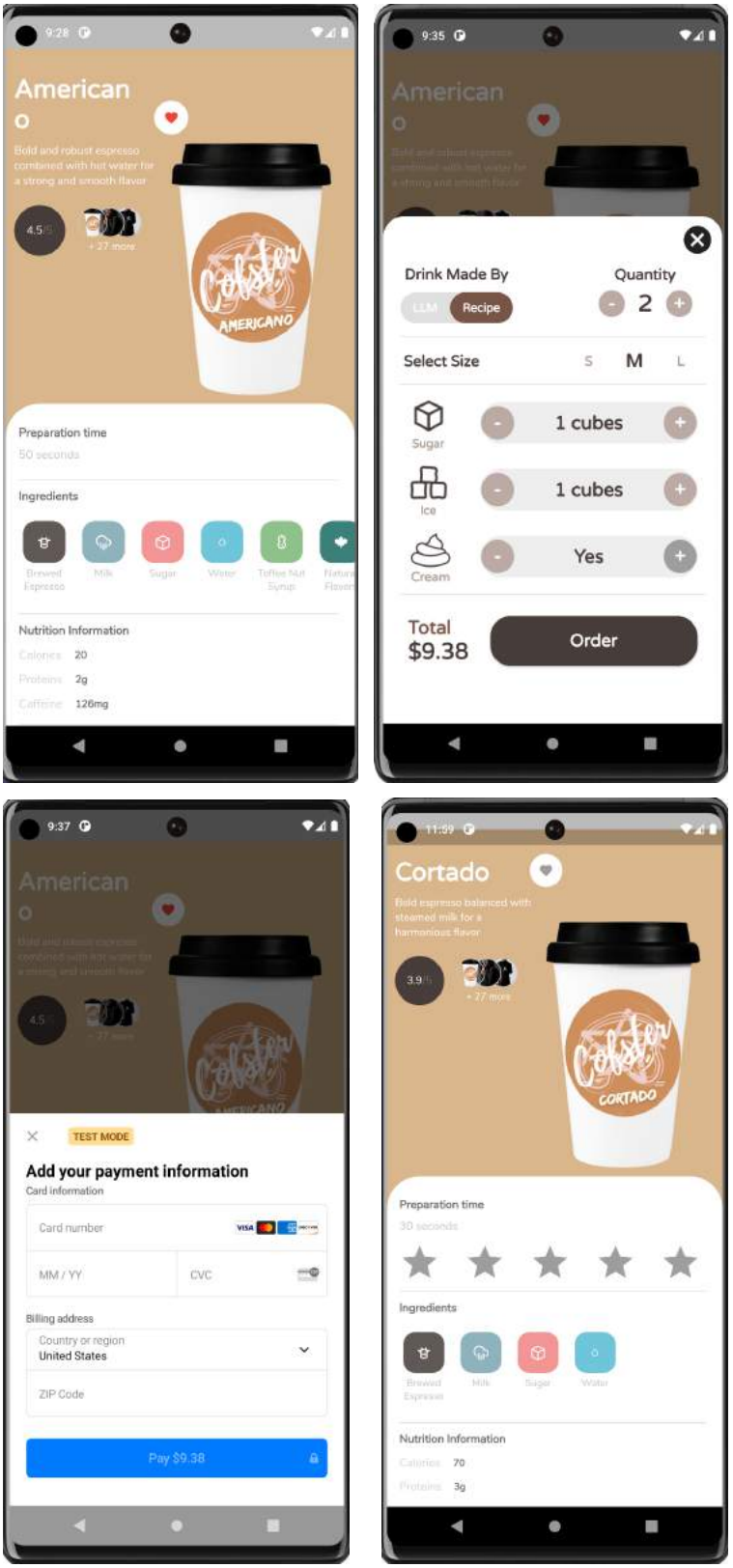


Figure 4.12 Details Screen

- *Free Gifts Screen:*
  Users redeem coffee gifts seamlessly, transitioning to the Stripe Payment Service for checkout and paying the tax. This feature on the backend is powered by AWS using Lambda, a RESTful API Gateway and DynamoDB.

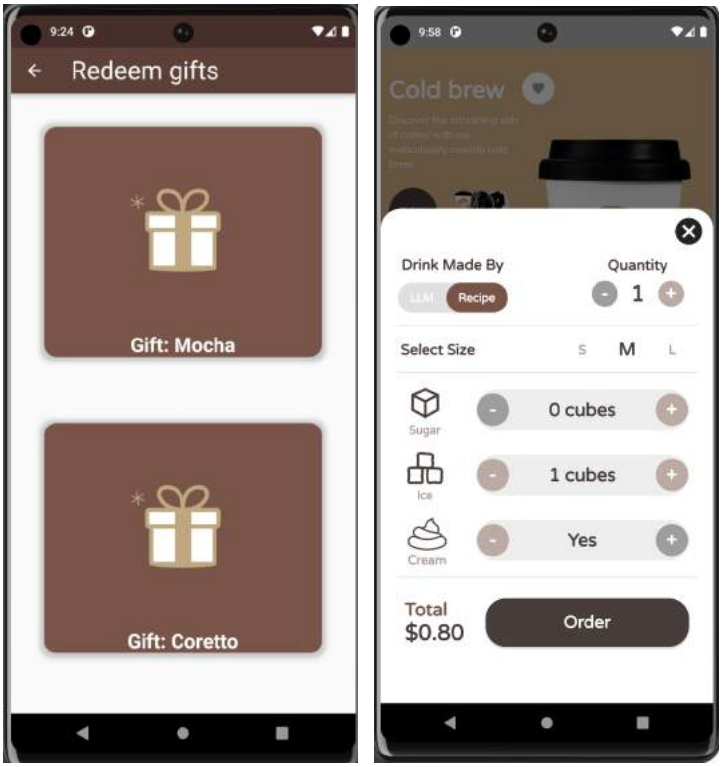


Figure 4.13 Free Gifts Screen

- *User Information Screen:*
  Used to act as an access point for all the screens left. At the top of the screen, it displays the photo chosen from the user at the start of the account creation, as well as their name.

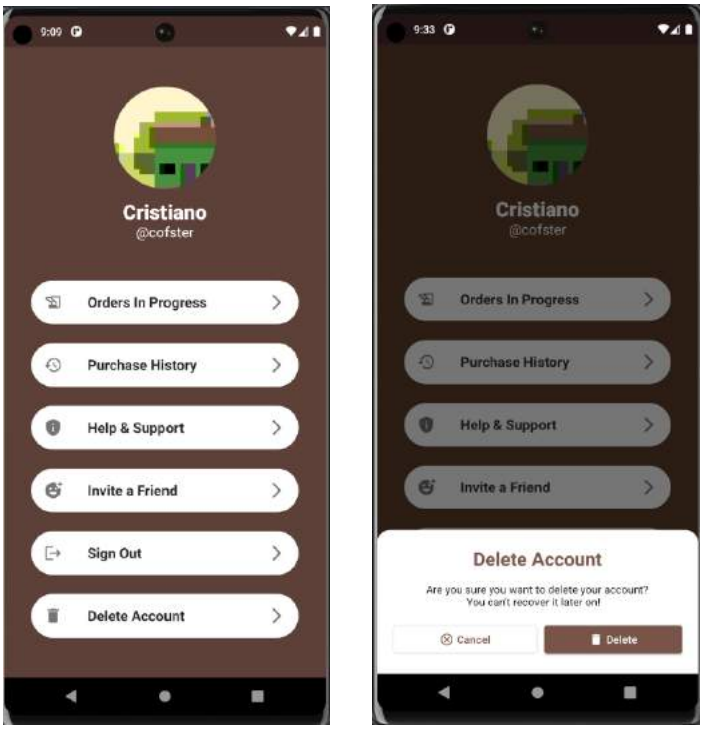


Figure 4.14 User Information Screen

- *Rest Of The Screens:*
  Represent the rest of the screens connected to the User Information Screen: Orders in Progress (view active orders in real-time with Firebase), Purchase History (stores all purchases for the account), Help & Support (assistance with FAQs, live chat, contact form), and additional features like signing out, deleting the account, and inviting a friend.

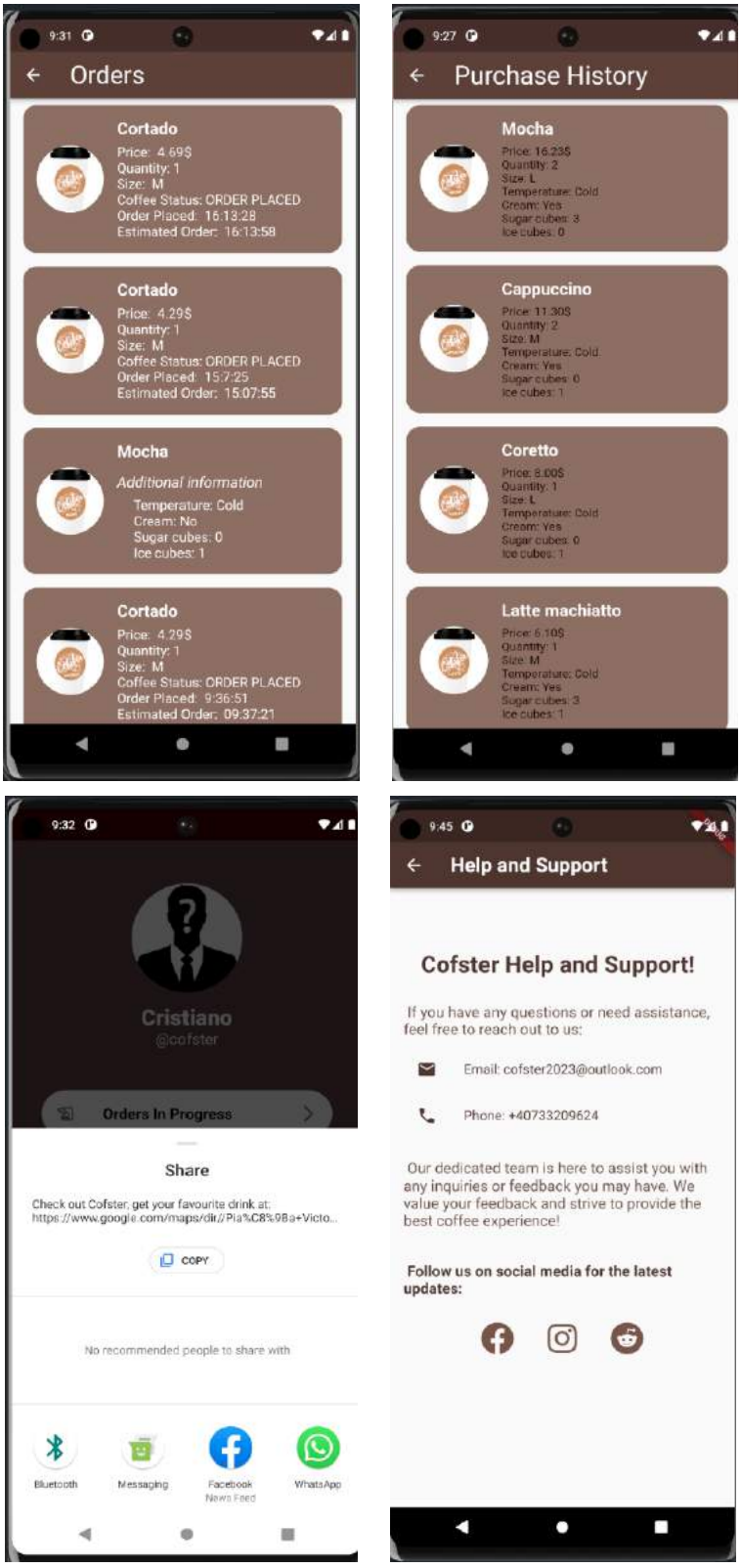


Figure 4.15 Rest Of The Screens

### 4.3.2 VOICE TO TEXT PIPELINE

This pipeline features a straightforward architecture that leverages a pre-trained model to convert voice to text. It employs the Levenshtein algorithm to compare the received word with the ground truth one. The Levenshtein algorithm calculates a similarity score between the two words. Padding of '0' is added either to the received word or the ground truth one, depending on which word is smaller, facilitating a direct comparison. The similarity score, ranging from 0 to 1, indicates how closely the two words resemble each other. Finally, the similarity score is compared against a threshold value to determine if the words are sufficiently similar. A simple example of the algorithm can be viewed here:

1. Convert the received word using the voice to text converter

received: cartados
actual: cortado

```
Future<void> startListening() async {
  try {
    await speechToText.listen(onResult: this._callbackSetSpeechResult);
  } catch (e) {
    ToastUtils.showToast(
        "Cannot start listening to users voice, error: ${e}");
    return;
  }
}
```

Figure 4.16 Convert Word Using Voice To Text Converter

2. Add padding to the word that is smaller, in my case, I add padding to the right

received: cartados
actual: cortado0

```
List<String> padStringsToMatchLength(
    String s1, String s2, String paddingCharacter) {
  int maxLength = s1.length > s2.length ? s1.length : s2.length;
  s1 = s1.padRight(maxLength, paddingCharacter);
  s2 = s2.padRight(maxLength, paddingCharacter);
  return [s1, s2];
}
```

Figure 4.17 Add Padding To The Smaller Word

3. Calculate the similarity of those two words, using a simple average mean formula:

received: cartados
actual: cortado0

cartados -> word length = 8
cortado0 -> word length = 8

_____
10111110 -> 6/8 = 0.75
similarity_score = 0.75

```
double scoreProbability(String s1, String s2) {
  List<String> strings = padStringsToMatchLength(s1, s2
  s1 = strings[0];
  s2 = strings[1];
  int matches = List.generate(
    s1.length,
    (i) => i,
    growable: false,
  ).where((i) => s1[i] == s2[i]).length;
  return matches / s1.length;
}
```

Figure 4.18 Calculate The Similarity Score

4. Compare the similarity score to a threshold, returning if either the given drink is found or not.

```
CoffeeCard getParticularCoffeeCardGivenTheNameOfTheCoffeeFromSpeech(
    String coffeeNameSpeech,
    [double threshold = 0.6]) {
  for (CoffeeCard objectCoffeeCard in this._objectsCoffeeCards) {
    double similarityScore =
        scoreProbability(coffeeNameSpeech, objectCoffeeCard.coffeeName);
    if (similarityScore >= threshold) {
      return objectCoffeeCard;
    }
  }
  return null;
}
```

Figure 4.19 Compare Similarity Score For Each Word To A Threshold

As we can see, this implementation uses this simple Levenshtein with padding algorithm to calculate the similarity score. I could change the approach, using a more state-of-the-art one, by calculating word similarity using vector databases.

Vector databases are used for fast and accurate similarity search and retrieval. They make use of word embeddings, which are vectors that represent words as numbers [20]. They can be converted to vectors using pre-trained models such as Word2Vec or BERT. The similarity score between words can be calculated using cosine similarity or dot product similarity, like in the paper [21], for the Transformer layer.

Using vector databases for this task, instead of the Levenshtein algorithm, adds semantic understanding, contextual similarity, scalability, and efficiency, which would drastically improve the initial algorithm.

### 4.3.3 COFFEE RECOMMENDER MICROSERVICE

Having in mind that I want to reward people that use my system, I've adopted a coffee recommendation pipeline that has a dual purpose: giving the user a free gift upon account creation, as well as displaying a list of his k favorite drinks. For my setup, I've chosen k to be 5, precisely half of the total number of available drinks in my inventory, which currently stands at 10.

The coffee recommendation system is structured around three fundamental components: data collection, training, and inference. Each of these elements plays a crucial role in the system's operation, contributing to its overall efficacy and user satisfaction. Now, let's delve deeper into the workings of each component:

- ☐ *Data collection:* In Figure 4.20, data collection involves real user responses collected via Google Forms. Users answer questions about their coffee preferences, including selecting their favorite coffee from my inventory. This question will act as the label, while the other 7 questions serve as features for my coffee recommendation.

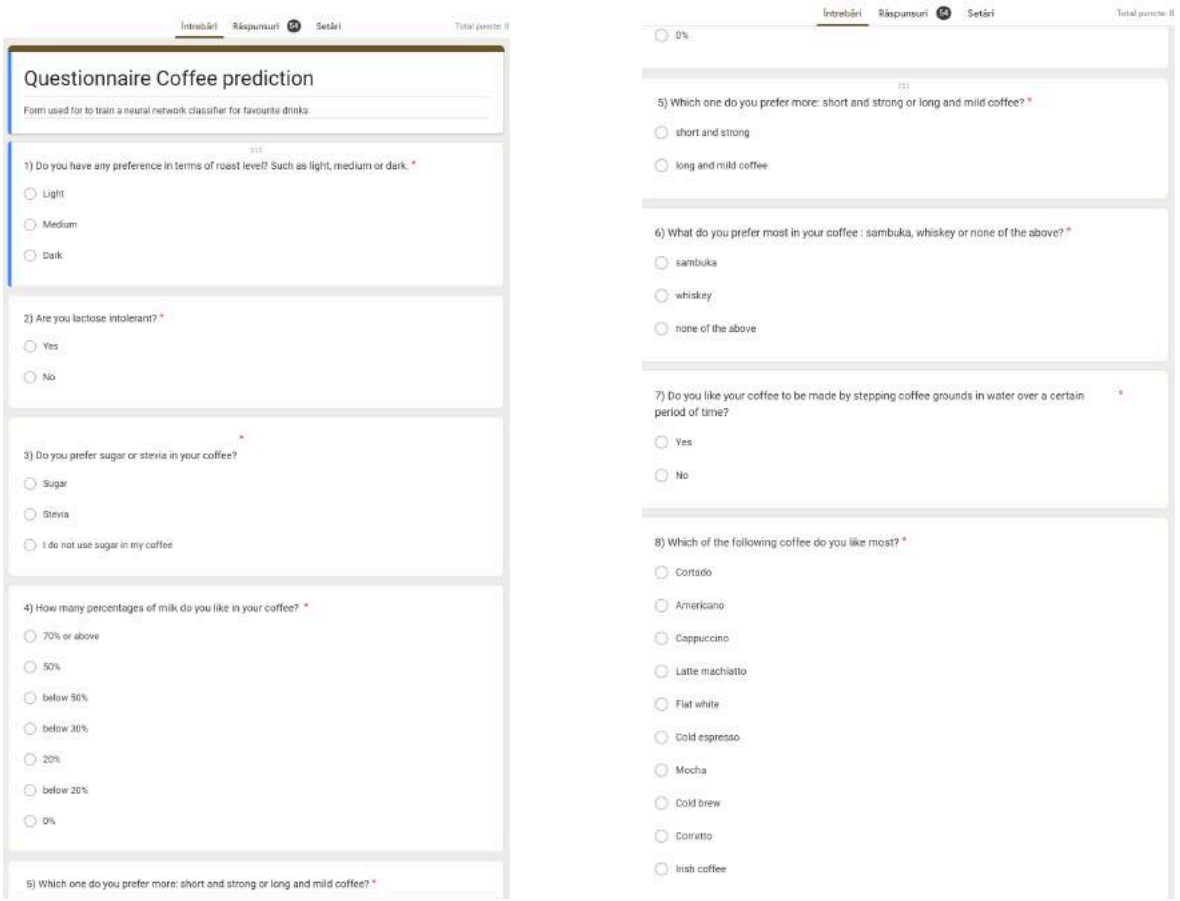


Figure 4.20 Data Collection From Google Forms

Sadly, we couldn't gather that many responses from real people, only 54, so I mimicked "real responses" using GPT 4, employing a random sampling technique in order to reduce the number of API calls to GPT, enhancing cost and speed along the way. The data collection part was done all programmatically, using a python script, where a part of the code can be seen, on Figure 4.21.

```python
def format_samples_and_features(questions : List[List[str]], answers_samples : List[List[str]]) -> Tuple[str, str]:
    columns, rows = "", ""

    for (i, q_and_o) in enumerate(questions):
        for (j, q) in enumerate(q_and_o):
            q = re.match(pattern=r"^.*?(?=\?|:)|^[^:]+$", string=q).group(0).strip()
            if i == (len(questions) - 1): columns += f"{q[3:].lower()}"
            else: columns += f"{q[3:].lower()},"
            break

    for answs in answers_samples:
        for (j, answ) in enumerate(answs):
            if j == len(answs) - 1: rows += f"{answ.lower()}"
            else: rows += f"{answ.lower()},"
        rows += f"\n"

    return (columns, rows)

def merge_samples_from_real_people_with_the_ones_generated_by_gpt4(path : str, features_people : str, samples_people : str) -> None:
    samples_gpt4 = read_and_trim_samples_generated_by_gpt4(os.path.join(DIRNAME, "samples_generated_with_gpt4.txt"))
    features_people, samples_people = format_samples_and_features(features_people, samples_people)
    create_dataset(os.path.join(DIRNAME, "coffee_dataset.txt"), features_people, samples_gpt4, samples_people)

def create_dataset(path : str, features_people : str, samples_gpt4 : str, samples_people : str) -> None:
    with open(path, "w") as file:
        file.write(f"{features_people}\n{samples_gpt4}\n{samples_people}")

if __name__ == "__main__":
    store = file.Storage('token.json')
    creds = None
    if not creds or creds.invalid:
        flow : client.OAuth2WebServerFlow = client.flow_from_clientsecrets(CREDENTIALS, SCOPES)
        creds = tools.run_flow(flow, store)
        service = discovery.build('forms', 'v1', http=creds.authorize(
        Http()), discoveryServiceUrl=DISCOVERY_DOC, static_discovery=False)
    form_id : str = "1tEauj7n4n86CiCB-49YOfKbe1I62QWOlluDzcjytxng"
    questions = service.forms().get(formId=form_id).execute()
    answers = service.forms().responses().list(formId=form_id).execute()
    work_json(questions, answers)
```

Figure 4.21 Integrating and Merging Real And AI-Generated Responses

The final dataset consists of 30.399 samples, with 8 features. Figure 4.22 offers a visual illustration of some samples from this dataset, showcasing its structure and characteristics.

```
dark,yes,i do not use sugar in my coffee,0%,short and strong,none of the above,no,cold espresso
dark,yes,i do not use sugar in my coffee,70% or above,short and strong,none of the above,no,latte machiatto
medium,yes,stevia,0%,short and strong,none of the above,no,cold espresso
dark,yes,stevia,0%,short and strong,none of the above,no,cortado
medium,no,sugar,below 30%,short and strong,none of the above,no,flat white
light,no,sugar,below 30%,short and strong,none of the above,no,flat white
medium,no,sugar,70% or above,long and mild coffee,none of the above,yes,latte machiatto
dark,no,i do not use sugar in my coffee,20%,short and strong,none of the above,no,flat white
light,no,stevia,0%,long and mild coffee,none of the above,no,americano
medium,no,sugar,70% or above,short and strong,none of the above,no,mocha
dark,yes,i do not use sugar in my coffee,0%,short and strong,none of the above,no,cold espresso
medium,yes,stevia,0%,short and strong,none of the above,no,coretto
dark,no,i do not use sugar in my coffee,70% or above,long and mild,none of the above,no,latte machiatto
light,no,stevia,0%,long and mild,none of the above,no,americano
dark,no,stevia,0%,short and strong,none of the above,yes,cold brew
medium,no,stevia,50%,long and mild coffee,none of the above,no,cappuccino
light,no,sugar,50%,short and strong,none of the above,no,mocha
dark,no,stevia,50%,short and strong,none of the above,yes,cold brew
dark,yes,stevia,0%,short and strong,none of the above,no,cold espresso
```

Figure 4.22 Some Samples From The Final Dataset

- ☐ *Model training:* At the first stage, the data was preprocessed, converting all the question answers from the dataset, to numerical. Figure 4.23 showcases this process.

```
def convert_questions_to_numerical(self) -> None:
    self.data["Question 0"] : pd.DataFrame = pd.DataFrame([0 if row_answer_q0 == "light" else (1 if row_answer_q0 == "medium" else 2) \
                            for row_answer_q0 in self.data["Question 0"]]).astype("float64")
    self.data["Question 1"] : pd.DataFrame = pd.DataFrame([0 if row_answer_q1 == "no" else 1 \
                                            for row_answer_q1 in self.data["Question 1"]]).astype("float64")
    self.data["Question 2"] : pd.DataFrame = pd.DataFrame([0 if row_answer_q2 == "sugar" else (1 if row_answer_q2 == "stevia" else 2) \
                            for row_answer_q2 in self.data["Question 2"]]).astype("float64")
    question_3_options = {row_answer_q3:index for (index, row_answer_q3) in enumerate(list(dict.fromkeys(self.data["Question 3"])))}
    self.data["Question 3"] : pd.DataFrame = pd.DataFrame([question_3_options[row_answer_q3] \
                                            for row_answer_q3 in self.data["Question 3"]]).astype("float64")
    self.data["Question 4"] : pd.DataFrame = pd.DataFrame([0 if row_answer_q4 == "short and strong" else 1 \
                                            for row_answer_q4 in self.data["Question 4"]]).astype("float64")
    self.data["Question 5"] : pd.DataFrame = pd.DataFrame([0 if row_answer_q5 == "sambuka" else (1 if row_answer_q5 == "whiskey" else 2) \
                                            for row_answer_q5 in self.data["Question 5"]]).astype("float64")
    self.data["Question 6"] : pd.DataFrame = pd.DataFrame([0 if row_answer_q6 == "yes" else 1 \
                                            for row_answer_q6 in self.data["Question 6"]]).astype("float64")
    question_7_option_labels : List[str] = {row_answer_q7:index for (index, row_answer_q7) in enumerate(list(dict.fromkeys(self.data["Question 7"])))}
    self.data["Question 7"] : pd.DataFrame = pd.DataFrame([question_7_option_labels["latte macchiato"] if row_answer_q7 == "latte machiatto" \
                                            else (question_7_option_labels["corretto"] if row_answer_q7 == "coretto" \
                                                 else question_7_option_labels[row_answer_q7]) \
                                                    for row_answer_q7 in self.data["Question 7"]])
    question_7_option_labels : List[str] = {row_answer_q7:index for (index, row_answer_q7) in enumerate(list(dict.fromkeys(self.data["Question 7"])))}
    self.data["Question 7"] : pd.DataFrame = pd.DataFrame([question_7_option_labels[row_answer_q7] for row_answer_q7 in self.data["Question 7"]])
```

Figure 4.23 Convert All The Question Answers To Numerical

Afterward, I standardize the data using a standard scaler. The formula for the standard scaler is shown in Figure 4.24, and the implementation is illustrated in Figure 4.25.

$$z = \frac{x - \mu}{\sigma}$$

$$\mu = \text{Mean}$$

$$\sigma = \text{Standard Deviation}$$

Figure 4.24 Standard Scaler Formula [22]

```python
def standard_scaler(column_data : pd.DataFrame, params : dict = None) -> List[float]:
    if params is None:
        mean : pd.DataFrame = column_data.mean()
        std : pd.DataFrame = column_data.std()
        append_standard_scaler_params(path=os.path.join(os.path.dirname(__file__), "cache_standard_scaler_params.txt"), \
                                      mean=mean, std=std)
    else:
        mean, std = params["mean"], params["std"]
    return (column_data - mean) / std

def standardize_data(self) -> pd.DataFrame:
    for (i, column) in enumerate(self.data.columns):
        if i >= len(list(self.data.columns)) - 1: # don't normalize the labels
            continue
        self.data[column] = pd.DataFrame(standard_scaler(self.data[column])).astype("float64")
    return self.data
```

Figure 4.25 Standard Scaler Implementation

Having the data scaled, with mean 0 and standard deviation 1, I'm also separating them into two NumPy arrays, holding both the features and the labels. After doing this, I make use of the Scikit-Learn library to split my data in four sets, the X_train, X_test, Y_train, Y_test, using the train_test_split function. The training set will be 80%, while the test set 20% of the actual data. Figure 4.26, showcases the shape of the NumPy arrays, after calling the train_test_split function on my data.

```
X_train shape: torch.Size([24318, 7]), Y_train shape: torch.Size([24318])
X_test shape: torch.Size([6080, 7]), Y_test shape: torch.Size([6080])
```

Figure 4.26 Shape Of Arrays After Splitting

After finishing the data preprocessing, the training can start. The model that Ibuilt is a simple Multiple Layer Perceptron, using Pytorch as the framework. For the model, the weights and biases are initialized using Xavier initialization. The architecture includes an input layer, followed by a batch normalization layer, a ReLU activation function, and a dropout layer with a probability between 0.4 and 0.6. These components are repeated between all the layers, where there are 6 layers in total. At the end, we get raw logits back, with 10 output features, representing the actual classes of the drinks in the inventory. Figure 4.27 shows us what a component looks like, while Figure 4.28 shows us the implementation.

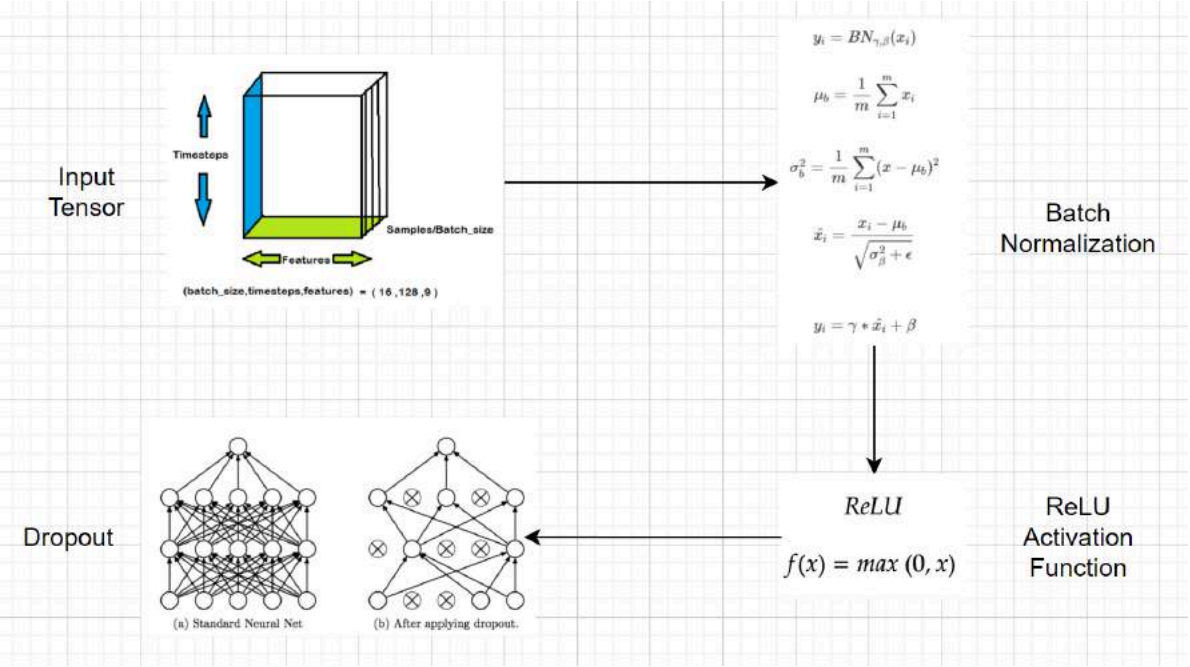


Figure 4.27 A Component With The Architectural Layers [23], [24], [25], [26]

```python
def forward(self, X : torch.float32) -> torch.Tensor:
    z1 = self.layer_input_hidden(X)
    z1 = self.bn1(z1)
    a1 = self.relu(z1)
    a1 = self.dropout1(a1)

    z2 = self.layer_hidden_hidden(a1)
    z2 = self.bn2(z2)
    a2 = self.relu(z2)
    a2 = self.dropout2(a2)

    z3 = self.layer_hidden_hidden_second(a2)
    z3 = self.bn3(z3)
    a3 = self.relu(z3)
    a3 = self.dropout3(a3)

    z4 = self.layer_hidden_hidden_third(a3)
    z4 = self.bn4(z4)
    a4 = self.relu(z4)
    a4 = self.dropout4(a4)

    z5 = self.layer_hidden_hidden_fourth(a4)
    z5 = self.bn5(z5)
    a5 = self.relu(z5)
    a5 = self.dropout5(a5)

    z6 = self.layer_hidden_output(a5)
    a6 = self.relu(z6)
    return a6
```

Figure 4.28 Code Of The Model

You can also visualize on Figure 4.29 the layered architecture of the Neural Network.

```
DrinkClassifier(
  (layer_input_hidden): Linear(in_features=7, out_features=128, bias=True)
  (bn1): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout1): Dropout(p=0.4, inplace=False)
  (layer_hidden_hidden): Linear(in_features=128, out_features=256, bias=True)
  (bn2): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout2): Dropout(p=0.6, inplace=False)
  (layer_hidden_hidden_second): Linear(in_features=256, out_features=128, bias=True)
  (bn3): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout3): Dropout(p=0.5, inplace=False)
  (layer_hidden_hidden_third): Linear(in_features=128, out_features=256, bias=True)
  (bn4): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout4): Dropout(p=0.6, inplace=False)
  (layer_hidden_hidden_fourth): Linear(in_features=256, out_features=128, bias=True)
  (bn5): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout5): Dropout(p=0.4, inplace=False)
  (layer_hidden_output): Linear(in_features=128, out_features=10, bias=True)
  (relu): ReLU()
)
```

Figure 4.29 Layered Architecture Of The Coffee Recommendation Model

The actual training was done both locally on my own GPU (RTX 3060), as well as on Google Colab (A100 Nvidia GPU), for 10.000 epochs. A batch size of 128 for the data loaders was adopted, meaning that the model will evaluate 128 samples at a given time, resulting in a tensor of shape (batch_size, nr_features), meaning (128, 7), for the X_train, while the shape of Y_train is (batch_size,), meaning (128,). When I get the raw logits from the model back, I convert them to probabilities using the softmax activation function, as it can be observed from Figure 4.30. The sum of the output classes must add to 1, representing the probability distribution over the classes, as seen on Figure 4.31.

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Figure 4.30 Softmax Formula [27]

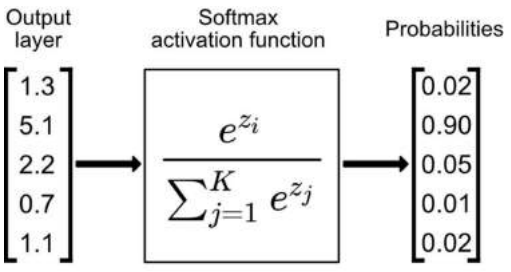


Figure 4.31 Softmax Example [28]

The training pipeline makes use of the Cross Entropy loss function, used for classification of multiple classes, where the optimizer is Stochastic Gradient Descent (SGD), with a learning rate of 0.1, and weight decay of 0.001. I've also tried to train the model with other hyperparameters, such as using Adam or RMSProp as the optimizer, but I saw a worse performance on the accuracy of the model. Figure 4.32 presents the formula for the Cross Entropy loss, which is used to optimize the model's performance on classification tasks by measuring the difference between the predicted and actual distributions.

$$H(y,p) = -\sum_{i=1}^{N} y_i log(p(y_i)) + (1-y_i) log(1-p(y_i))$$

Figure 4.32 Cross Entropy Loss Function [29]

At the end, we use the argmax function on the output of the softmax layer to determine the class with the highest probability. To update the model parameters effectively, we use the Stochastic Gradient Descent (SGD) optimizer, as seen in Figure 4.33. SGD helps us find a local or global minimum (what I would like) of the loss function, allowing the model to better approximate the target function, making use of the chain rule. This process is called backpropagation, where the algorithm moves backwards through the network to iteratively adjust the parameters, aiming to minimize the loss and find the optimal parameters.

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Figure 4.33 Stochastic Gradient Descent (SGD) [30]

The code for the training and evaluation can be visualized on Figure 4.34. Within the Pytorch framework, I need to switch to training mode with model.train(), keeping batch normalization and dropout active. During testing, model.eval() turns these off for consistent results.

```
    loss_fn : torch.nn.modules.loss.CrossEntropyLoss = nn.CrossEntropyLoss()
    optimizer : torch.optim.SGD = torch.optim.SGD(params=model.parameters(), lr=params_dict["LEARNING_RATE"],
                                                  weight_decay=params_dict["WEIGHT_DECAY"])
    for epoch in range(params_dict["EPOCHS"]):
        model.train()
        train_loss : int = 0
        train_acc : int = 0
        for _, (X_train, y_train) in enumerate(train_dataloader):
            X_train, y_train = X_train.to(device), y_train.to(device)
            y_logits = model(X_train)
            y_probs = torch.softmax(y_logits, dim=1)
            y_pred = y_probs.argmax(dim=1).to(torch.float32)
            loss = loss_fn(y_logits, y_train.to(torch.long))
            train_loss += loss
            acc = accuracy_fn(y_true=y_train, y_pred=y_pred)
            train_acc += acc
            optimizer.zero_grad()
            nn.utils.clip_grad_norm_(model.parameters(), params_dict["GRADIENT_CLIPPING_VALUE"])
            loss.backward()
            optimizer.step()
        train_loss /= len(train_dataloader)
        train_acc /= len(train_dataloader)
        model.eval()
        test_loss : int = 0
        test_acc : int = 0
        with torch.inference_mode():
            for X_test, y_test in test_dataloader:
                X_test, y_test = X_test.to(device), y_test.to(device)
                test_logits = model(X_test)
                y_test_probs = torch.softmax(test_logits, dim=1)
                test_pred = y_test_probs.argmax(dim=1).to(torch.float32)
                test_loss += loss_fn(test_logits, y_test.type(torch.long))
                test_acc += accuracy_fn(y_true=y_test, y_pred=test_pred)
            test_loss /= len(test_dataloader)
            test_acc /= len(test_dataloader)
        if (epoch+1) % 100 == 0:
            print(f"Epoch: {epoch} | train loss: {train_loss:.5f}, train acc: {train_acc:.2f}% | test Loss: {test_loss:.5f}, test Acc: {test_acc:.2f}%")
    path : str = os.path.join(os.path.dirname(__file__), "model_parameters.pickle")
    save_params(path, model)
```

Figure 4.34 Code For Model Training

- ☐ *Model inference:* For the inference, I created a small RESTful API, with a single route, using the Flask framework. In order to call the API, a POST request is made when the user creates an account for the first time, passing all the responses from the questionnaire to this microservice. It will output the top 5 favorite drinks back, saving them in a DynamoDB table, on AWS, where all the user information lies. Figure 4.35 shows us the DynamoDB table, with all the user information. This will be called inside the application when the user signs in, displaying the top 5 favorite drinks to the UI, as well as gifting them their favorite drink. Figures 4.36 and 4.37 showcase the expected outputs generated by the model.

```
    ],
    "id": "13",
    "name": "Marin",
    "password": "$2a$10$MhPEMLnlzy7Cv5ws3K7tuu8SrGXy1VWVCQ00r9LelGaGKJ56rMuRC",
    "username": "darieglanda@outlook.com"
  },
  {
    "favouriteDrinks": [
      {
        "drink 1": "Cold espresso"
      },
      {
        "drink 2": "Cold brew"
      },
      {
        "drink 3": "Americano"
      },
      {
        "drink 4": "Coretto"
      },
      {
        "drink 5": "Cortado"
      }
```


Figure 4.35 DynamoDB Table Entry With Top K Favorite Drinks

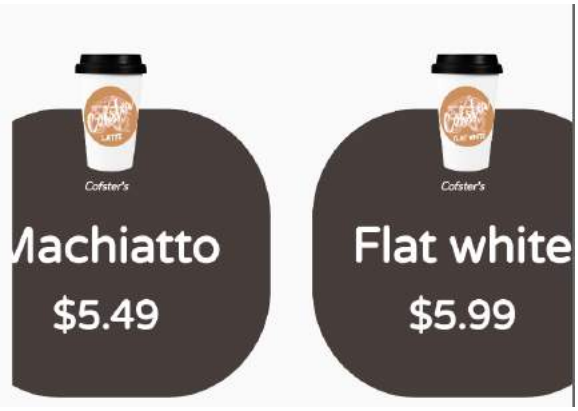


Figure 4.36 Top K Favorite Drinks


Figure 4.37 Free Gift

The actual pipeline for inference can be seen on Figure 4.38. One more important thing is that this microservice is also deployed to Docker, and the Dockerfile can be illustrated on Figure 4.39.

```python
@app.route("/prediction_drinks", methods=["POST"])
def prediction_drinks() -> json:
    if request.method == "POST":
        try:
            data : Dict[str, str] = request.get_json()
            data : Dict[str, str] = data["body"]
        except Exception as e:
            return jsonify({
                "statusCode" : 400,
                "body" : f"Error {e} when adding the user information"
            })
        if len(questions) != len(data):
            return jsonify({
                "statusCode" : 500,
                "body" : "Invalid server error"
            })
        predictions : List[int] = predict(data)
        predictions : Dict[str, str] = \
            {f"drink {i+1}": labels[pred] for i, pred in enumerate(predictions)}
        return jsonify({
            "statusCode" : 201,
            "favouriteDrinks" : predictions
            })
    return jsonify({
        "statusCode": 405,
        "message": "Method not allowed"
    })
```

```python
    out_responses : Dict[str, str] = {}
    standard_scalar_params : List[str] = read_standard_scalar_params(path_params_standard_scaler)
    for i, ((_, value), line_params) in enumerate(zip(responses.items(), standard_scalar_params)):
        mean_str, std_str = line_params.split()
        mean, std = np.float64(decimal.Decimal(mean_str)), np.float64(decimal.Decimal(std_str))
        params : Dict[str, str] = {"mean" : mean, "std" : std}
        out_responses[f"Question {i}"] = standard_scaler(value, params)

    X_test : torch.Tensor = torch.tensor(list(out_responses.values()), requires_grad=False, dtype=torch.float32)
    X_test : torch.Tensor = X_test.unsqueeze(0)

    nr_features : Tuple[int, int, int, int, int, int, int] = len(responses), 128, 256, 128, 256, 128, 10
    in_features, hid_features_hid_in, hid_features_hid_hid, hid_features_hid_hid_second, \
        hid_features_hid_hid_third, hid_features_hid_hid_fourth, out_features_hid_out = \
        cast(Tuple[int, int, int, int, int, int, int], nr_features)

    loaded_state_dict : OrderedDict = load_model(path_state_dict)
    model : DrinkClassifier = DrinkClassifier(in_features=in_features, hid_features_hid_in=hid_features_hid_in,
                                              hid_features_hid_hid=hid_features_hid_hid,
                                              hid_features_hid_hid_second=hid_features_hid_hid_second,
                                              hid_features_hid_hid_third=hid_features_hid_hid_third,
                                              hid_features_hid_hid_fourth=hid_features_hid_hid_fourth,
                                              out_features_hid_out=out_features_hid_out)
    model.load_state_dict(loaded_state_dict)

    model.eval()
    with torch.inference_mode():
        test_logits : torch.Tensor = model(X_test)
        y_test_probs : torch.Tensor = torch.softmax(test_logits, dim=1)
        if k == 1:
            prediction : torch.Tensor = y_test_probs.argmax(dim=1).to(torch.float32)
            return int(prediction.item())
        top_predictions : Tuple[torch.Tensor, torch.Tensor] = torch.topk(y_test_probs, k=k, dim=1)
        _, top_predicted_indecies = top_predictions
    return top_predicted_indecies.tolist()[0]
```

Figure 4.38 Code For Model Inference

```
1   FROM python:3.11
2
3   WORKDIR /usr/src/app
4   COPY requirements.txt .
5   RUN pip install --no-cache-dir -r requirements.txt
6   COPY . .
7   WORKDIR /usr/src/app/classifier
8
9   EXPOSE 8000
10  CMD ["python", "server.py"]
```

Figure 4.39 Coffee Recommender Microservice Dockerfile

### 4.3.4 AUTOREGRESSIVE COFFEE PROMPT UPDATER PIPELINE

The coffee updater pipeline consists of three microservices, where each of them communicate to one another, using the REST architecture. The microservices are deployed on Docker, and are run using Docker Compose, with all the other microservices there.

The actual architecture of the autoregressive coffee prompt updater pipeline can be viewed on Figure 4.42 and Figure 4.43. // this should be after (at the end), when explaining

The implementation is as follows: at the start, a popup window appears after the user drinks the coffee (the timer will wait approximately 30 minutes for this in production). It's worth noting that this will also work if you close the application (when you sign in again to the main page, it will check if the elapsed time has passed, and if it did, it will display that popup window there). This popup window is for you to rate the actual drink that you drank, in order for my Langchain agent to improve it. The popup window can be visualized on Figure 4.40.

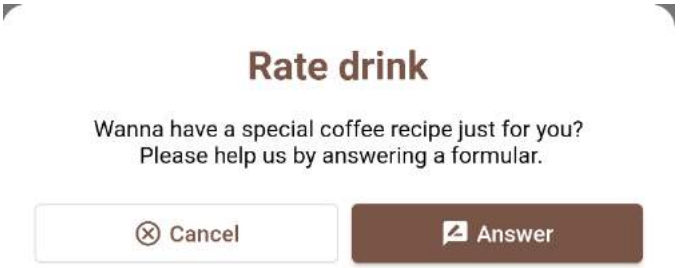


Figure 4.40 Popup For Custom Drink Updater

There, you will see two questions, where the sample questions can be visualized on Figure 4.41.

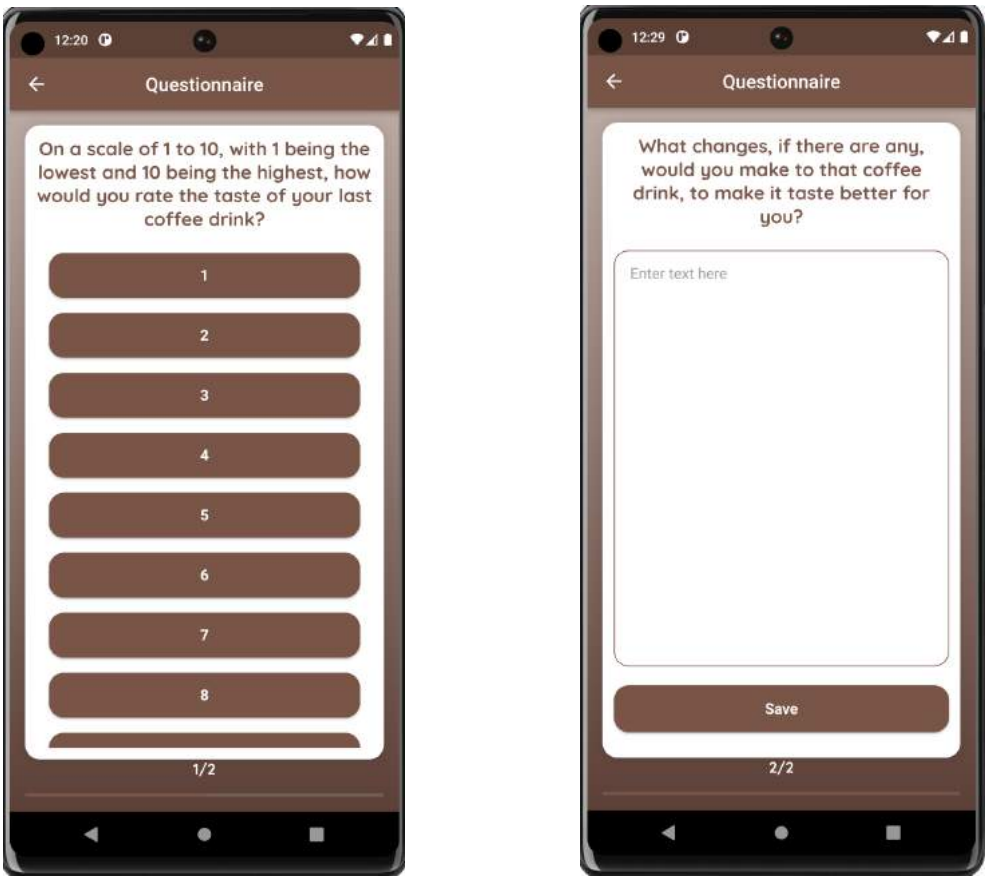


Figure 4.41 Drink Updater Questionnaire (Prompt Updater)

The answers collected, as well as the user name and the coffee that the user rates, will be passed to the questionnaire_llm_updater microservice using Mosquitto as a message broker.

Microservice questionnaire_llm_updater has two main responsibilities: preprocessing the data received from the frontend and saving the new response for a given user and drink in a PostgreSQL database using SQLAlchemy as the ORM. The database resides in a Docker container, utilizing Docker Volumes to persist all entries on the local server. The entity where we save all the entries is seen on Figure 4.42. This is the structure of it, consisting of the following fields: id, question_1, question_2, user_coffee, user_name and timestamp.

```python
Base : DeclarativeMeta = declarative_base()

class QuestionnaireEntity(Base):
    __tablename__ : str = "questionnaire"

    id : Column = Column(Integer, primary_key=True)
    question_1 : Column = Column(String)
    question_2 : Column = Column(String)
    user_coffee : Column = Column(String)
    user_name : Column = Column(String)
    timestamp : Column = Column(DateTime, server_default=func.now())

    def __str__(self) -> str:
        return (
            f"Person(id={self.id}, "
            f"question_1={self.question_1}, "
            f"question_2={self.question_2}, "
            f"user_coffee={self.user_coffee}, "
            f"user_name={self.user_name}, "
            f"created_timestamp={self.timestamp})"
        )
```

Figure 4.42 The Questionnaire Entity Holding All The User Responses

The abstract implementation for saving a new response is illustrated in Figure 4.43.

```python
def on_message(client : Any, userdata : Any, msg : Any):
    data : str = msg.payload.decode()
    LOGGER.info(f"Received data: {data} on topic {msg.topic}")
    cli_arguments_postgres : Dict[str, Any] = ArgumentParser.get_llm_updater_arguments_postgres()

    customer_name : str = get_mqtt_response_value(data=data, key_splitter="name")
    coffee_name : str = get_mqtt_response_value(data=data, key_splitter="coffee name")

    monad_preprocessing : MonadPreprocessing = MonadPreprocessing(args=None)
    spark_preprocessor_strategy : SparkPreprocessorStrategy = SparkPreprocessorStrategy(
        session_name=msg.topic, \
        data=data, \
        table_save_data="Questionnaire", \
        loader_db_dao_strategy=PostgresStrategyDAO( \
            database=cli_arguments_postgres["database"], \
            username=cli_arguments_postgres["username"], \
            password=cli_arguments_postgres["password"], \
            host=cli_arguments_postgres["host"], \
            port=cli_arguments_postgres["port"], \
        ) \
    )

    monad_preprocessing \
        .bind(callback=spark_preprocessor_strategy.start_session) \
        .bind(callback=spark_preprocessor_strategy.transform) \
        .bind(callback=spark_preprocessor_strategy.save) \
        .bind(callback=spark_preprocessor_strategy.stop_session)

    coffee_prompt : CoffeePrompt = CoffeePrompt()
    response_data : str = coffee_prompt.put( \
        base_url="http://ingredients-updater:8030", \
        endpoint="/coffee_recipe", \
        headers={"Content-Type" : "application/json"}, \
        customer_name=customer_name, \
        coffee_name=coffee_name, \
        llm_model_name=GPTModelType.GPT_3_5_TURBO_0125.value \
    )
    LOGGER.info(f"Response data: ${response_data}")
```

Figure 4.43 Code Implementation For Saving A New Response

After saving the new entry in the database, a RESTful API call will be made from this microservice, to the ingredients_updater one.

The ingredients_updater microservice orchestrates this pipeline. It takes the current prompt from a folder hierarchy, which consists of all users stored in the DynamoDB database, as input, along with the latest k responses from the entity, which represents the chat history. Figure 4.44 shows us the implementation of this.

```
def get_customer_responses(self, customer_name : str, coffee_name : str, limit_nr_responses : int = 10) -> Query[Tuple[DateTime, String, String]]:
    """
    Retrieves the customer responses from the questionnaire table.

    Args:
        customer_name (str): The name of the customer.
        limit_nr_responses (int): The maximum number of responses to retrieve (default is 10).

    Returns:
        List[Tuple[DateTime, String, String]]: A list of tuples containing responses.
    """
    try:
        questionnaire_table = self.__get_questionnaire_table()

        if questionnaire_table is None:
            raise NoSuchTableError(f"Table {self.table_name} could not be found in the reflected metadata.")

        person_coffee_query : Query[Tuple[DateTime, String, String]] = self.session.query(
            questionnaire_table.c.timestamp,
            questionnaire_table.c.question_1,
            questionnaire_table.c.question_2
        ).filter(
            questionnaire_table.c.user_name == customer_name,
            questionnaire_table.c.user_coffee == coffee_name,
        ).order_by(desc(questionnaire_table.c.timestamp))\
        .limit(limit_nr_responses)
        return person_coffee_query.all()
    except NoSuchTableError as table_error:
        raise table_error
```

Figure 4.44 Code Implementation To Fetch The Latest K Responses

For the chat history, I add one more field to each of the responses that represents the importance of those. Newer responses should be more important than older ones, assigning some kind of probability for them. For this, I made an equation inspired by the famous Bellman one, commonly used in Reinforcement Learning to determine the optimal policy. Figure 4.46 shows us the actual formula how it looks like:

$$T = \{(e_i, P_0 \times \gamma^{i-1})\}_{i=1}^{n}$$

Figure 4.45 Formula Inspired By The Bellman Equation

ei is a response in the list of all the responses, P0 is the initial probability value, while gamma represents the discount factor, similar as in the Bellman Equation. The output will represent this trajectory, which is a list of tuples, with all the fields that were before and the addition of this new response importance field.

Figure 4.46 displays us how the chat history will look like after performing this form of the Bellman equation on it.

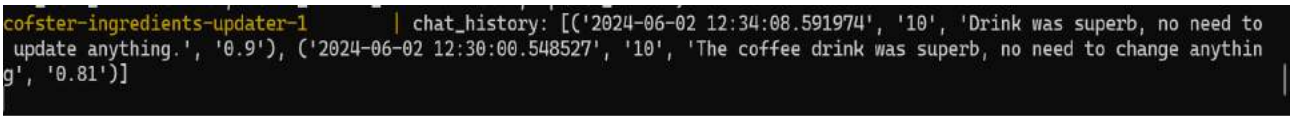


Figure 4.46 Chat History For Langchain Agent

The chat history and a Large Language Model (can use any GPT model available from OpenAI) are used for the Langchain Agent. I use the Retrieval Augmented Generation, or RAG in short, technique in order to give additional information to the Large Language Model. So, this pipeline will take in a Large Language Model, and a chat history, from whatever datastore, in my case it is the PostgreSQL database, and for the Large Language Model, we pass the previous prompt as a parameter as well. This will produce a prompt that has all that information in mind, updating the prompt each time in this manner.

Figure 4.47 shows us how the RAG pipeline works on my implementation.

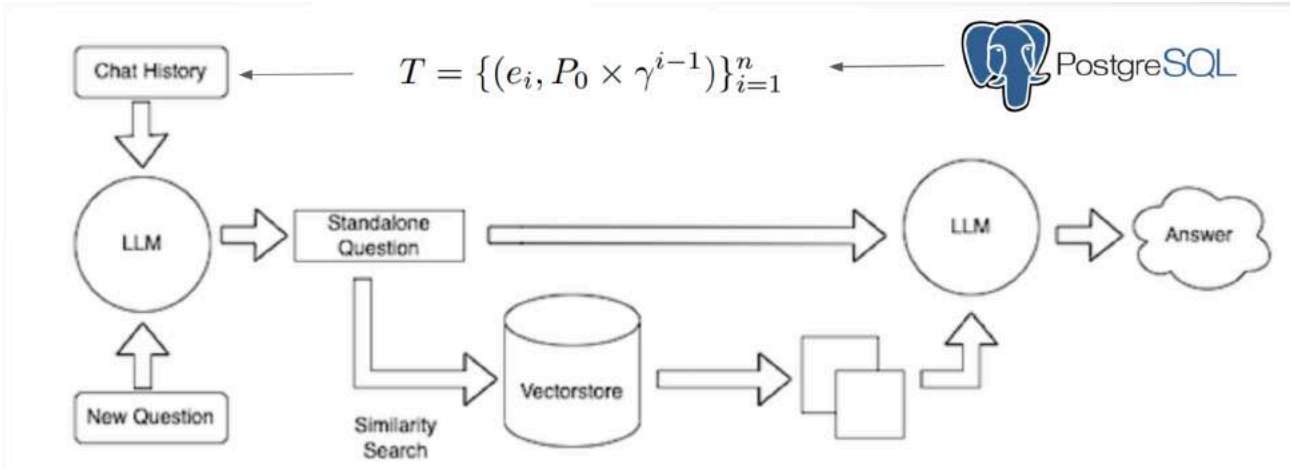


Figure 4.47 Retrieval Augmented Generation To Update Prompt [31]

The code to update the prompt can be seen on Figure 4.48.

```python
def __generate_available_ingredients(self, prompt : str, \
                                     model : str = "gpt-3.5-turbo", \
                                     temperature_prompt : float = 0, \
                                     chat_history : List[Any] = []) -> Dict[str, str]:
    """
    Helper method to generate a response to a query using the OpenAI model.

    Args:
        query (str): The input query or prompt.
        model (str): The OpenAI model to use (default is "gpt-3.5-turbo").
        temperature_prompt (float): The temperature for controlling response randomness (default is 0).
        chat_history (List[Any]): Some past information for the LLM (default is an empty List).

    Returns:
        Dict[str, str]: The generated response.
    """
    text_loader : TextLoader = TextLoader(file_path=self.file_path)
    index : VectorStoreIndexWrapper = VectorstoreIndexCreator().from_loaders([text_loader])
    chain : ConversationalRetrievalChain = ConversationalRetrievalChain.from_llm(
        llm=ChatOpenAI(model=model, temperature=temperature_prompt),
        retriever=index.vectorstore.as_retriever(search_kwargs= {"k": 1}),
    )
    result : Dict[str, str] = chain({"question": prompt, "chat_history": chat_history})
    return result["answer"]
```

Figure 4.48 Code For RAG To Update Prompt

The final microservice, user_file_prompt_updater, has a singular purpose: it manages a tree-like folder structure to store prompts associated with specific users and coffee names. This microservices makes use of two threads, where on one thread is a CRON job that fetches all the user updates (for example if a user is added or deleted), and it will be triggered every 8 minutes. The other thread is responsible for running a Flask Web Server in order to receive traffic from the ingredients_updater microservice, and update the tree-like folder structure with new prompts for the LLM that will create the recipe at the end. Having in mind that this microservice is deployed to Docker, it will utilize a Docker Volume to retain updates within the folder hierarchy. This approach ensures that content updates occur within Docker without erasing existing data each time.

A snippet of this microservice's code is illustrated in Figure 4.49.

```python
if customer_name is None:
    return jsonify({
        "error_message" : "Customer name is missing in the query parameters"
    }), 400

user_prompt_generator : UserPromptGenerator = UserPromptGenerator(root_path=ROOT_PATH)

user_prompt_information : Dict[str, Any] = user_prompt_generator.get_user_prompt_file_information( \
    customer_name=customer_name.lower(), \
    coffee_name=coffee_name \
)

if "content" not in user_prompt_information:
    return jsonify({
        "error_message" : "Could not fetch the user prompt content.",
        "error_information" : f"Error information: {user_prompt_information['error_message']}"
    }), 500

return jsonify(user_prompt_information), 200
```

Figure 4.49 User_file_prompt_updater Microservice

The folder hierarchy can also be visualized on Figure 4.50.


Figure 4.50 User_file_prompt_updater Folder Hierarchy

Considering the limitations of space and scope, it was impractical to delve into every aspect of the autoregressive prompt updater's code implementation and intricacies. However, I've developed a concise software architecture that offers a glimpse into the underlying workings of the pipeline. To gain a deeper insight, please check Figures 4.51 and 4.52, which provide a visual representation of the system's architecture and workflow.

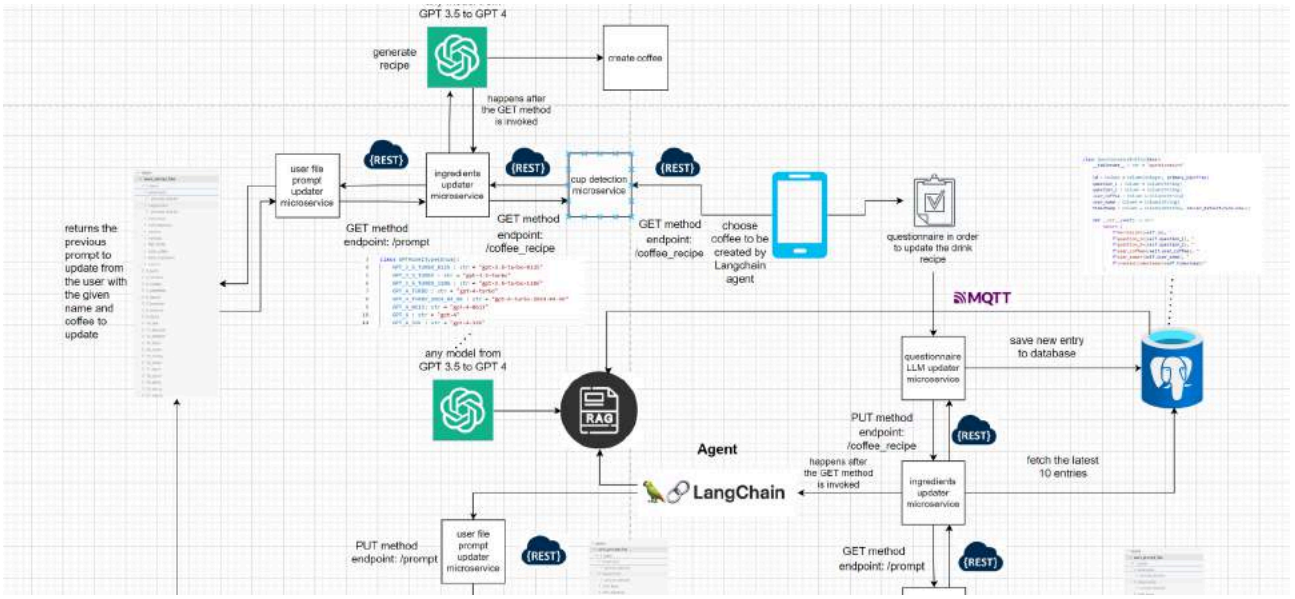


Figure 4.51 Autoregressive Coffee Prompt Updater Pipeline (1)

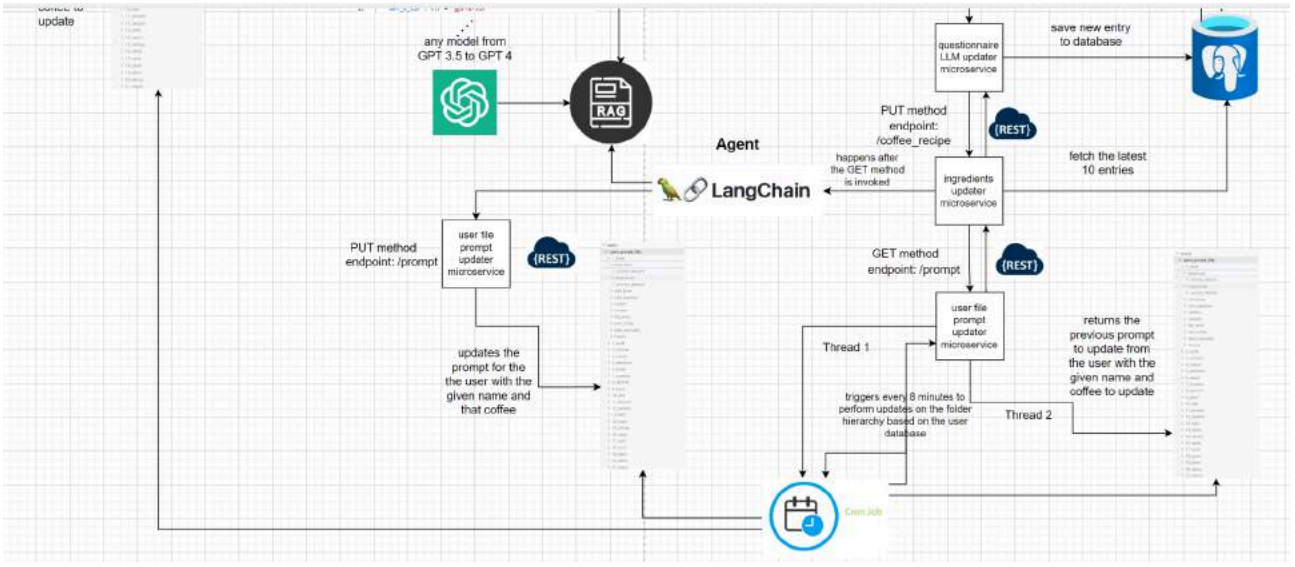


Figure 4.52 Autoregressive Coffee Prompt Updater Pipeline (2)

### 4.3.5 CUP DETECTION MICROSERVICE

Once the user completes the payment for the drink via Stripe, the system seamlessly transitions to the cup detection pipeline. This pipeline comprises two critical detection tasks: identifying the custom-designed cup and determining the region of interest (ROI) for the drip area.

State-of-the-art (SOTA) detectors, namely YOLOv8 and YOLOv9, are employed to detect the custom cup. The choice between YOLOv8 and YOLOv9 is facilitated by a factory design pattern, allowing for the selection of the desired algorithm for this specific task.

#### 4.3.5.1 DATASET

The dataset comprises 540 images of my custom cups. In those 540 images, it involves horizontal flipped images, images that are saturated and so on. For the image labeling, as well as to generate the dataset, I have used the help of Roboflow, which is commonly used with the YOLO models. It offers a simple way to tackle this problem. The output images at the end, are resized to have a size of (3, 640, 640).

Figures 4.53 and 4.54 show us an example of how the image labeling process works with Roboflow.

Figure 4.53 Image Labeling (1)

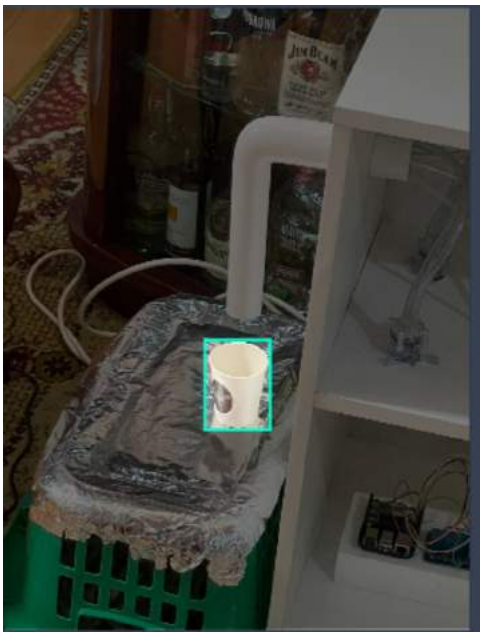

Figure 4.54 Image Labeling (2)

### 4.3.5.2 MODEL TRAINING

The training was conducted on a RTX 3060 GPU, on my own server. Both of the models are trained with the exact same hyperparameters using Transfer Learning, with the model parameters already pre-trained on the MS COCO Dataset. The first 5 layers are freezed, again on both of the models, keeping consistency between them.

The hyperparameters utilized are as follows: training on for 400 epochs with a early stopping after 50 epochs, a batch size of 16, 16 workers, label smoothing of 0.05, an optimizer in the form of AdamW, dropout between the layers of 0.4, and a weight decay of 0.01.

```python
def freeze_layers(trainer, num_freeze : int = 5) -> None:
    """
    Freeze layers in the YOLO model.

    Args:
        trainer: The YOLO trainer object.
        num_freeze (int, optional): The number of layers to freeze. Defaults to 5.
    """
    model : YOLO = trainer.model
    print(f"Freezing {num_freeze} layers")
    freeze : list[str] = [f'model.{x}.' for x in range(num_freeze)]
    for k, v in model.named_parameters():
        v.requires_grad = True
        if any(x in k for x in freeze):
            print(f"freezing {k}")
            v.requires_grad = False
    print(f"{num_freeze} layers are freezed.")

def main() -> None:
    """
    Main function to train the YOLO model for custom cup detection.

    Returns:
        None
    """
    build_path : str = find_single_matching_path(build_path=os.path.join(os.getcwd()))
    assert " " not in build_path, build_path.split("/")[-1]
    build_path : str = find_single_matching_path(build_path, "roboflow_key.json")
    assert " " not in build_path, build_path.split("/")[-1]
    yolo_cred : Dict[str, str] = read_credentials(build_path)

    rf : Roboflow = Roboflow(api_key=yolo_cred["api_key"])
    project : Project = rf.workspace("universitatea-politehnica-timioara").project("custom-cup-detection")
    project.version(3).download("yolov8")

    model : YOLO = YOLO("yolov8n.pt")
    model.add_callback("on_train_start", freeze_layers)
    model.train(
        data=os.path.join(os.getcwd(), "Custom-Cup-Detection-3", "data.yaml"),
        epochs=400,
        batch=16,
        workers=16,
        pretrained=True,
        cache=True,
        label_smoothing=0.05,
        optimizer="AdamW",
        dropout=0.4,
        weight_decay=0.01,
    )
```

Figure 4.55 Code For Cup Detection Training

### 4.3.5.3 MODEL RESULTS

For the results, I managed to get for the YOLOv8 a training box loss of 0.50846, after approximately 100 epochs, because as I said, early stopping was enabled after 50. The FPS that it is achieved when running inference on my RTX 3060 GPU, is approximately 30 FPS. For the YOLOv9, I managed to get a better training box loss of 0.29018, but a big FPS drop, around 12,

Figure 4.56 and Figure 4.57 show us the box loss for both of the models, while Figure 4.58 and Figure 4.59 show us the confusion matrix for them.

For the FPS, we can see the results of the detection on Figures 4.60 and 4.61.

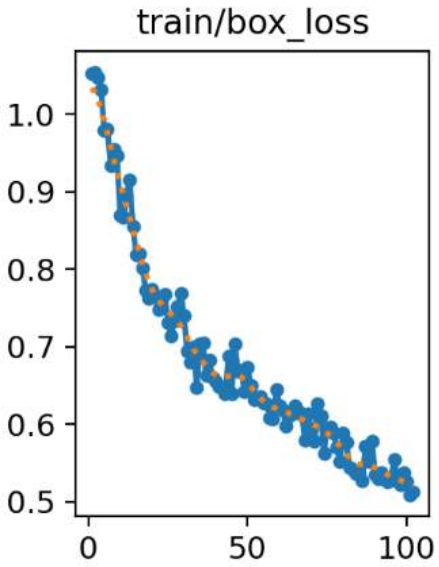


Figure 4.56 YOLOv8 box loss

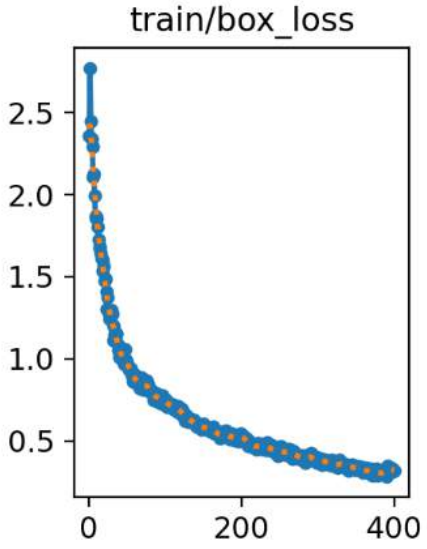


Figure 4.57 YOLOv9 box loss

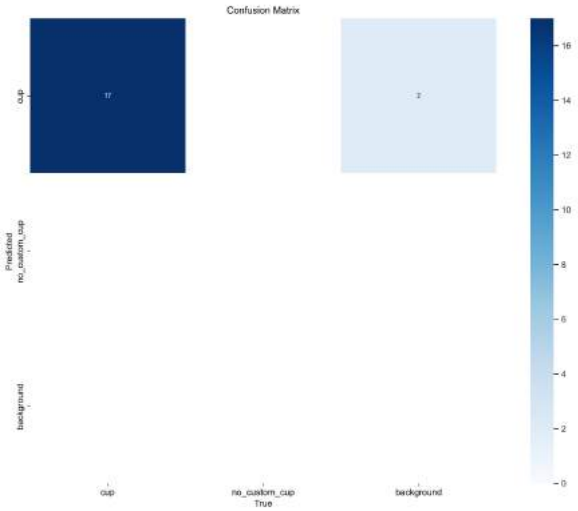


Figure 4.58 YOLOv8 Confusion Matrix

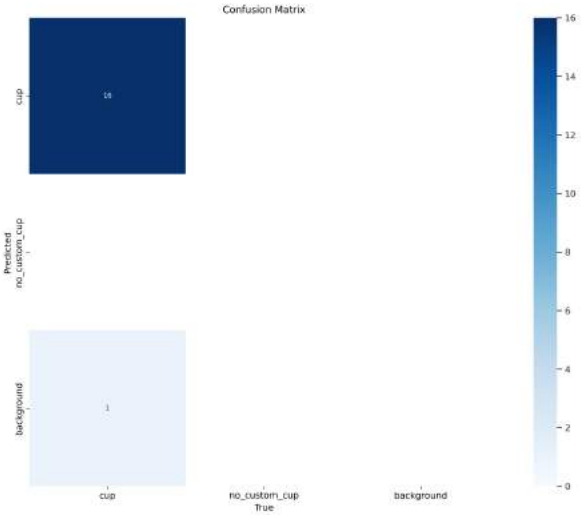


Figure 4.59 YOLOv9 Confusion Matrix

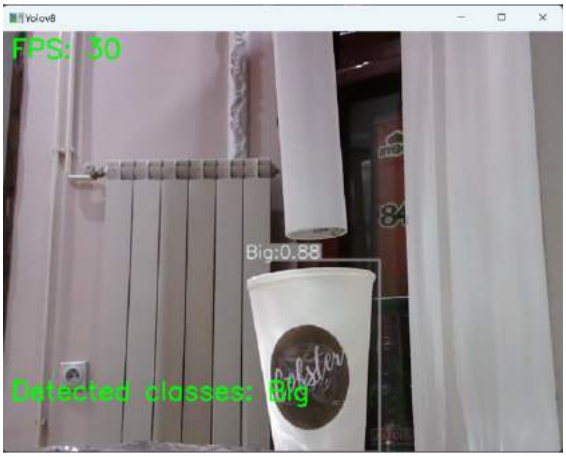


Figure 4.60 FPS YOLOv8

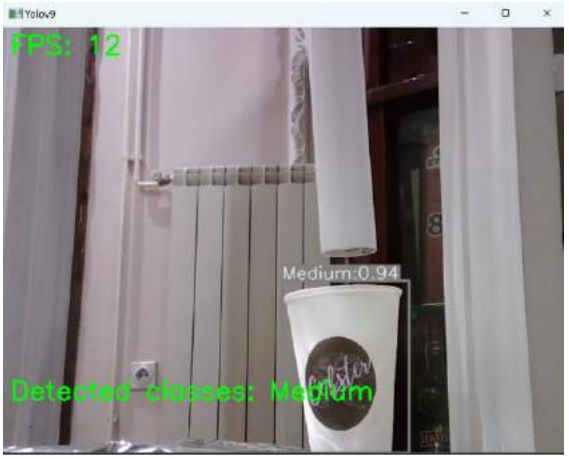


Figure 4.61 FPS YOLOv9

We can also visualize on Figure 4.62 some results from the model training on the YOLOv8 model.

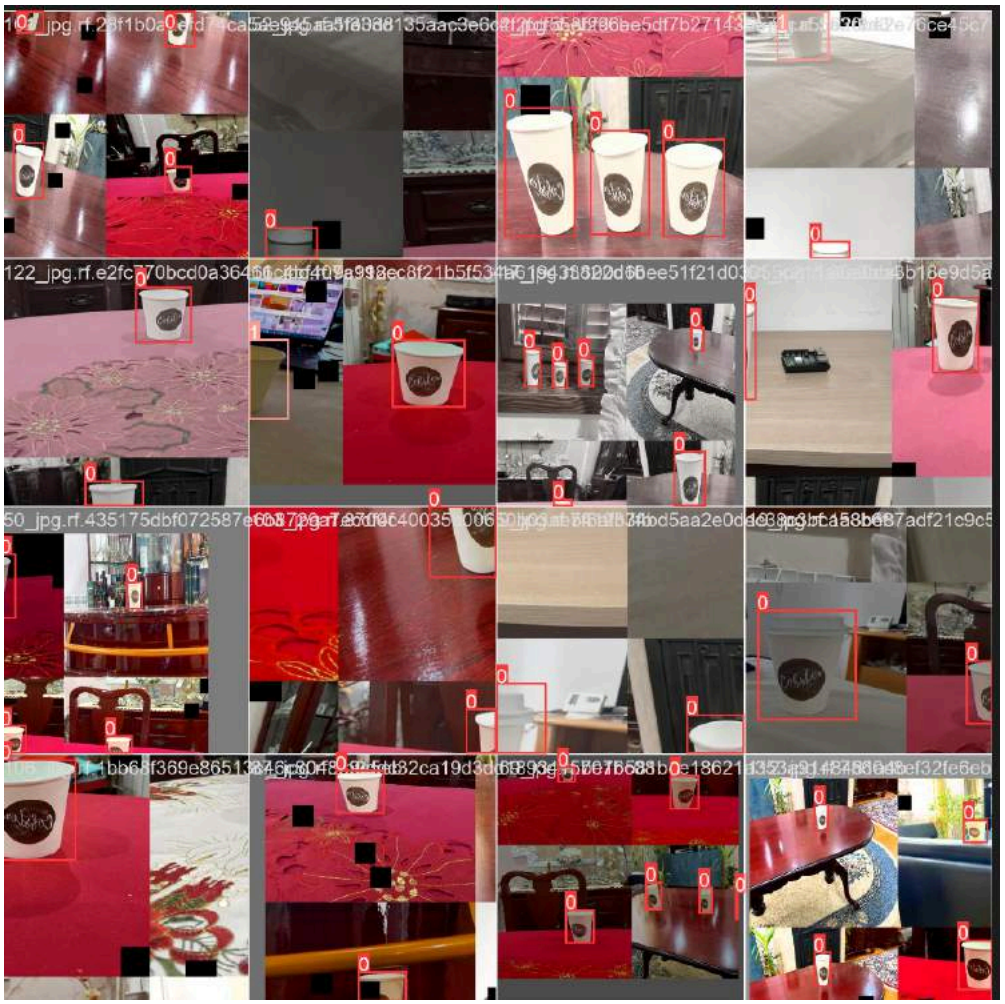

Figure 4.62 Visual Results For YOLOv8 From Model Training

Given the uniqueness of my custom cup detection model and the innovative pipeline I developed, I was unable to find directly comparable studies. Therefore, I opted to compare my results with state-of-the-art papers focused on the broader task of custom object detection. Table 4.2 provides a detailed comparison of my model's performance against these leading studies.

| | My YOLOv8 model | My YOLOv9 model | YOLOv3 [32] | YOLOv4 [32] | YOLOv8 [33] |
|---|---|---|---|---|---|
| **FPS** | **30** | **13** | **55** | **49** | **98** |
| **mAP** | **92.33%** | **93.44%** | **95.65%** | **96.06%** | **90.7%** |
| **GPU for inference** | **RTX 3060** | **RTX 3060** | **GTX 3060** | **GTX 3060** | **RTX 3090** |

Table 4.2 Comparisons Between My Models With Other SOTA Ones

The paper [32] delves into real-time cup detection methods, extending its capabilities by not only detecting cups but also classifying which specific cup is which. In contrast, my approach distinguishes itself by focusing solely on cup detection in the initial stage. The classification of each cup is subsequently performed by calculating the distance from the camera to the cup. This distance-based classification is refined through a thresholding process, enabling precise identification of each cup. Paper [33] is used for road defect detection,

Our model achieved results comparable to state-of-the-art benchmarks in terms of mean Average Precision (mAP). However, it slightly underperformed in frames per second (FPS). This discrepancy is primarily because paper [32] utilizes more lightweight models, such as YOLOv3 and YOLOv4, compared to my implementation of YOLOv8 and YOLOv9. Additionally, paper [33] leverages a more powerful GPU, the RTX 3090 one, contributing to higher FPS. The drop in FPS also appeared because of my complex pipeline, using multiple threads and calls in between the cup detection process.

It's also noteworthy that my models were not trained for an extensive number of epochs due to early stopping mechanisms, which frequently halted the training process, particularly for the YOLOv8 model.

### 4.3.5.4 INFERENCE PIPELINE

For the inference, I employed a pipeline that consists of both the custom cup detection, as well as the drip area. Both of them need to be detected at the same time in order for the coffee creation to start.

To detect the drip area, implementation wise, I leverage the region of interest (ROI) technique for the drip area, focusing on the top of the cup along the y-axis. The region of interest focuses on finding a drip area with a certain area size and a certain color, while comparing the area with a threshold value.

The drip area detection can be seen on Figure 4.63.

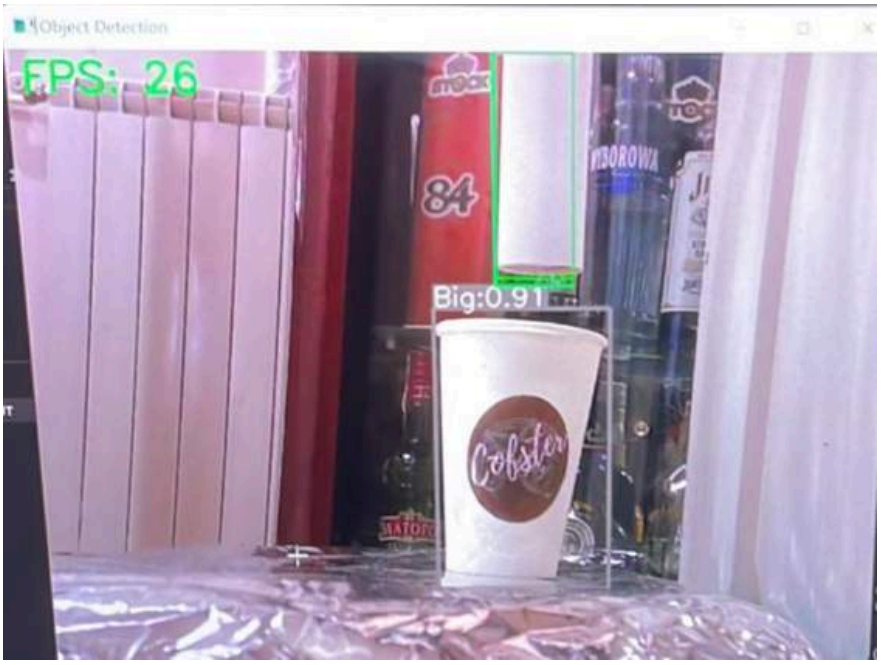


Figure 4.63 Drip Area Detection

A snippet of the code implementation for the drip area can be visualized on Figure 4.64 and Figure 4.65.

```
roi_frame : Optional[np.ndarray] = None
if len(classes_coordinates) == 1:
    detected_pipe, frame, roi_frame, ignore_frame = pipe_detector_builder_service \
        .create_roi_subwindow(frame=frame, classes_coordinates=classes_coordinates) \
        .find_white_pipe(draw=True) \
        .collect()
```

```python
sys.path.append("../")

from utils.constants import (
    MIN_AREA_PIPE,
    MIN_ASPECT_RATION_PIPE,
    THRESHOLD_WHITE_PIXELS,
    THRESHOLD_MEAN_WHITE,
    COORDINATE_NAMES,
    COORDINATE_OFFSETS
)

class PipeDetectorBuilderService:
    """
    Service class for building and detecting pipes in images.
    """
    def __init__(self):
        """
        Initialize the PipeDetectorBuilderService.
        """
        self.__frame : Optional[np.ndarray] = None
        self.__roi_frame : Optional[np.ndarray] = None
        self.__contours : Optional[np.ndarray] = None
        self.__white_pipe_found : Optional[bool] = None
        self.__ingore_frame : Optional[bool] = None

    def create_roi_subwindow(self, frame : np.ndarray, classes_coordinates : List[float]) -> Self:
        """
        Create a region of interest (ROI) subwindow based on provided class coordinates.

        Args:
        - frame (np.ndarray): The input image frame.
        - classes_coordinates (List[float]): List of coordinates for different classes.

        Returns:
        - Self: The instance of PipeDetectorBuilderService.
        """
        roi_subwindow : Dict[str, int] = {}

        for class_coordinates in classes_coordinates:
            for class_coordinate, coordinate_name in zip(class_coordinates, COORDINATE_NAMES):
                offset : int = COORDINATE_OFFSETS.get(coordinate_name, 0)
                roi_subwindow[coordinate_name] = max(0, round(class_coordinate) - offset)

        self.__frame : np.ndarray = frame.copy()
        self.__roi_frame = self.__frame[
```

Figure 4.64 Code Drip Area Detection (1)

```python
        return self

    def find_white_pipe(self, draw : bool = False) -> Self:
        """
        Find white pipes in the region of interest (ROI) frame.

        Args:
        - draw (bool): Optional parameter whether to draw rectangles around detected pipes.

        Returns:
        - Self: The instance of PipeDetectorBuilderService.
        """
        roi_w, roi_h, _ = self.__roi_frame.shape
        if roi_w == 0 or roi_h == 0:
            self.__ingore_frame = True
            return self

        gray_roi : np.ndarray = cv2.cvtColor(self.__roi_frame, cv2.COLOR_BGR2GRAY)

        _, binary_roi = cv2.threshold(gray_roi, THRESHOLD_WHITE_PIXELS, 255, cv2.THRESH_BINARY)
        kernel : np.ndarray = np.ones((3, 3), np.uint8)
        binary_roi : np.ndarray = cv2.dilate(binary_roi, kernel, iterations=1)

        self.__contours, _ = cv2.findContours(binary_roi, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        self.__ingore_frame = False
        for contour in self.__contours:
            area : float = cv2.contourArea(contour)
            x, y, w, h = cv2.boundingRect(contour)
            aspect_ratio : float = float(w) / h
            if area > MIN_AREA_PIPE and aspect_ratio >= MIN_ASPECT_RATION_PIPE:
                roi_color : np.ndarray = self.__roi_frame[y:y+h, x:x+w]
                mean_color : np.float64 = np.mean(roi_color, axis=(0, 1))
                if np.any(mean_color > THRESHOLD_MEAN_WHITE):
                    if draw:
                        self.__roi_frame = cv2.rectangle(self.__roi_frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
                    self.__white_pipe_found = True
                    return self
        self.__white_pipe_found = False
        return self

    def collect(self) -> Tuple[bool, np.ndarray, np.ndarray, bool]:
        """
        Collect the results of pipe detection.
```

Figure 4.65 Code Drip Area Detection (2)

For cup size detection, I used a non-Deep Learning approach. By calculating the distance from the cup to a fixed camera and comparing it to predefined thresholds, the system categorizes the cup size as Small, Medium, or Big, as shown in the previous figures. Sample code for this can be seen in Figure 4.66.

```python
    def get_cup_class_name(self, cup_height : float, thresholds : Dict[int, CupClasses]) -> str:
        """
        Get the cup class name based on the cup height.

        Args:
            cup_height (float): Height of the cup.
            thresholds (Dict[int, CupClasses]): Thresholds mapping to cup classes.

        Returns:
            str: Cup class name.
        """
        low_bound_error : float = cup_height - (CUP_ERROR_TOLERANCE_PERCENTAGE / 100) * cup_height
        high_bound_error : float = cup_height + (CUP_ERROR_TOLERANCE_PERCENTAGE / 100) * cup_height

        for threshold, cup_class in thresholds.items():
            if low_bound_error <= threshold <= high_bound_error:
                return cup_class.value

        return UNKNOWN_CLASS
```

Figure 4.66 Code To Categorize Cup Size

In order for the client to view the camera's perspective, I created a pipeline that produces the frames using a Kafka producer, featuring a single partition and a single replica, a Kafka consumer, WebSocket, and a React-based client. Additionally, these services are containerized. Kafka was employed to decouple the services and facilitate horizontal scaling, thus enabling the addition of multiple clients without any issues.

A snippet of the code implementation can be seen from Figures 4.67, 4.68 and 4.69

```
    kafka_producer : KafkaProducer = KafkaProducer(
        bootstrap_servers=BOOTSTRAP_SERVERS,
    )

_, encoded_frame = cv2.imencode(".jpg", frame)
image_bytes : bytes = encoded_frame.tobytes()

kafka_producer.send(topic=TOPIC_NAME, value=image_bytes)
kafka_producer.flush()

start_fps_time = end_fps_time

cv2.imshow(WINDOW_NAME_CUP_DETECTION, frame)
cv2.waitKey(1)

success, frame = camera.read()
if drink_finished.get_state():
    drink_finished.set_state(False)
    break
```

Figure 4.67 Kafka Frame Producer

```
18  async def frame_consumer(websocket : websockets.server.WebSocketServerProtocol) -> Coroutine[Any, Any, Coroutine]:
19      consumer : KafkaConsumer = KafkaConsumer(
20          TOPIC_NAME,
21          bootstrap_servers=BOOTSTRAP_SERVERS,
22          auto_offset_reset=AUTO_OFFSET_RESET
23      )
24
25      try:
26          for message in consumer:
27              image_bytes : bytes = message.value
28              await websocket.send(image_bytes)
29      finally:
30          consumer.close()
31
32  async def connect_and_consume() -> None:
33      async with websockets.serve(frame_consumer, "localhost", 8765):
34          await asyncio.Future()
```

Figure 4.68 Kafka Consumer And WebSocket Connection

```
7  export const FrameRenderer : React.FC<FrameRendererProps> = ({ websocketUrl }) => {
8    const [imageSrc, setImageSrc] = useState<string | null>(null);
9    const [isError, setIsError] = useState<boolean>(false);
10
11   useEffect(() => {
12     const socket : WebSocket = new WebSocket(websocketUrl);
13
14     socket.addEventListener('open', _ => {
15       socket.send('Connection established')
16     });
17
18     socket.addEventListener('message', async (event) => {
19       if (event.data instanceof Blob) {
20         const reader : FileReader = new FileReader();
21         reader.onload = () => {
22           const base64Frame : string = reader.result as string;
23           setImageSrc(base64Frame);
24         };
25         reader.readAsDataURL(event.data);
26       }
27     });
28
29     socket.addEventListener('error', (error) => {
30       if (isError) {
31         console.error('WebSocket error:', error);
32         toast.error('WebSocket connection error');
33       }
34       setIsError(true);
35     });
36
37     socket.addEventListener('close', () => {
38       console.log('WebSocket connection closed');
39     });
40
41     return () => {
42       socket.close();
43     };
44   }, [websocketUrl]);
45
```

Figure 4.69 React Frame Renderer Component

Figure 4.70 illustrates the pipeline's output (the cup detection on the web application). The Kafka producer captures frames from OpenCV with horizontal scaling enabled, allowing for the use of an unlimited number of Kafka consumers. Currently, there is only one Kafka consumer, but adding additional consumers should be straightforward. Each consumer utilizes a WebSocket for direct full-duplex communication with the frontend, facilitating seamless backend-frontend interaction. Presently, there is a single client, which is a React website.

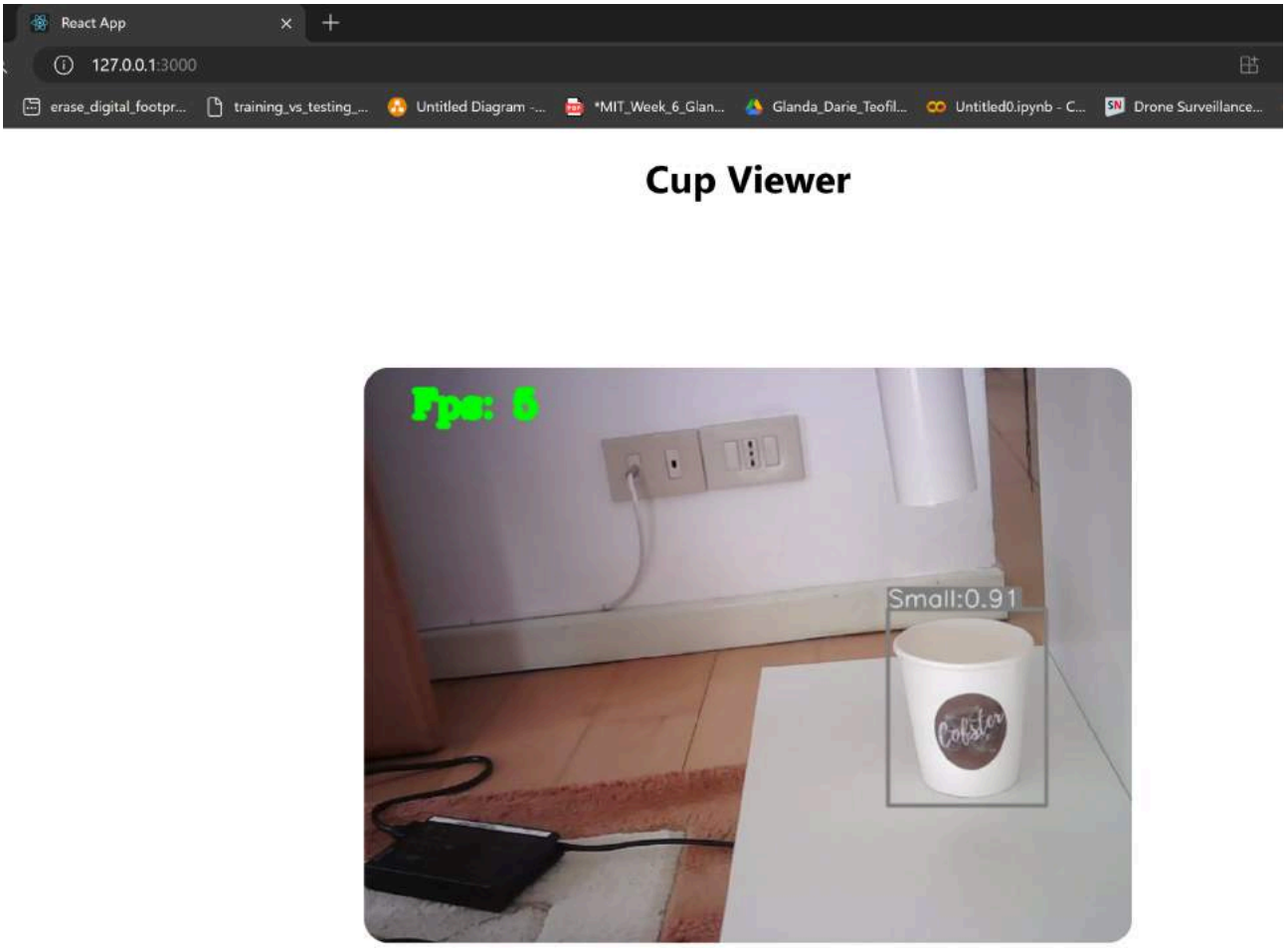


Figure 4.70 The React Cup Viewer Page

As mentioned earlier, detection stops once the cup and the drip area are continuously detected for 10 seconds. At that point, a timer starts. If the timer completes without interruption, an API request is sent to initiate the coffee creation microservice. If either the cup or the drip area is not detected in the frame, the timer resets and the process restarts until both are detected for 10 consecutive seconds. This ensures that the coffee creation process only begins when the cup is correctly positioned, preventing any potential accidents from happening.

### 4.3.6 COFFEE CREATION MICROSERVICE

After the cup and drip detection finishes, it will call this microservice for the coffee creation, which is written using the FastAPI framework. The sole purpose of this one here is to create the coffee, using the hardware that I specified previously.

At the center of this microservice, to orchestrate the coffee creation, lies a Raspberry Pi Model 4 with 8 GB RAM. He sends commands to the switches from the relay, in order for them to trigger the 12V pumps. The pumps are responsible to pull liquid and really light solid, from one pipe to another one, navigating the coffee creation in this manner.

This microservice is not inside a container, it runs as a standalone process, but it has a Docker file if the user would want to change the code later on. Having in mind that this microservice is on another server, the communication between those 2 services should be done with the IP, if a DNS is not bought for the coffee creation microservice. Figure 4.71 and 4.72 show us the code of how the communication is achieved.

```
9   class CoffeeCreationSimpleFacadeService:
10      @staticmethod
11      def create(payload : Dict[str, Any]) -> str:
12          coffee_creation_service_response : str = CoffeeCreationController(CoffeeCreationService()) \
13              .create_coffee(
14                  base_url="http://192.168.0.192:5000/coffee",
15                  endpoint="create",
16                  verbose=False,
17                  payload=payload
18          )
19          return coffee_creation_service_response
```

Figure 4.71 Request From Cup Detection Microservice (Flask Framework)

```
microservices > coffee_creation > app.py > ...
1   import uvicorn
2   from fastapi import (
3       FastAPI,
4   )
5   from controllers.coffee_controller import CoffeeController
6   from services.coffee_service import CoffeeService
7   from routes.coffee_router import router as coffee_router
8
9   app : FastAPI = FastAPI()
10
11  app.include_router(coffee_router, prefix="/coffee")
12  app.dependency_overrides[CoffeeController] = lambda: CoffeeController(CoffeeService())
13
14  if __name__ == "__main__":
15      uvicorn.run(app, host="192.168.0.192", port=5000)
```

Figure 4.72 Coffee Creation Microservice (FastAPI Framework)

In order for the server to run on boot, whenever the Raspberry Pi is powered, I create a new .service file in the */etc/systemd/system* directory. Figure 4.73 illustrates the content of the .service file.

```
[Unit]
Description=Coffee creation process
After=network-online.target

[Service]
WorkingDirectory=/home/darie/Desktop/cofster/coffee_creation/
ExecStart=/home/darie/Desktop/cofster/coffee_creation/coffee-creation-venv/bin/python \
/home/darie/Desktop/cofster/coffee_creation/app.py > \
/home/darie/Desktop/cofster/coffee_creation/boostrap_service_logging.log 2>&1
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Figure 4.73 The .Service File To Run Process On Boot

To actually start the the FastAPI server on boot, we need to enable it using the following command, which is illustrated on Figure 4.74:

```
sudo systemctl enable boostrap-coffee-making-service.service
```

Figure 4.74 Running The WebServer On Boot

For the software architecture implementation, I am following a simplified Domain-Driven Design (DDD) approach. This simpler version happened to be, because my code does not involve any databases, making the design of it simpler. The following folder structure was used, illustrated on Figure 4.75.

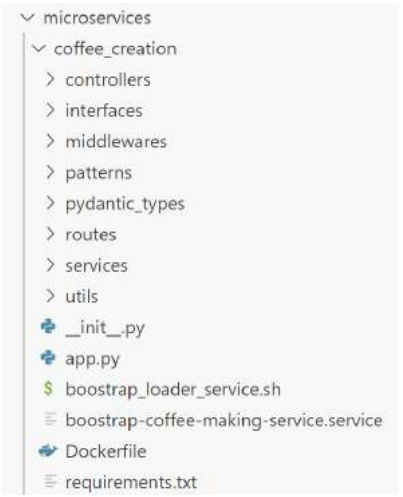


Figure 4.75 Folder Structure Coffee Creation Microservice

Each drink has its own recipe, with its own drink size, for example if the user would choose a cortado drink, with a medium size, it would trigger all the pumps to fetch the liquid for that given drink. In order to run multiple ingredients independently, without waiting

for each pump to finish executing, in a sequential manner, I add threading to this, running each ingredient in a separate thread.

The class implementation code for the parallel ingredient runner, can be visualized on Figure 4.76, while the coffee creator for the cortado drink, with the parallel ingredient runner passed as an dependency, that was taken as an example, can be seen on Figure 4.77.

```python
class ParallelCoffeeCreatorService(TypeCoffeeCreatorService):
    def __init__(self):
        self.workers : Dict[str, Thread] = {}

    def __len__(self) -> int:
        return len(self.workers)

    def __eq__(self, other : str) -> bool:
        return CoffeeCreatorType.PARALLEL.value == other

    def __getitem__(self, key : str) -> Thread:
        return self.workers[key]

    def start_workers(self, data : Dict[str, Any], module : ModuleType, pins: List[int]) -> Dict[str, Thread]:
        pin_index : int = 0
        for thread_id, (key, value) in enumerate(data.items()):
            if match(INGREDIENT_WITH_LAST_CHARACTER_NUMBER, key) is not None:
                ingredient : str = value.lower().replace(" ", "_")
                if hasattr(module, ingredient):
                    target : Callable = getattr(module, ingredient)
                    params : Tuple[int] = (pins[pin_index],)
                    worker_name : str = f"thread{thread_id + 1}"
                    self.workers[worker_name] = Thread(target=target, args=params)
                    self.workers[worker_name].daemon = True
                    self.workers[worker_name].start()
                    pin_index += 1
        return self.workers

    def wait_for_workers_to_finish(self):
        for worker_value in self.workers.values():
            worker_value.join()

    def __str__(self) -> str:
        return f"ParallelCoffeeCreatorService(workers={self.workers})"
```

Figure 4.76 Code for the Parallel Ingredient Runner

```python
class CortadoCreator(CoffeeCreator):
    def __init__(self, parallel_coffee_creator_service : TypeCoffeeCreatorService):
        self.parallel_coffee_creator_service : TypeCoffeeCreatorService = parallel_coffee_creator_service

    @initialize_drink(*drinks_pin_mapper[CoffeeTypes.CORTADO.value], type_input=GPIO.OUT)
    def do(self, cortado_data : Dict[str, Any], *, pins : List[int] | None = None) -> Tuple[str]:
        params : Dict[str, Any] = {
            "drink_data" : cortado_data,
            "pins" : pins,
            "parallel_coffee_creator_service" : self.parallel_coffee_creator_service,
            "attrs_path" : CORTADO_ATTRS_PATH
        }
        return CoffeeCreatorSimpleFacade.create(params=params)
```

Figure 4.77 Code For Creating a Cortado Drink

Because the user can choose a Langchain Agent using the RAG method to create a coffee recipe for them without manually selecting it, I generate all drink recipes at runtime. This approach leverages the probabilistic nature of the LLM, as its responses are not deterministic. Even with the temperature value set to 0, intended to minimize the creativity of the LLM responses, I sometimes receive responses that do not conform to the desired drink format.

To address this, I use Python's *getattr* function to dynamically create methods at runtime. To add an extra layer of protection against potential system failures, I implement a *hasattr* check to verify if a function with the given name exists. You could go back to check the code on Figure 4.76, where we add only the functions with valid names as separate workers, ignoring the rest of the recipes that do not match those.

The CoffeeCreatorSimpleFacade encapsulates most of the logic for implementing the coffee creation system, as illustrated in Figure 4.78.

```python
class CoffeeCreatorSimpleFacade:
    @staticmethod
    def create(params : Dict[str, Any]) -> Tuple[str, Dict[str, Any]]:
        GPIO.output(LED_INDICATOR, GPIO.HIGH)

        drink_data : Dict[str, Any] = params.get("drink_data", None)
        pins : List[int] = params.get("pins", None)
        parallel_coffee_creator_service : TypeCoffeeCreatorService = params.get("parallel_coffee_creator_service", None)
        attrs_path : str = params.get("attrs_path", None)

        drink_size : str = drink_data.get("coffeeCupSize", "S")
        if drink_size in DRINK_SIZES:
            attrs_path = os.path.join(attrs_path, drink_size)
        else:
            raise NoSuchDrinkSizeError()

        if drink_data is None or pins is None or parallel_coffee_creator_service is None or attrs_path is None:
            raise NoAllParamsPassedError()

        start_time : float = time()
        message : str | None = None
        history : Dict[str, Any] = {}
        verbose : bool = drink_data.get("verbose", False)

        module : ModuleType = get_module_from_path(module_path=attrs_path)

        if parallel_coffee_creator_service == CoffeeCreatorType.PARALLEL.value:
            parallel_coffee_creator_service.start_workers(data=drink_data, module=module, pins=pins)
            if len(parallel_coffee_creator_service) > 0:
                parallel_coffee_creator_service.wait_for_workers_to_finish()
        elif parallel_coffee_creator_service == CoffeeCreatorType.SEQUENTIAL.value:
            pass
        else:
            message = NoSuchCoffeeTypeCreatorService().message

        if message is None:
            drink_name: str = drink_data.get("coffeeName", "Coffee drink")
            customer_name: str = drink_data.get("customerName", "Anonymous").capitalize()
            drink_quantity: str = drink_data.get("quantity", "1")

            message : str = f"Customer {customer_name} successfully recieved {drink_quantity} {drink_name if drink_quantity == 1 else f'{drink_name}s'}"
            history : Dict[str, Any] = {"time_for_order" : time() - start_time}

        GPIO.output(LED_INDICATOR, GPIO.LOW)
        return (message, history) if verbose else (message, {})
```

Figure 4.78 The CoffeeCreatorSimpleFacade Class

# 5. APPLICATION TESTING

For the testing phase of the on-premise backend, I primarily used the Pytest framework in Python to develop unit tests. Figure 4.79 presents a sample code test from the coffee_recommender microservice, illustrating how the model's accuracy is validated to ensure it surpasses a threshold of 0.8.

```python

def test_model_accuracy() -> None:
    path : str = os.path.join(os.path.dirname(__file__), "coffee_dataset.txt")
    expected_accuracy : int = 0.8
    nr_samples : int = 700

    with open(path, "r") as input_file:
        lines : List[str] = input_file.readlines()
        lines.pop(0)
        lines : List[str] = lines[:-2]
        correct : int = 0
        for line in lines[:nr_samples]:
            answers : str = line.split(",")
            ground_truth : str = answers[-1].rsplit("\n")[0]
            answers.pop(-1)
            sample_dict : Dict[str, str] = {}
            for j, answer in enumerate(answers):
                sample_dict[f"Question {j}"] = answer
            prediction : int = predict(responses=sample_dict, k=1)
            predicted : str = labels[prediction].lower()
            print(f"predicted_label : {predicted}, ground_truth_label : {ground_truth}")
            if ground_truth == predicted:
                correct += 1
        actual_accuracy : float = ((correct / nr_samples)*100)

        assert actual_accuracy > expected_accuracy, f"Error: expected accuracy: {expected_accuracy}, actual accuracy: {actual_accuracy}"
```

```
PROBLEMS 3   TERMINAL   PORTS

TERMINAL

PS C:\Users\darie\Documents\faculty\MCCA\code\Cofster\microservices\coffee_recommender\classifier> pytest .\test_model_accuracy.py
============================= test session starts =============================
platform win32 -- Python 3.11.0, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Users\darie\Documents\faculty\MCCA\code\Cofster\microservices\coffee_recommender\classifier
plugins: anyio-3.6.2
collected 1 item

test_model_accuracy.py .

============================== 1 passed in 5.32s ==============================
```

Figure 4.79 Unit Test To Check If Model Accuracy Is Right

For the cloud-based code, I utilized manual testing, employing Postman as the primary tool for this. An example of this can be found in Figure 4.80, where I demonstrate how to insert a new entry (user) into the DynamoDB users_credentials table.

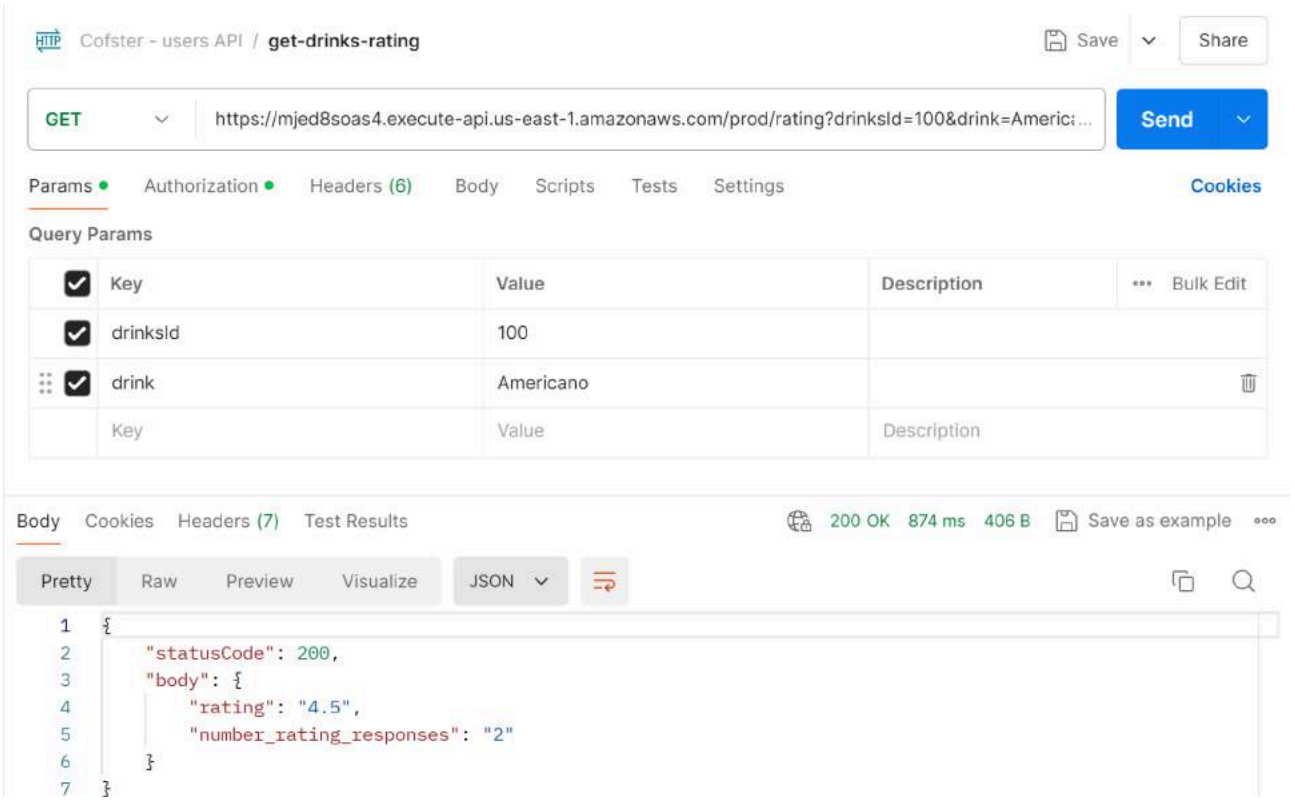


Figure 4.80 Manual Test To Return The Drink Rating Back

The frontend code is designed using the Dependency Inversion Principle, in order to enhance the testing experience. This approach not only facilitates the creation of unit tests, but it also seamlessly integrates mock tests as well, ensuring comprehensive and efficient testing [34].

For the Flutter frontend, I employed Mockito as the testing framework. The Mockito framework is widely used in modern applications for its ability to create mock objects, which simulate the behavior of real objects in a controlled way [35].

Figure 4.81 and Figure 4.82 show us some sample mock tests on the frontend side.

```
12    static Future<String> setSharedPreferenceValue(
13        {@required String key, @required dynamic value}) async {
14      _sharedPreferences[key] = value;
15      return await "Key: ${key}, value: ${value}";
16    }
17
18    static dynamic getSharedPreferenceValue(String key) {
19      return _sharedPreferences[key];
20    }
21  }
22
23  void main() {
24    testWidgets(
25      "Test setTimer method",
26      (WidgetTester tester) async {
27        final String sharedPreferenceKey = "<elapsedTime>";
28        MockSharedPreferences.setSharedPreferenceValue(
29          key: sharedPreferenceKey,
30          value: null,
31        );
32        String futureDateAndTime =
33            MockSharedPreferences.getSharedPreferenceValue(sharedPreferenceKey);
34        expect(futureDateAndTime, equals(null));
35
36        final DialogFormularTimerSingletonProvider dialogFormularTimerProvider =
37            DialogFormularTimerSingletonProvider.getInstance(
38          sharedPreferenceKey: sharedPreferenceKey,
39          futureDateAndTime: futureDateAndTime != null
40              ? DateTime.tryParse(futureDateAndTime)
41              : null,
42          onSetSharedPreferenceValue: (String key, {@required dynamic value}) =>
43              MockSharedPreferences.setSharedPreferenceValue(
44            key: sharedPreferenceKey,
45            value: null,
46          ),
47          onGetSharedPreferenceValue: (String key) =>
48              MockSharedPreferences.getSharedPreferenceValue(sharedPreferenceKey),
49          debug: true,
50        );
51
52        expect(dialogFormularTimerProvider.displayDialog, isFalse);
```

Figure 4.81 Mock Test To Start Dialog Questionnaire

mobile_app_frontend > lib > tests > units > firebaseOrderInformationDao.dart > ...

```dart
import 'package:flutter_test/flutter_test.dart';
import 'package:mockito/mockito.dart';
import 'package:coffee_orderer/data_access/FirebaseOrderInformationDao.dart'
    show FirebaseOrderInformationDao;
import 'package:firebase_core/firebase_core.dart';
import 'package:firebase_core_platform_interface/firebase_core_platform_interface.dart'
import 'package:flutter/services.dart';
import 'package:coffee_orderer/models/orderInformation.dart'
    show OrderInformation;
import 'package:coffee_orderer/utils/constants.dart' show ORDERS_TABLE;

typedef Callback = void Function(MethodCall call);

void setupFirebaseAuthMocks([Callback customHandlers]) {
  TestWidgetsFlutterBinding.ensureInitialized();

  setupFirebaseCoreMocks();
}

Future<T> neverEndingFuture<T>() async {
  while (true) {
    await Future.delayed(const Duration(minutes: 5));
  }
}

class MockFirebaseApp extends Mock implements FirebaseApp {}

Run | Debug | Profile
void main() {
  setupFirebaseAuthMocks();

  setUpAll(() async {
    await Firebase.initializeApp();
  });

  testWidgets('Test FirebaseOrderInformationDao', (WidgetTester tester) async {
    List<OrderInformation> ordersList =
        await FirebaseOrderInformationDao.getAllOrdersInformation(ORDERS_TABLE);
    expect(ordersList, isA<List<Map<String, dynamic>>>());
  });
}
```

Figure 4.82 Mock Test To Fetch All Orders From Firebase

# 6. CONCLUSIONS AND FUTURE WORK

Therefore, Cofster stands as a baseline to the remarkable potential of integrating advanced AI technologies into the coffee-making process. By enhancing user experience and operational efficiency, it sets a new standard for automation in this domain. The strategic adoption of a microservice architecture, which seamlessly combines on-premise servers with cloud infrastructure, underscores the forward-thinking approach of this project. Furthermore, the deployment of state-of-the-art models, such as YOLOv8 and YOLOv9, highlights the technical sophistication of this implementation.

The system's cutting-edge features, including an autoregressive coffee prompt updater, powered by a LangChain agent utilizing the RAG method, and real-time data streaming to clients via Kafka, signify a paradigm shift in the coffee creation industry. These advancements not only improve the automation process but also introduce a level of personalization and efficiency previously unseen.

Future work will concentrate on refining the personalization algorithms, both for the autoregressive coffee prompt updater and the coffee recommender system. This continuous improvement aims to elevate the autonomous coffee-making process to even greater heights. This research not only showcases the current capabilities of AI and automation but also paves the way for more intelligent and efficient solutions across various domains, heralding a new era of technological innovation.

# REFERENCES

[1] L. D. Tyson and J. Zysman, “Automation, AI & work,” Daedalus, vol. 151, no. 2, pp. 256–271, 2022, https://www.amacad.org/publication/automation-ai-work

[2] R. Rathore, “The application of the robot for the coffee manufacturer,” Asian Journal of Research in Social Sciences and Humanities, vol. 11, no. 12, pp. 148–152, 2021, https://www.indianjournals.com/ijor.aspx?target=ijor:ajrssh&volume=11&issue=12&article=028

[3] D. Kim, S. Lee, N.-M. Sung, and C. Choe, “Real-time object detection using a domain-based transfer learning method for resource-constrained edge devices,” in 2023 International Conference on Artificial Intelligence in Information and Communication (ICAIIC), pp.457–462, 2023, https://ieeexplore.ieee.org/abstract/document/10067064

[4] S. Swain, A. Deepak, A. K. Pradhan, S. K. Urma, S. P. Jena, and S. Chakravarty, “Real-Time Dog Detection and Alert System using Tensorflow Lite Embedded on Edge Device,” in 2022 1st IEEE International Conference on Industrial Electronics: Developments & Applications (ICIDeA), pp. 238–241, 2022, https://ieeexplore.ieee.org/abstract/document/9969906

[5] A. Parmar, R. Gaiiar, and N. Gajjar, “Drone based Potholes detection using Machine Learning on various Edge AI devices in Real-Time,” in 2023 IEEE International Symposium on Smart Electronic Systems (iSES), pp. 22–26, 2023, https://ieeexplore.ieee.org/abstract/document/10467087

[6] C. Nigam, G. Kirubasri, S. Jayachitra, A. Aeron, and D. Suganthi, “Real-Time Object Detection on Edge Devices Using Mobile Neural Networks,” in 2024 International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics (IITCEE), pp. 1–4, 2024, https://ieeexplore.ieee.org/abstract/document/10467220

[7] V. A. M. Luis, M. V. T. Quinones, and A. N. Yumang, “Classification of Defects in Robusta Green Coffee Beans Using YOLO,” in 2022 IEEE International Conference on Artificial Intelligence in Engineering and Technology (IICAIET), pp. 1–6, 2022, https://ieeexplore.ieee.org/abstract/document/9936831.

[8] K. Che, Y. Liang, Y. Zeng, T. Li, X. Zhu, and W. Lv, “Revolutionizing Traditional Chinese Medicine Image Classification and Recognition with an Improved YOLOv5,” in 2023 2nd International Conference on Health Big Data and Intelligent Healthcare (ICHIH), pp. 1–5,
2023, https://ieeexplore.ieee.org/abstract/document/10396627

[9] P. Athira, T. P. M. Haridas, and M. H. Supriya, “Underwater object detection model based on YOLOv3 architecture using deep neural networks,” in 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), vol. 1, pp. 40–45, 2021, https://ieeexplore.ieee.org/abstract/document/9441905

[10] K. Tseng, “Predicting victories in video games: using single XGBoost with feature engineering: IEEE BigData 2021 Cup-Predicting Victories in Video Games,” in 2021 IEEE International Conference on Big Data (Big Data), pp. 5679–5682, 2021, https://ieeexplore.ieee.org/abstract/document/9671454

[11] K. Tseng, “Predicting victories in video games: using single XGBoost with feature engineering: IEEE BigData 2021 Cup-Predicting Victories in Video Games,” in 2021 IEEE International Conference on Big Data (Big Data), pp. 5679–5682, 2021, https://ieeexplore.ieee.org/abstract/document/9671454

[12] S. Melenli and A. Topkaya, “Real-time maintaining of social distance in covid-19 environment using image processing and big data,” in Trends in Data Engineering Methods for Intelligent Systems: Proceedings of the International Conference on Artificial Intelligence and Applied Mathematics in Engineering (ICAIAME 2020), pp. 578–589, 2021, https://link.springer.com/chapter/10.1007/978-3-030-79357-9_55

[13] Stripe Payment Service: https://stripe.com/

[14] Raspberry Pi Image Processing Programming: https://link.springer.com/book/10.1007/978-1-4842-2731-2

[15] Image Raspberry PI: https://www.raspberrypi.com/products/raspberry-pi-5/

[16] Raspberry PI 5 vs Raspberry PI 4 - https://raspberrytips.com/raspberry-pi-5-vs-pi-4/

[17] What Is Docker: https://sematext.com/glossary/docker/

[18] Domain Driven Design - https://medium.com/@karahanozen/all-you-need-to-know-about-domain-driven-design-9d06c5234990

[19] Y Prajwal, Jainil Viren Parekh, and Rajashree Shettar. A brief review of micro-frontends. United International Journal for Research and Technology, 2(8), 2021, https://uijrt.com/articles/v2/i8/UIJRTV2I80017.pdf

[20] Ikun Han, Chunjiang Liu, and Pengfei Wang. A comprehensive survey on vector database: Storage and retrieval technique, challenge. arXiv preprint arXiv:2310.11703, 2023, https://arxiv.org/abs/2310.11703

[21] Shish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing system, 30, 2017. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[22] Image Standard Scaler: https://towardsai.net/p/l/which-feature-scaling-technique-to-use-standardization-vs-normalization

[23] Image 3D Tensor Shape: https://medium.com/@raman.shinde15/understanding-sequential-timeseries-data-for-lstm-4da78021ecd7

[24] Image Batch Normalization: https://medium.com/nerd-for-tech/batch-normalization-51e32053f20

[25] Image ReLU activation function: https://www.v7labs.com/blog/neural-networks-activation-functions -

[26] Image Dropout: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

[27] Image Softmax Formula: https://developer.apple.com/documentation/accelerate/bnnsactivationfunction/2915301-softmax

[28] Image Softmax Explanation: https://www.researchgate.net/figure/The-Softmax-function-and-an-example-of-how-the-function-transforms-the-weights-from-the_fig4_371120098

[29] Image Cross Entropy Formula: https://chris-said.io/2020/12/26/two-things-that-confused-me-about-cross-entropy/ - cross

[30] Image Stochastic Gradient Descent: https://www.geeksforgeeks.org/difference-between-batch-gradient-descent-and-stochastic-gradient-descent/

[31] Image Retrieval Augmented Generation: https://blog.langchain.dev/retrieval/

[32] Wen-Sheng Wu and Zhe-Ming Lu. A real-time cup-detection method based on yolov3 for inventory management. Sensors, 22(18):6956, 2022, https://www.mdpi.com/1424-8220/22/18/6956

[33] Xueqiu Wang, Huanbing Gao, Zemeng Jia, and Zijian Li. Bl-yolov8: An improved road defect detection model based on yolov8. Sensors, 23(20):8361, 2023, https://www.mdpi.com/1424-8220/23/20/8361

[34] The Mockito Framework: https://site.mockito.org/

[35] What is Mock Testing: https://www.radview.com/glossary/what-is-mock-testing/