University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

Final Report

Kitchen Buddies

Group 2023.50

Shadi Zargari	sszargar
Maya Bishop	mvpbisho
Aurchon Datta	a25datta
Caleb Chacko	ctchacko
Marley Liu	m289liu

Consultant: Prof. Werner Dietl

March 22, 2023

Abstract

It has been proven that cooking alongside friends or family has many psychological benefits such as healthy social bonding time, feeling connected to cultural backgrounds, and being an act of self-care. However, cooking is usually done individually and recipes are made one at a time. The objective of our project is to optimize cooking processes and encourage positive group cooking experiences. For our system's input, the user will submit several recipes to cook at once and our program will parse the steps using natural language processing. These steps are organized using adaptive algorithms and scheduling procedures to create a dynamic workflow that ensures all dishes are warm at the same time. The user can also submit constraints such as the number of users who are cooking, their relative skill level, and kitchen equipment available to be considered in our optimization process. Subsequently, the steps are sent to each user's device through networking protocols. Our app prioritizes a hands-on collaborative experience, unlike alternatives that encourage socializing by simply sharing recipes between friends. Another advantage of our design is that we allow the user to input any recipe online or manually typed rather than pulling it from a preset database.

Acknowledgements

Associate Professor Werner Dietl played an instrumental part in the success of this project, and we would like to thank him for his support and feedback throughout the process of this project.

Professor Lukasz Golab provided valuable insights into our natural language processing (NLP) strategy throughout this project. Without his advice, we would have lost many hours struggling to build an NLP model. We thank him for his time and advice in this matter.

Table of Contents

List of Figures

List of Tables

1. High-level Description

1.1 Motivation

Nowadays, cooking is done individually and recipes are made one at a time. When eating a meal with family or friends, it is usually done by going to a restaurant or ordering delivery. Cooking for large gatherings is often a healthier and cheaper alternative, but organizing multiple dishes can be time consuming. In addition, cooking alongside other people can be disorderly as it is difficult to keep track of multiple recipes being completed in real-time. However, it has been proven that cooking alongside friends or family has many psychological benefits such as healthy social bonding time, feeling connected to cultural backgrounds, and being an act of self-care [1]. Cooking being done with multiple people (as opposed to be being done individually) could be more common if there was a way to organize and coordinate the completion of multiple recipes efficiently, such that multiple users can easily follow.

1.2 Project Objective

The objective of our project is to design a service that creates a cooking workflow where multiple people are following multiple recipes simultaneously. This will in turn create coordination and minimize cooking time by up to 15% as compared to doing each recipe sequentially. It will also encourage positive interactions while cooking. To measure these changes, it will validate the recipes we extract and balance the work each user is assigned.

1.3 Block Diagram

1.3.1 User Interface

In the user interface, users can configure the app to edit and save their user skill level and kitchen equipment (referred to as kitchen constraints in our app). Users can also link other users to their account as friends and upload information about multiple recipes. All of this information is then utilized when creating the optimized set of meal steps, which is sent to all users who are in the meal session. For a sample workflow, please see Appendix A.

1.3.2 Input Processing

This subsystem is the first step done after the user inputs their desired recipes for the meal session. This system translates recipes which are either from URLs or are manually typed into a standardized format. It then extracts required information and outputs it in a JSON file so that the system can start processing it as usable information.

1.3.3 Recipe Processing

The goal of recipe processing is to convert each individual recipe into certain data structures we have defined. In addition, the algorithm must identify which steps are dependent on each other, which steps must be completed in chronological order, and which steps can be done simultaneously.

For example, preheating the oven, chopping onions, and boiling water are not dependent on each other. However, you need to have the onions chopped before cooking them.

1.3.4 Optimization Algorithm

The focus of the optimization algorithm is to combine recipes and optimize a workflow to send to users. It takes in the data structure outputted by recipe processing and the user’s profile in the database which includes their skill level and kitchen constraints. With this knowledge, it creates a workflow that parallelizes tasks, reduces total required cooking time, and ensures all recipes finish around the same time.

1.3.5 Server-Side Utilities

These utilities include the cloud server for running the backend of our app and the three databases we used, namely MySQL (for user and recipe metadata), Firebase Cloud Firestore (for user device tokens and push notifications), and Neo4j (for recipe data, i.e. steps and ingredients).

1.3.6 Network Communication

We use APIs to communicate information between the client (mobile device) and the server. Once the server-side finishes creating the parallelized recipe task list, the tasks are transmitted to the client using an API.

Additionally, we have client-client communication in the frontend. When a user is blocked on a task due to others, they can notify other users of the required supplies and ingredients they need to progress. This allows for easier coordination between more complex tasks.

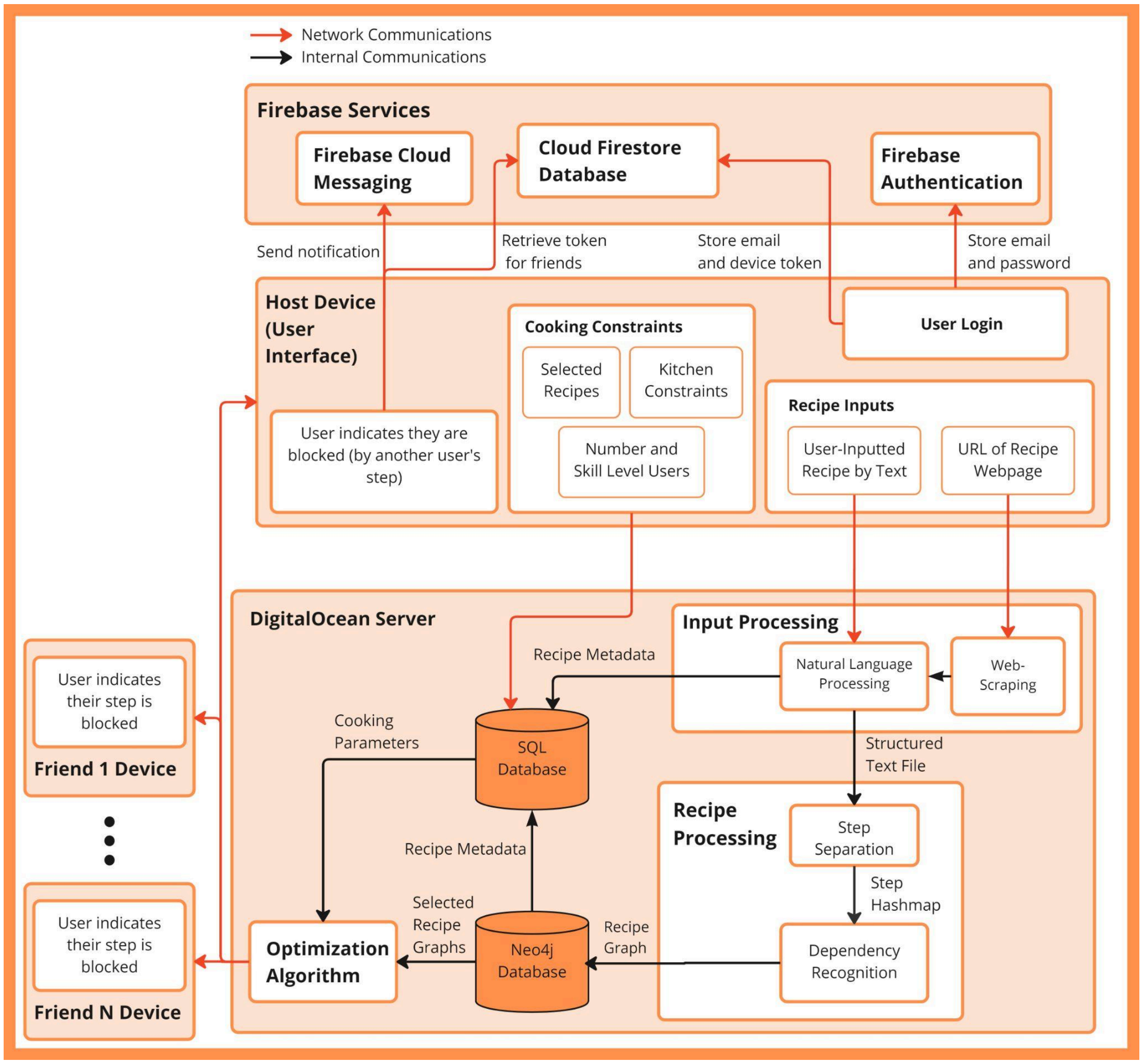


Figure 1: Block Diagram of Kitchen Buddies

2. Project Specifications

2.1 Functional Specifications

Table 1: Functional Specifications for Kitchen Buddies

ID	Specification	Necessity	Description
FS1	Output: Recipe Processing: Optimized	Essential	Kitchen Buddies will output an optimized set of steps based on the constraints that

	scheduling of steps		the user has inputted
FS2	Output: Communication between friends on different devices	Essential	Kitchen Buddies must send a notification to other users while the next step in their recipe is currently blocked by one of their Buddies
FS3	Input Processing: Allows multiple Buddies to cook the recipe(s) at once	Essential	Kitchen Buddies will parallelize the recipe(s) such that up to 3 Buddies are able to cook all at once
FS4	Input Processing: Allows user(s) to cook multiple recipes at once	Essential	Kitchen Buddies will parallelize up to 3 different recipes such that each user has a subset of steps
FS5	Input Processing: Allow user to add constraints based on kitchen equipment	Essential	Kitchen Buddies will allow users to input their access to kitchen equipment (i.e. number of cutting boards, pots, etc.), which recipe processing will account for
FS6	Input Processing: Allow user to add constraints based on skill level	Essential	Kitchen Buddies will allow users to input skill levels of beginner, intermediate and advanced, which recipe processing will utilize when creating each set of steps
FS7	Input Processing: Text recognition from an online recipe	Non-Essential	Kitchen Buddies must have 90% accuracy with webscraping recipes from a URL to add them to the user's personal cookbook
FS8	Input Processing: Text recognition from a set of steps in app	Essential	Kitchen Buddies must be able to accept a set of recipe instructions, manually typed on the user's phone, into their cookbook
FS9	Input Processing: Authentication	Non-Essential	Kitchen Buddies will authenticate new users to add security by making sure emails are unique and have passwords

2.2 Non-Functional Specifications

Table 2: Non-functional Specifications for Kitchen Buddies

ID	Specification	Necessity	Description
NSF1	Cookbook database size	Essential	Kitchen Buddies must be able to support a minimum of 100 recipes in the user's personal cookbook database
NSF2	Application size	Essential	Kitchen Buddies must not exceed 1 GB in size

NSF3	Supportability	Essential	Kitchen Buddies must be supported on Android 10.0.0 and higher to give access to most of the Android userbase
NSF4	Backend server uptime	Essential	Kitchen Buddies' backend services must have an uptime of 99%
NSF5	Battery consumption	Non-Essential	Kitchen Buddies must use at most 10% battery, as measured by the Android usage metrics, per hour use of the app

3. Detailed Design

3.1 User Interface

With users requiring easy access in the kitchen and simple communication with other devices, we decided to implement our product as a mobile application. We considered creating a web application, however, this seemed a bit cumbersome given the end goal of our project.

The user interface of our application must be simple, intuitive, and responsive. To meet these requirements, we assessed many different frameworks including React, Objective C, Kotlin and Flutter. After evaluating these choices, we initially decided to use Kotlin as it is cross-platform, familiar to our team, and simple to design. We used Kotlin for the first prototype of the app that was presented to our team's consultant. We have since pivoted over to using Flutter because of its fast development cycles, hot reloading, and widgets that provide a customizable and flexible user interface.

The user's experience with the app begins with a signup/login page that authenticates new users based on a combination of their username, email and password, allowing us to fulfill FS9.

To focus on simplicity, we made three tabs for app navigation which the user sees after they login to the app.

1. Home Tab: This page is a central location for adding and editing information about the user's skill level, their kitchen constraints, and their friends who they have added in the app. This tab allows us to fulfill the functional specification FS5 and FS6.
2. Cookbook Tab: This page allows users to input new recipes and view those that they have already added to the app.
3. New Meal Session: This page is where users can start a new meal session, taking into account the recipes, number of cooks, user skill level and additional kitchen equipment. Each user is sent their optimized set of recipe steps, allowing them to all cook at once and fulfilling functional specification FS3.

3.2 Input Processing

There are 2 types of recipe formats that the user can input: manual input from the application and web URLs. By allowing these 2 input types, we fulfill specifications FS7 and FS8. The the result of input processing is a text file that is structured in the following way: name, total cook time, ingredients, and instructions.

```
Recipe Name
Total Time: 35 (in minutes)

Ingredients:
8 cups ingredient 1
1/2 pound ingredient 2
2 tablespoons ingredient 3

Instructions:
1. Cut ingredient 1
2. Mix together ingredient 2 and 3
3. Fry ingredient 1, wait 30 seconds and then add mixture from Step 2
```

Figure 2: Resulting output from any input processing type

3.2.1 Manual Entry

With manual recipe entry, users can type up their recipe from scratch within the app. This data can then be quickly and easily converted into a text file to be sent to recipe processing.

3.2.2 Webscraping

When a recipe URL is inputted, it undergoes a webscraping process that outputs the recipe data in the form of a text file. We are using the Requests Python library to extract the information from the website. We decided to use Requests as it extracts the HTML from the page using a light-weight program, unlike Selenium. The Requests package checks if the website has any recipe metadata. Since many recipe websites would like to be found on Google, they add easily accessible recipe metadata called a microformat [2]. This recipe data is public and identifies important information in the recipe (ingredients, time required, etc.). If the website contains the metadata, we can use a JSON package to extract it.

While the function of inputting a URL to add a recipe is not essential, we believe it was important to include for usability. Many people use links from online recipes and this would make our app more user friendly. Since it is not essential, we decided to throw an error if we received a URL that didn't have the microformat described above. This decision was made in order to focus more time on essential components.

3.2.3 Text Processing

After the recipe is processed by the webscraper into the specified format (Figure 2) or the user inputs the recipe directly, it undergoes the text processing algorithm which extracts the required information and outputs a JSON. Each node is a dictionary, where the key is the step number and the value is the object pertaining to each step of the recipe. Our initial goal was to build our own NLP model to process the recipe data, but we realized that this was extremely time consuming and beyond the scope of our knowledge. We decided to use an NLP library in Python called spaCy that is capable of parsing, part-of-speech (POS) tagging, and dependency recognition between words. The figure below shows the sentence visualizer feature in spaCy.

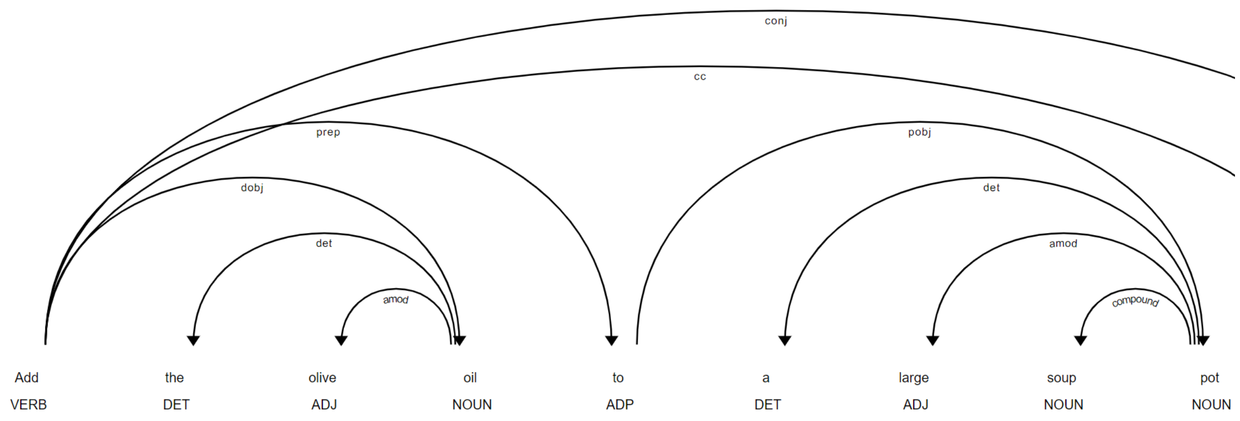


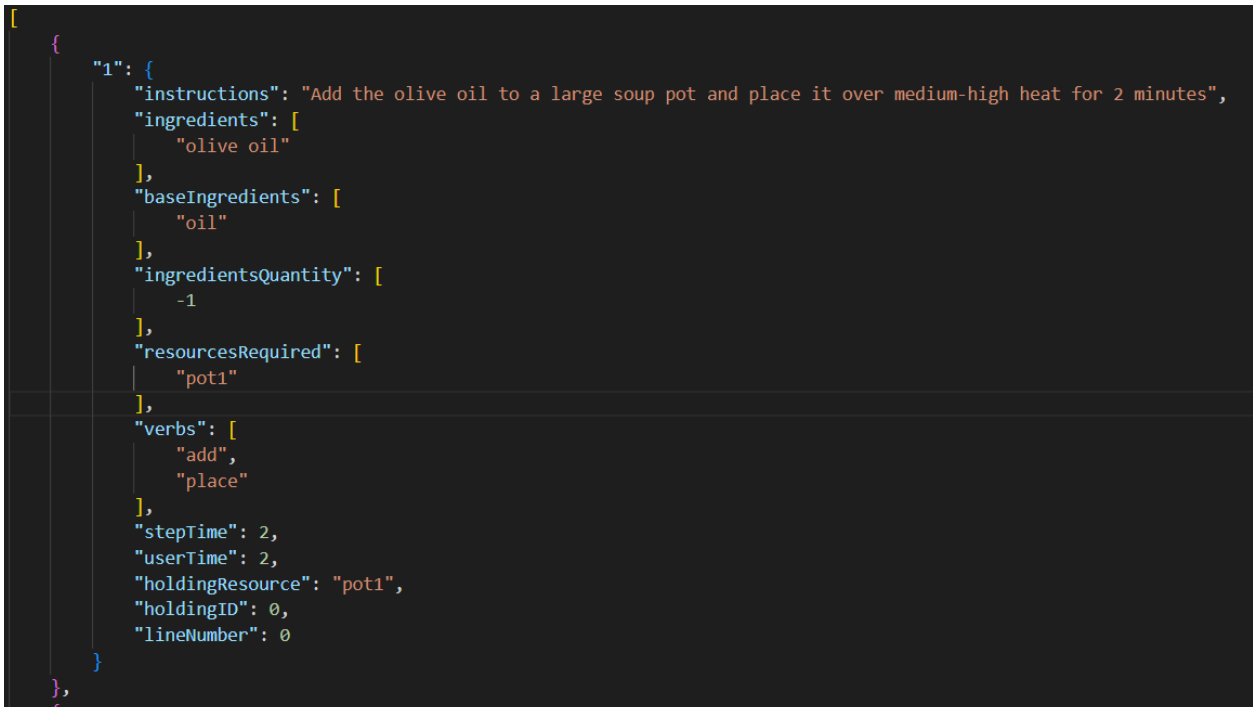
Figure 3: Sentence Visualizer in spaCy

The spaCy library is a tool that we use to extract the required information from each step of the recipe. It is important to note that there are countless ways that recipe steps can be structured, so our algorithm cannot be 100% accurate. A significant portion of our time was spent determining edge cases of our algorithm to increase processing accuracy across a variety of recipes. Figure 4 below shows the required attributes that need to be extracted from each step of the recipe.

In this algorithm, there is an outer loop that loops through each step of the recipe, and an inner loop that loops through each word of the step. The ingredients are found by identifying which words are tagged as nouns by spaCy, and then checking that the word is in the ingredient list from the recipe or an outsourced, exhaustive list of ingredients. If so, it is saved in the list of 'baseIngredients', which represent the base word of an ingredient. The full ingredient word is then extracted by checking dependencies to other words. For example, in the sentence visualizer above, the word 'olive' is linked to 'oil' as an adjective modifier. The ingredient quantity is found by checking any numerical modifiers to the base ingredient word. If there is no specified quantity, a value of -1 is used. The 'resourcesRequired' attribute represents which kitchen supplies are required in the current step (e.g. knife) and is found in a similar way to the list of ingredients.

The ‘stepTime’ represents the time required for each step, from start to completion. It is found by checking which nouns indicate units of time (‘seconds’, ‘minutes’, ‘hours’) and then checking any numerical modifiers. If a numerical modifier exists, it is set as the stepTime in minutes. If not, the verbs identified by spaCy are used to help calculate the required time for each user in the step. We hardcoded a dictionary which relates a verb to an approximate time related to that action. For example, we estimated that glazing an ingredient would take 3 minutes per minute, slicing something would take 3 minutes, etc. The ‘userTime’ attribute accounts for situations where a user is told to bake the pan for 20 minutes, but does not actually require the user to stand there for 20 minutes. In this case, the userTime would be set to 2 minutes.

The holding resource represents the object that holds the ingredients required in the step, such as a pot for soup or a bowl for a mix of fruits. The holding resource is found by looking for phrases such as ‘into a’, ‘onto’, ‘in a’ or anything that indicates placement of ingredients onto a resource. If a holding resource is not found in this manner, it is found through another algorithm outside of the inner loop that determines the holding resource by analyzing the ingredients and resources of the current step, as well as the holding resources of the previous steps.



```
[
    {
        "1": {
            "instructions": "Add the olive oil to a large soup pot and place it over medium-high heat for 2 minutes",
            "ingredients": [
                "olive oil"
            ],
            "baseIngredients": [
                "oil"
            ],
            "ingredientsQuantity": [
                -1
            ],
            "resourcesRequired": [
                "pot1"
            ],
            "verbs": [
                "add",
                "place"
            ],
            "stepTime": 2,
            "userTime": 2,
            "holdingResource": "pot1",
            "holdingID": 0,
            "lineNumber": 0
        }
    },
```

Figure 4: Example of JSON output after text processing

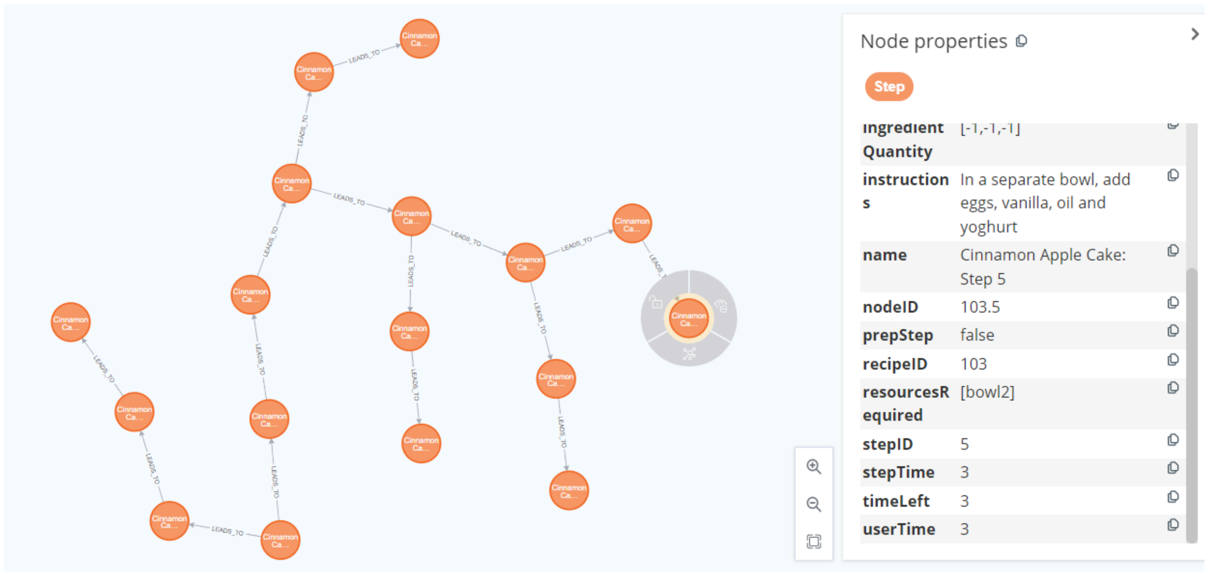
3.3 Recipe Processing

The goal of the scheduler is to take the individual nodes of the recipe coming from the output of the text processing (explained in section 3.2.3) and create connections between nodes which represent dependencies between different steps of the recipe. There are two types of dependencies: resource dependencies and time dependencies. Resource dependencies refer to steps that must be scheduled sequentially as they rely on the same ingredient or kitchen resource. Time dependencies are tasks where two steps must be scheduled a specific time apart. The output of this subsystem should be a recipe graph for each individual recipe inputted.

Our initial idea for recipe processing was to create a list of all the attributes that is ordered by which attribute we want to use to organize the graph first. Originally, we identified the most important attribute to be the ‘list of ingredients’, followed by the ‘holding resource’ and then the ‘list of resources required’. We later decided to create dependencies between ‘base ingredients’ instead of ‘list of ingredients’. The ‘base ingredient’ attribute of each node (step) (explained more in Section 3.2.3) provided a more accurate way of recognizing dependencies from ingredients between steps. For example, the recipe may involve melting chocolate chips, and step 3 includes ‘chocolate chips’ as an ingredient and step 7 includes ‘melted chocolate’ as an ingredient.

First, we loop through all nodes to gather all attributes and make a hashmap for every attribute that will cause dependencies. Once all the steps are converted to nodes and the text has been processed to find attributes, we loop through the dependencies hashmaps to create the edges in our recipe graphs. This ensures steps are only connected to the next step that is using the same resource or it has a strict time dependency on. By implementing this strategy, earlier steps in the chain are guaranteed to precede later steps with higher step numbers. We chose this method over the previous one as it involves a more structured process to create the recipe graphs. In addition, this allows us to create the graph in one pass rather than start a graph and rearrange many times when considering other dependencies.

Once the graph is created, it is saved into our Neo4j database which is explained further in Section 3.5.2.3. The figure below shows a Cinnamon Apple Cake recipe graph in our NEO4J database after the recipe processing algorithm, with some of the attributes of step 5 shown on the right. Note that the ‘prepStep’ attribute in the figure is no longer used.



ingredient Quantity	[-1,-1,-1]
instructions	In a separate bowl, add eggs, vanilla, oil and yoghurt
name	Cinnamon Apple Cake: Step 5
nodeID	103.5
prepStep	false
recipeID	103
resourcesRequired	[bowl2]
stepID	5
stepTime	3
timeLeft	3
userTime	3

Figure 5: Cinnamon Apple Cake graph in Neo4j database

3.4 Optimization Algorithm

3.4.1 Design Decisions

We started by analyzing the various goals of the algorithm to decide our design approach. First, the algorithm must take in the dependency graphs of multiple recipes. The result of the algorithm must have all the recipes “finished” at approximately the same time. This is critical for dishes that need to be hot or that have a large time constraint before they can be finished. Also, the algorithm needs to load-balance the steps between the requested number of people to ensure that everyone is involved and has roughly equal amounts of work. As a result, the algorithm must combine recipes and keep track of kitchen resources when assigning them to various users. Lastly, the algorithm needs to handle different skill levels (i.e., beginner, intermediate, and advanced). These skill levels are factored in by adding a time buffer to every step based on a user’s skill. For instance, a “beginner” user will have an additional 4 minutes added to their task, while an advanced user will receive no time buffer. At the end of the algorithm, we send the steps for each user to their devices. By creating this algorithm, we fulfill functional specification FS1 and FS4.

We had a choice to either view time as chronological (i.e., plan from the start of the recipes) or view time as reverse chronological order (i.e., we work backwards from the end result). We chose reverse chronological order so that we could ensure that recipes finished at the same time when possible.

In order to determine a method of assigning tasks to individual users, we considered what would lead to each user being kept busy continuously. We initially considered other task scheduling processes (such as assigning the longest critical path first), however, we found that taking a round robin inspired approach would result in each user being occupied with a task more often.

3.4.2 Optimization Implementation

Optimization begins by scheduling the final tasks of each recipe. The algorithm chooses which task to schedule next, based on the recipe with the largest remaining cook time using a priority queue. Afterwards, the algorithm checks each user’s current workload. Each user has a counter that represents the amount of time that they already have allotted to work on tasks. At the beginning, all counters are set to zero. The algorithm adds the current task to the user with the least amount of work. If all users have the same amount of work, the algorithm decides who to assign the task to based on other factors.

Before assigning the task to a user, the algorithm checks if the current task’s holding resource is available. This is done using a hashmap with the key being the type of resource and the value being a list of available resources (i.e. (key, value) -> (knife, [knife 1, knife 2])). If the resource is available, we store the earliest time the resource is free.

After finding the user and the earliest time the resources are free, the algorithm confirms that the user is free at the same time as the resources. If the user is free, we update the kitchen resources hashmap and the task is added to the user's task list. This list is a doubly linked list where the user tracks when it has assigned tasks and available gaps of time. These gaps are created due to time dependencies. For example, baking a chicken has a time-sensitive task dependency to heat the oven five minutes beforehand. If this type of task is given to a user, we prioritize scheduling the time-sensitive task. This ensures that the task is completed at the appropriate time.

To finish scheduling the current task, the user's counter is updated accordingly, the task is popped off from the priority queue and its resource dependencies are added to the queue. This process is repeated until all tasks are assigned to a user. After the optimization scheduling is completed, the user starts the tasks at the end of the linked list. This works because we start scheduling from the end of the recipes.

3.5 Server-Side Utilities

The server-side utilities are composed of two components. The first component is a cloud server (DigitalOcean) that runs our algorithm and holds our user database. The next component is made up of our three databases; the first is a relational database (MySQL) that holds user-specific data, the second is a relational document-based database (Cloud Firestore) which holds client-side information for push notifications and user authentication, and the third is a graph database (Neo4j) where each graph represents a recipe split into its different steps.

3.5.1 Cloud Server

When choosing a cloud server, there were two options that were the most relevant for Kitchen Buddies, namely DigitalOcean and AWS. AWS is more well-known and reputable within the industry. It has a robust ecosystem with high reliability, but its major drawback is a higher cost and a more complicated infrastructure to set up.

DigitalOcean was the better option as we do not need the extended feature set of AWS. It was much easier to set up and was cheaper, which made sense for us given the constraints on budget and time. We initially budgeted $6/month to host a DigitalOcean Droplet, which would give us access to a virtual machine to run our backend server on as well as 1GB memory and 25GB SSD. We thought that the 1GB memory would be enough to hold our relational database (as the other databases were hosted on third party cloud servers). However, during development of the project, we realized that our Java program was quite large and we hit 100% CPU usage when we first tried to run it on the server. Consequently, we resized our Droplet once at this time, and as we continued to add specifications to our project (e.g. running a Python script to parse recipes, improving the optimization algorithm), we needed to resize our Droplet again.

The final Droplet at the time of our demo cost $24/month, with 2 vCPUs, 4GB memory, and 60GB SSD. Although this cost was more than we predicted, we did not use our budget anywhere else in the project. A comparable AWS plan is priced at $24 USD/month ($27.5 CAD/month), which is still more expensive [3]**.**

3.5.2 Databases

3.5.2.1 MySQL Database

All metadata regarding both recipes and users are saved in the MySQL database. This includes profile info, friends, skill level, kitchen constraints, etc. We chose MySQL as our preferred management system due to familiarity. This database was stored server-side within DigitalOcean.

Figure 6 shows the tables we have in MySQL and the keys shared between them. RecipeID is a unique number that is auto-incremented whenever a new recipe is added to the AllRecipes table; it acts as the primary key. When users add recipes to their own profile, they are stored as a row in UserLinkedRecipes. RecipeID is therefore used as a foreign key in that table. UserEmail is a unique email used as the primary key for the UserInfo table, and used as a foreign key in most of the other tables. For instance, when adding a friend by email, we check that their email already exists in our UserInfo table. UserEmail is guaranteed to be unique because it is first passed to Firebase Authentication in the frontend, which checks that each account is generated with a new email.

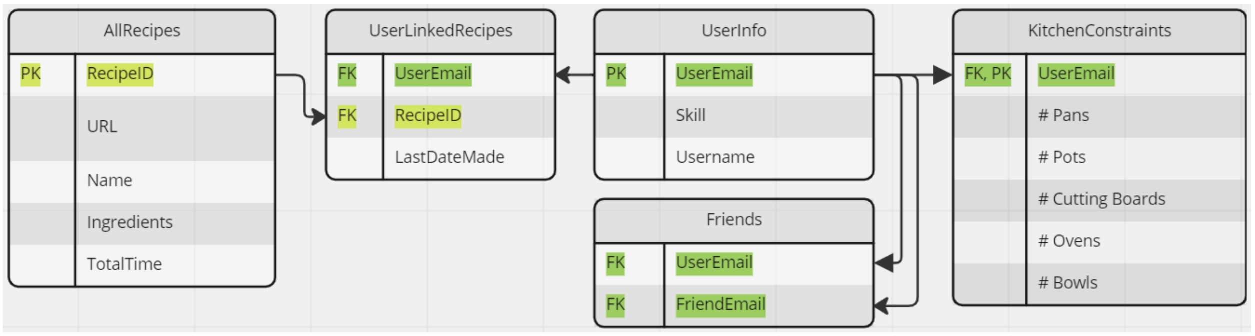


Figure 6: Table structure in MySQL database

3.5.2.2 Firebase Cloud Firestore Database

On our sign-in and sign-up pages, we use email and password verification provided by Firebase Authentication. There is a direct connection from the frontend to Firebase Cloud Firestore. We decided to use Firestore because it is easily integrated with other Firebase services.

Upon registration, the user email is stored in Firestore alongside the device token, which is a unique identifier for their phone. The user email is also propagated to the SQL database. It is practical to store the device tokens in Firestore because the backend will never need such information. Thus, we can reduce the number of API calls we need to propagate data to the server. When a host creates a meal session, we retrieve the device tokens of the host's friends so that the instructions will be sent to each of their phones. We look through the list of friends added to the meal session and query Firestore for their matching device tokens.

3.5.2.3 Neo4j Graph Database

When the user requests to start a meal session, they input the recipes they want to cook. If the recipes have been used in a previous session, they are already stored as a graph in our database. If not, a new graph is created for each recipe. We find each of the necessary graphs in the database and copy them to a new graph before beginning the optimization algorithm.

Neo4j is an open-source, NoSQL, native graph database that is offered as a managed service by AuraDB. Other graph databases include ArangoDB, which is known to have better performance than Neo4j [4]. However, Cypher for Neo4j is the most widely used graph query language, and there is much more support available online. This made Neo4j our preferred service for a graph database.

Nodes in the graph database can hold any number of attributes. They are connected by relationships, which have a direction, start node, and end node. We use the Neo4j-OGM library, an object graph mapper that abstracts the database and makes it convenient to persist a domain model in the graph, allowing us to query it without using low level drivers. We annotate Java objects with @NodeEntity to represent them as nodes in the graph, and use @RelationshipEntity to indicate the relationships between nodes. @Id is used to annotate a property and make it the primary identifier in our graph. Figure 14 shows how we use these 3 annotations in the Step class.

```java
@NodeEntity(label="Step") //this defines our step node and all its attributes
public class Step {
    @Id
    private Double nodeID;
    private Long recipeID; //number identifying recipe in database
    private Integer stepID; //step number from original recipe
    private String holdingResource; //what the ingredients are being kept in
    private Integer stepTime; //how much time it takes to complete step
    private Integer userTime; //time required for the user to be actively working
    private Integer timeLeft; //time remaining to complete original recipe
    private List<String> ingredientList; //names of ingredients
    private List<Float> ingredientQuantity; //quantities of ingredients
    private List<String> resourcesRequired; //kitchen equipment
    private String instructions;

    @Relationship(type = Connection.TYPE, direction = Relationship.OUTGOING)
    private Set<Connection> connections = new HashSet<>();

    [...] //constructors, getters, and setters go here
}
```

Figure 7: Java code snippet showing Step class

3.6 Network Communication

3.6.1 Client-Server Communication

Our app requires a form of web communication from multiple clients to a server. The application needs to be able to set up a meal session where the host can invite other users to cook with them. The server must be able to distribute steps to all users in the session.

Web communication can be handled by two distinct types of APIs: REST-based APIs or Web Socket-based APIs. Web sockets have lower overhead compared to HTTP polling, so messages are transmitted much faster compared to REST APIs [5]. However, the majority of our group members have more experience using HTTP. Therefore, we use REST APIs in this project for client-server communication.

3.6.1.1 REST APIs

There are 10 REST APIs that are used throughout the communication between the client and server. Each API and its use case is further explained in the table below. Note that all of the GET requests below utilize the user's email (as a unique identifier) to get data specific to their account.

Table 3: Description of REST APIs

API Name	**GET / POST**	**UI Page(s)**	**API Description**
AddSkillLevel	POST	Home	This API receives a user's email and their skill level. It adds this to the MySQL database table that contains user information.
GetUserSkill	GET	Home	The API retrieves the user's skill level from the MySQL database.
AddKitchenConstraints	POST	Home	The API receives the number of items that are in a user's kitchen as well as the user's email. The API saves this information in the MySQL database.
GetKitchenConstraints	GET	Home	The API retrieves the user's kitchen constraints from the MySQL database.
AddFriend	POST	Home	The API receives a user's email and the email of a new friend. It adds this information to the Friends table in the MySQL database.

GetFriendList	GET	Home / New Meal Session	The API retrieves a specified user's friends from the MySQL Friends table and returns the friends' emails.
RequestMealSessionSteps	POST	New Meal Session	The API takes in a user's email, kitchen constraints, the selected recipe IDs, and the selected friends for the meal session. It uses this data to run the optimization algorithm. It returns the steps for each user to the client once optimization is complete.
GetPastRecipes	Get	New Meal Session / Cookbook	The API retrieves the recipe information of all recipes tied to their account.
RequestRecipeByInput	POST	Cookbook	This API takes in a text input from the client. It sends the text file directly to input processing. After recipe processing is run and the recipe is saved to the Neo4j database.
RequestRecipeByURL	POST	Cookbook	This API takes in a URL from the client. It will use a webscraper to get all relevant recipe information. After recipe processing is run and the recipe is saved to the Neo4j database.

3.6.2 Client-Client Communication

FS2 describes the need to send a notification to other users for bottleneck steps, i.e. when a step needs to be completed by User 1 before User 2 can proceed with their next step. When we were designing our prototype, we researched several REST APIs to control notification overflow. We initially chose Twilio Notify because it only charges $0.00025 USD for each notification delivery attempt, which worked well for our budget. However, we found that the documentation was outdated and difficult to follow, so we switched to using Firebase services.

To create a push notification service for Android, we created a project on a platform called Firebase Console. We used the Firebase Cloud Messaging (FCM) service for push notifications. Figure 12 shows a new class made to send notifications through a HTTP post request. The POST URI is the FCM send datapoint, the data is an encoded JSON, and the key is our FCM credential. We use notifications of two types: one type sends a set of instructions from the meal session host user to a friend, and the other type is a notification alerting other users that the sender is blocked on one of their instructions. The first type is a very large JSON, so we do not want the receiver's phone to display the whole message. Hence, we differentiate between these two types of notifications by only copying the instructions into the notification body if it is a short blocking message. Reducing the size of the data JSON also reduces latency.

```dart
class NotificationProvider extends ChangeNotifier {

 sendNotification ({
   required String token, required String title, required String body, required bool
isBlocked}) async {
     const postUrl = 'https://fcm.googleapis.com/fcm/send';

     Map<String, dynamic> data;

     data = {
       "registration_ids": [token],
       "collapse_key": "type_a",
       "notification": {
         "title": title,
         "body": isBlocked ? body : "" , //isBlocked indicates it is a short message
       },
       'data': {
         "title": title,
         "body": body,
       },
     };

     final response =
         await http.post(Uri.parse(postUrl), body:json.encode(data), headers: {
           'content-type': 'application/json',
           'Authorization': FCM_KEY //FCM credential
         });
 }
}
```

Figure 8: Dart code snippet for a notification provider

When the instruction type of notification is received, the receiver is redirected to a new page that displays these instructions, as shown in Figure 9. If a blocking notification is received, we call a Flutter plugin for displaying local notifications, so the user can see it at the top of their screen.

```dart
FirebaseMessaging.onMessage.listen((event) {
 final splitMessage = (event.data.toString().split('title: '))[1]
     .split('}');

 //we categorize our notifications based on title
 if (splitMessage[0] != "Step Blocked") {
   if (mounted) {
     Navigator.of(context).push(MaterialPageRoute(  //push new page onto the stack
         builder: (context) => ReceivedInstructionScreen(message: event)));
   }
 }

 else {
   LocalNotificationService.init(); //Flutter plugin to show notification on phone
   LocalNotificationService.displayNotification(event);
 }
});
```

Figure 9: Dart code snippet that receives a notification

Figure 10 is a condensed code snippet from ReceivedInstructionsScreen showing how to send a blocking notification. We send a notification using fcmProvider (an instance of our notification provider from Figure 8) where the token is from our friends list and the body is a simple message about what step the user is blocked on.

```
final fcmProvider = Provider.of<NotificationProvider>(context);

fcmProvider.sendNotification(
   token: friendToken,
   title: "Step Blocked",
   body: "I'm blocked on Recipe " + id + " - Step " + number + "!",
   isBlocked: true);
```

Figure 10: Dart code snippet that sends a notification

4. Prototype Data

4.1 Login Screen

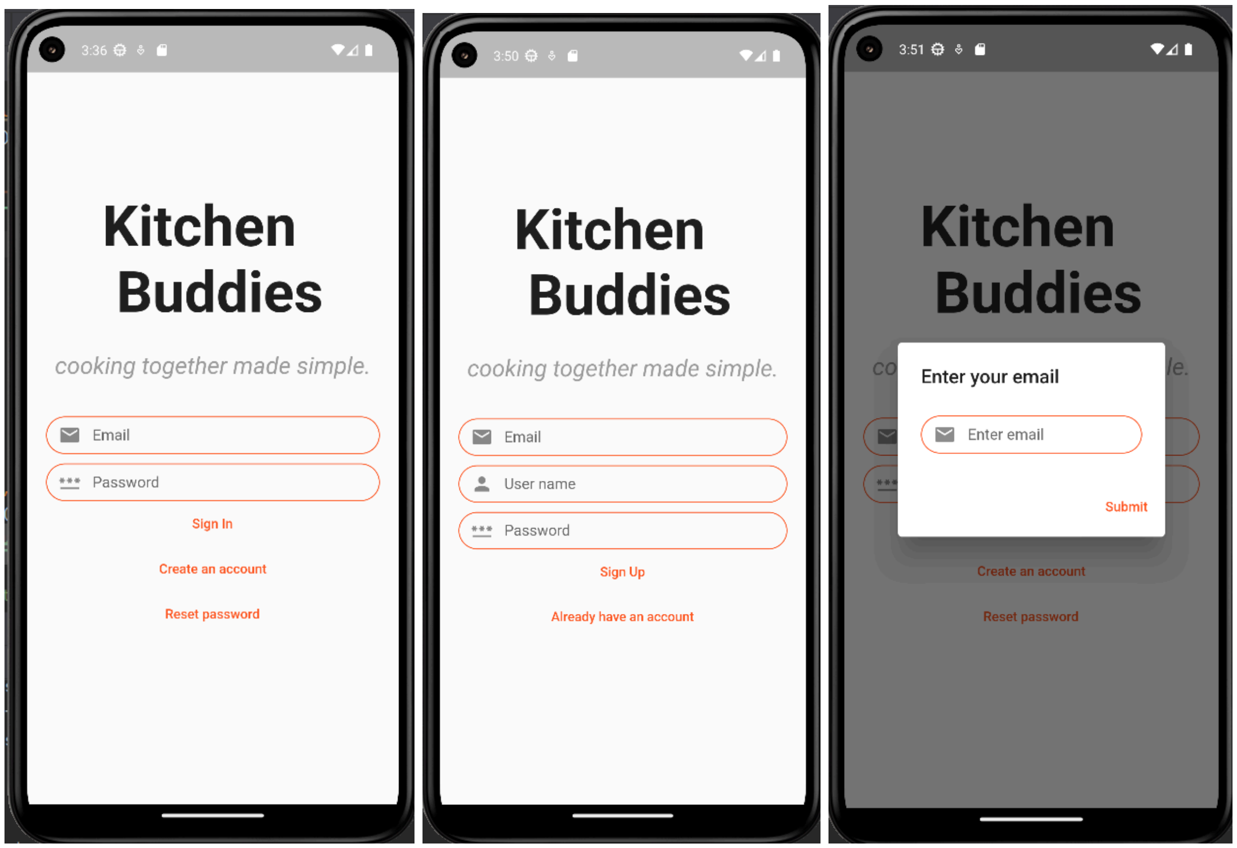


Figure 11: Screenshot of login and sign-up page

The user interface was successfully implemented to meet all the essential requirements described in Section 2. The user logs in to the device by entering their email and password credentials, as shown in Figure 11. A new account can also be created by clicking “Create an account”, upon which the user is prompted for a username alongside their desired email and password. If the user is registered with Kitchen Buddies and wants to change their password, they can click “Reset password”, where they are prompted for their email address. An email will be sent to that address with a one-time link to reset the password (See Appendix A Figure 17).

4.2 Home Page

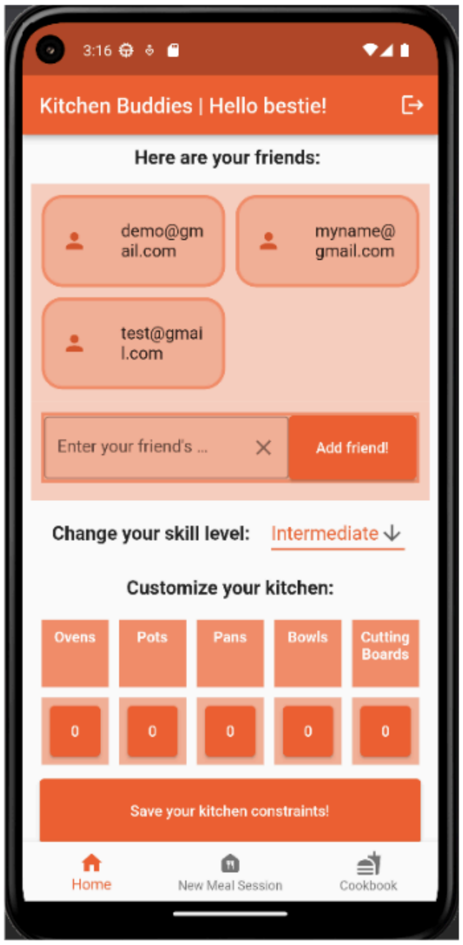


Figure 12: Screenshot of home page

4.2.3 Friends List

The user is able to add other users as friends from the home page. A text box is available, shown in Figure 12, to enter the friend’s email. After the user finishes typing, they click the “Add friend!” button. This prompts an API call to the backend to insert a new row in the FriendsList table in MySQL. Subsequently, there is another API call from the frontend to retrieve the user’s updated list of friends, and the first box will refresh to display a new tile with the friend’s email besides the other friends.

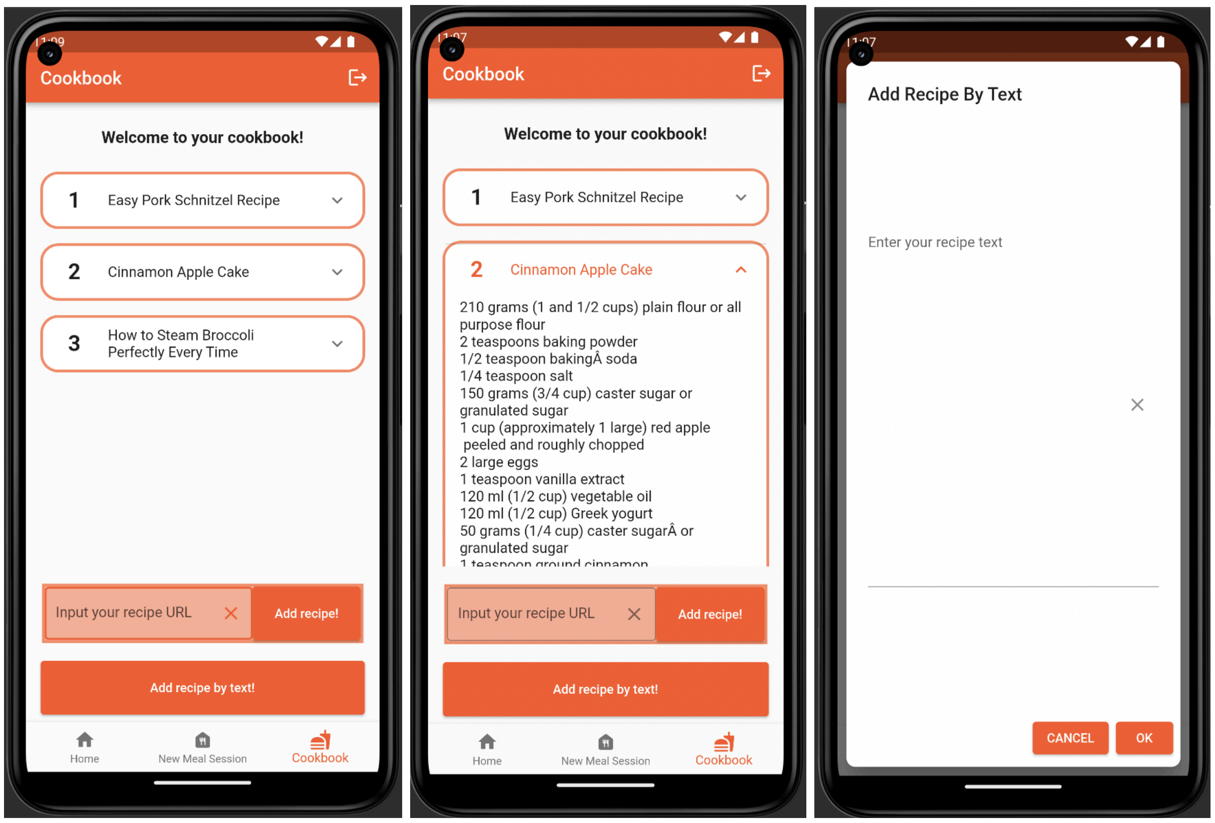
4.2.4 Skill Level

The user can select a skill level (beginner, intermediate, or advanced) from a dropdown menu. As described before, this prompts 2 API calls: one to update the user's skill level in the MySQL database, and a second to retrieve the new skill level to be displayed.

4.2.5 Kitchen Constraints

The user can enter the number of supplies they have in their kitchen (i.e. number of ovens, pots, pans, knives, bowls, and cutting boards) as constraints. These specific supplies were determined to be the most commonly used when developing the optimization algorithm. The method for entering the kitchen constraints is the same as described above. The user taps each number to overwrite it, and clicks "Save your kitchen constraints!" to make the update.

4.3 Cookbook Page



210 grams (1 and 1/2 cups) plain flour or all purpose flour
2 teaspoons baking powder
1/2 teaspoon bakingÂ soda
1/4 teaspoon salt
150 grams (3/4 cup) caster sugar or granulated sugar
1 cup (approximately 1 large) red apple peeled and roughly chopped
2 large eggs
1 teaspoon vanilla extract
120 ml (1/2 cup) vegetable oil
120 ml (1/2 cup) Greek yogurt
50 grams (1/4 cup) caster sugarÂ or granulated sugar

Figure 13: Screenshots of Cookbook page

4.3.1 Recipes List

The user is able to view all the recipes which are tied to their account in a list of dynamically expandable containers. When each recipe container is clicked on, it expands to show all of the recipe's ingredients and their respective quantities.

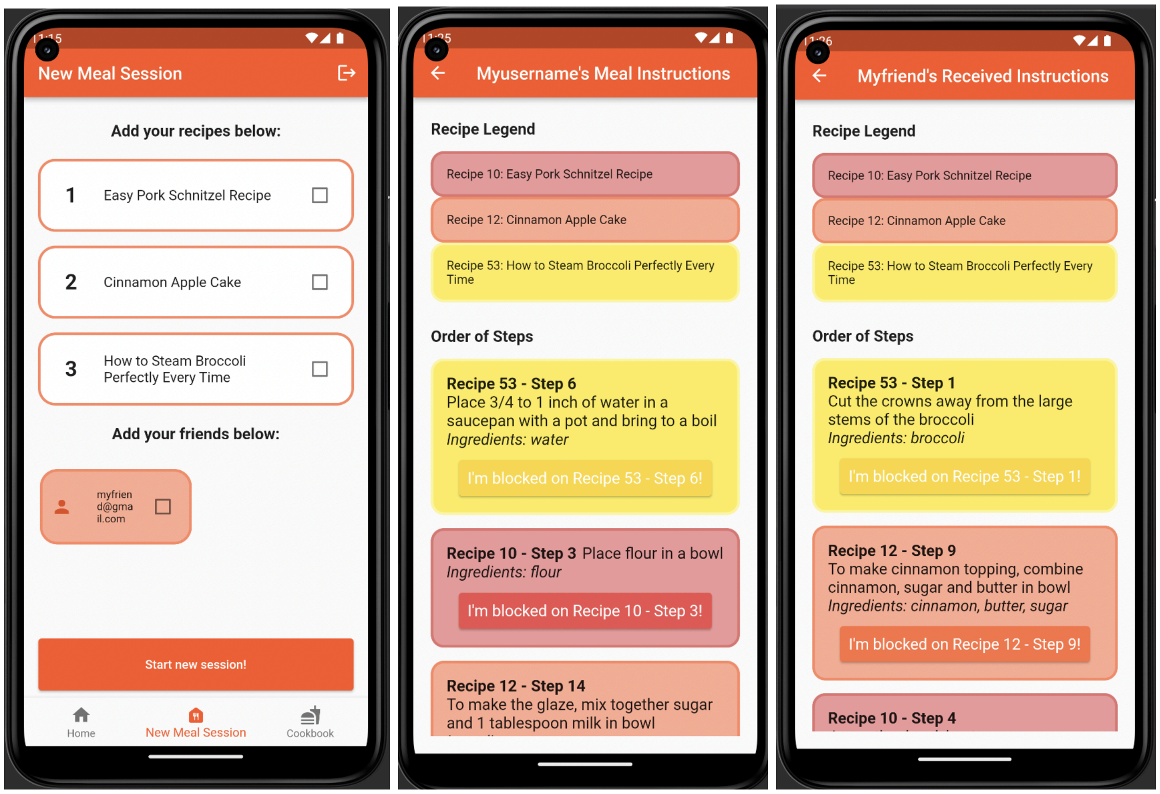
4.3.2 Add Recipe by URL

The user can input any URL that they find online for the app to parse through and grab all the necessary information.

4.3.3 Add Recipe by Text

The user can manually input a recipe for the app following a specified format for the app to parse through and grab all the necessary information.

4.4 New Meal Session


New Meal Session

Add your recipes below:

1 Easy Pork Schnitzel Recipe ☐

2 Cinnamon Apple Cake ☐

3 How to Steam Broccoli Perfectly Every Time ☐

Add your friends below:

myfriend@gmail.com ☐

Start new session!

Home | New Meal Session | Cookbook

Myusername's Meal Instructions

Recipe Legend

Recipe 10: Easy Pork Schnitzel Recipe

Recipe 12: Cinnamon Apple Cake

Recipe 53: How to Steam Broccoli Perfectly Every Time

Order of Steps

Recipe 53 - Step 6
Place 3/4 to 1 inch of water in a saucepan with a pot and bring to a boil
Ingredients: water
I'm blocked on Recipe 53 - Step 6!

Recipe 10 - Step 3 Place flour in a bowl
Ingredients: flour
I'm blocked on Recipe 10 - Step 3!

Recipe 12 - Step 14
To make the glaze, mix together sugar and 1 tablespoon milk in bowl

Myfriend's Received Instructions

Recipe Legend

Recipe 10: Easy Pork Schnitzel Recipe

Recipe 12: Cinnamon Apple Cake

Recipe 53: How to Steam Broccoli Perfectly Every Time

Order of Steps

Recipe 53 - Step 1
Cut the crowns away from the large stems of the broccoli
Ingredients: broccoli
I'm blocked on Recipe 53 - Step 1!

Recipe 12 - Step 9
To make cinnamon topping, combine cinnamon, sugar and butter in bowl
Ingredients: cinnamon, butter, sugar
I'm blocked on Recipe 12 - Step 9!

Recipe 10 - Step 4

Figure 14: Screenshots of (left to right) starting a New Meal Session as a user, the host user's recipe instructions, a friend's (received) recipe instructions

4.4.1 Start New Meal Session

The user selects a combination of the recipes in their cookbook that they want to make as well as the friends they want to include in the new meal session. The numbering of the recipes follows the numbering notation of recipes within the Cookbook page so that users can cross-reference more details about each recipe from the Cookbook page if necessary. This is shown in the leftmost screenshot in Figure 14 above.

After they finish selecting recipes and friends included in their new meal session, they can click the “Start new session!” button in the app which will run the optimization algorithm and send recipe steps to all included friends.

4.4.2 Meal Instructions Screen

After the “Start new session!” button is clicked by the user of the app (the host of the new meal session), the app sends back a list of meal instructions to the host and all friends who are included in the meal session. The steps are optimized based on the output of the algorithm, which takes into account the recipes, the number of users, their skill levels and the kitchen constraints. The top of each user’s screen has a colour-coded legend which indicates what colour corresponds with each recipe in their meal session. This is because we found that certain recipe titles were exceedingly long so we did not want them to be displayed within the “Order of Steps” section.

4.5 Backend Algorithms Quantitative Analysis

4.5.1 Webscraping Accuracy

Within our input processing subsystem, we included the option for users to add recipes they find online. For these recipes to work in our system, they must be accurately parsed into a format for our text processing component to read. As mentioned in Section 3.1.2.2, our prototype only works on URLs with microformat, which is included in the majority of cooking websites. When scraping from these websites, our algorithm identifies the name, total time to cook the recipe, recipe ingredients, and recipe instructions from within the microformat.

With nearly 100 webscraped recipes in our database, we found that our webscraping algorithm properly parses the data 95% correctly. This 5% inaccuracy is due to recipe site authors incorrectly placing recipe data inside of their microformat data structure. This meets our FS7 requirement which expects at least a 90% accuracy for webscraping.

4.5.2 Text Processing Accuracy

The accuracy of our overall system depends on the accuracy of the text processing algorithm described in Section 3.3.1. We tested the accuracy of our text processing algorithm with 9 recipes. We first ran each recipe through the algorithm to generate a JSON file, and then modified the JSON manually to obtain our desired/expected output. Our accuracy analyzer compares the actual output JSON with the expected output by comparing the individual attributes of each node. The final percentage represents the degree of similarity between the expected and actual output.

However, not every attribute is weighted equally. We believed that the ‘ingredients’, ‘baseIngredients’, ‘resourcesRequired’, ‘stepTime’, and ‘holdingResource’ were the most important attributes for the recipe processing and optimization algorithm. Therefore, we assigned those attributes to have weights of 10%, 12.5%, 20%, 20% and 20% respectively. The less important attributes were ‘instructions’, ‘ingredientsQuantity’, ‘userTime’, and ‘holdingID’ have weights of 5%, 2.5%, 5% and 5%. The instruction is read from the formatted file, so its accuracy will almost always be 100% which is why we assigned less weight to it, focusing more on attributes that test the ability of the algorithm to understand the recipe steps. Across 9 different recipes, we obtained an average accuracy of 82%, which is considered to be a favorable and realistic accuracy [6].

4.5.3 Optimization Results

For our demonstration, we combined 3 recipes to create a meal for our project consultant. For this meal, a team of 3 friends cooked 3 recipes, including steamed broccoli, pork schnitzel, and cinnamon apple cake. To better compare recipe times, we assumed each user is set to an advanced skill level, so there is no time buffer taken for each task in the recipe.

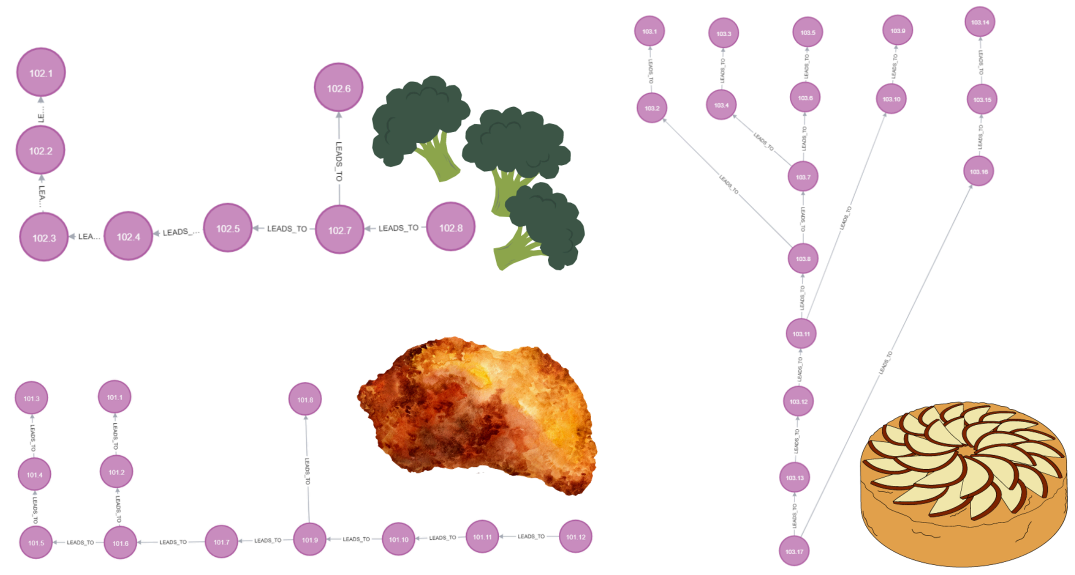


Figure 15: Dependency graphs of demo meal (top left - Broccoli, bottom left - Pork Schnitzel, right - Apple Cinnamon Cake)

Our first objective was to demonstrate an overall acceleration in cooking. First, we compared the time projected within the recipe URL to the critical path time in our dependency graph. The critical path time is the least amount of time it would take to complete a recipe assuming that all other branches can be done in parallel with the kitchen constraints and friends provided.

Note that each step has a predicted step time that we use to calculate the total critical path time. This prediction is done within recipe processing (See Section 3.2).

Table 4: Optimization Results - Cooking Time Comparison

Recipe	**Total Time**	**Predicted Critical Path Time**
Steamed Broccoli [7]	15 minutes	20 minutes
Pork Schnitzel [8]	45 minutes	22 minutes
Cinnamon Apple Cake [9]	65 minutes	71 minutes

Notice that pork schnitzel only requires 22 minutes using our optimization model. With further inspection, this is due to the preparation steps taking up the most time within the recipe. However, we have parallelized some of the preparation steps which reduces the total time.

After running the optimization algorithm, each user received a list of instructions. The total time each user spent is listed below.

Table 5: Optimization Results - User Time Allocation

User	**Time Spent on Tasks (User Time)**	**Overall Time Cooking (Cumulative Step Time)**
User 1	47 minutes	82 minutes
User 2	46 minutes	64 minutes
User 3	45 minutes	47 minutes

The second column is the time that the user actually spent cooking (e.g. chopping veggies). In contrast, the third column refers to the total time of the steps the user worked on (e.g. the user could place the cake in the oven within 2 minutes, but the cake needs 30 minutes to bake).

There are two observations we can take from these results. First, the total time to complete this entire meal is bottlenecked by whoever had the largest cumulative step time. In this case, User 1 spent the longest overall time cooking with 82 minutes. The second observation is that each user is working approximately the same amount of time.

Based on our project objective, we wanted to minimize the overall cooking time and load-balance the amount of time each user works. From our demo, the overall time to cook 3 recipes was 193 minutes (cumulative step time sum = 82 + 64 + 47). Based on the estimated times provided from the recipe URLs, the total time for all 3 recipes should be 125 minutes. This sets our predicted total time approximately 50% slower. However, when we tried cooking some of these recipes, we found the original time estimated to complete the recipes were too fast for a typical home cook to finish. Our algorithm actually allowed for more accurate step times.

As for load balancing, our algorithm perfectly load-balanced the number of minutes each user works. In this example, User 1 works 47 minutes, User 2 works 46 minutes, and User 3 works 47 minutes. We managed to load-balance the amount of work to roughly 46 minutes with a 1 minute error.

Therefore, our project meets the load balancing objective, but does not reduce the amount of time taken to cook all recipes. The time for cooking all recipes could be larger based on how our program predicts the step time. However, in our testing we found that these predictions, while longer, were more realistic to the time required for most people to finish these steps.

5. Discussion and Conclusions

5.1 Evaluation of Final Design

The final design has six subsystems that work together to meet the project functional and non-functional specifications. The most essential functional specifications (FS1, FS3, FS4) describe the main objective of the project, which is to reduce hassle in the kitchen by parallelizing recipe steps and equally distributing the amount of work across users. These are delivered by the scheduling and optimization algorithms.

The input processing specifications are implemented such that the user can manually input a recipe into the application using their phone's keyboard, thus achieving FS8. We described how to use webscraping recipes from a URL to meet FS7, but it only works if the recipe link includes a specialized microformat JSON. Since we cannot guarantee retrieval of online recipe data, this specification can only partially be met. FS8 was marked as non-essential, so this is acceptable.

The database specifications outline that we can store user data and recipe data to be continuously used in the future. The use of a relational database, MySQL, allows the user to input information about their kitchen equipment once when they configure the app instead of being prompted every time. This design choice fulfills FS5. Similarly, based on the allowed size of the free Neo4j graph database server, we fulfill NFS1 and store 100 recipes per user.

One of the essential specifications, FS6, describes taking in the user's skill level and using it as a constraint in the optimization algorithm. In our algorithm, we utilized the skill level to determine a time buffer for each task. The more advanced the user, the less buffer we provide. This allows less experienced users to have more time to finish their tasks, which also satisfies FS6.

Finally, we tested many of the non-functional specifications with NSF1 to NSF5 meeting requirements. NSF1 is satisfied by introducing over 100 recipes in a cookbook per user. With an application size of approximately 10MB, we successfully achieved NSF2. We tested our code on several emulators and two physical devices, each running Android 10.0.0 or higher, to meet NFS3. When testing our program, the server was 100% active and remained active throughout testing periods. This satisfied NSF4. Finally, during our programming and debugging sessions, we left the devices turned on for several hours at a time and noted the decrease in battery at the end of each session. We confirmed that Kitchen Buddies uses a minimal amount of battery; less than 10% per hour of use as specified by NSF5.

Overall, all of our essential specifications are achieved with the final design. After reviewing our non-essential specifications, some of these components were met including FS8. Some non-essential specifications were not met due to essential requirements taking priority.

5.2 Use of Advanced Knowledge

Throughout the planning of our project we involved many different concepts from our software design knowledge as well as upper year algorithms knowledge.

While planning server-side utilities of the project, we used the understanding of databases from ECE 356 (Databases). To reduce data redundancy and improve efficiency, we split up recipe metadata into a relational database and recipe steps into a graph database. We followed best practices for creating good relational designs by ensuring functional dependencies were accounted for and implementing one-to-one and one-to-many relationships. This approach enabled us to effectively organize and store large amounts of data, improving the overall performance of our application.

Within our optimization algorithm, we utilized task scheduling concepts similar to those found within ECE 350 (Operating Systems). Specifically, we use load balancing as the scheduling technique by checking the amount of time a user has been cooking. This approach prioritizes time-dependent tasks first, ensuring that users complete those tasks on time. Additionally, our algorithm equally distributes the amount of work assigned to each user, preventing any one user from being overloaded with tasks.

When implementing communication across devices and between frontend and backend systems, we relied on our knowledge of ECE 358 (Computer Networks). To enable client-server communication, we used HTTP protocols such as GET/POST, allowing devices to send and receive data from a server. To host an API service, we configured different endpoints using the Spring Java framework, allowing us to effectively manage the various functions of our application. Additionally, to enable client-client communication, we utilized Firebase Cloud Messaging, which allowed devices to communicate with each other in real-time. To ensure secure communication, we also configured firewall and port access with our DigitalOcean Droplet, which protected our application from potential security threats.

Our project involved working with complex software architecture concepts that required careful consideration and planning based on concepts taught in ECE 452 (Software Design and Architecture). To ensure that our application was designed effectively, we prominently used Client-Server Communication as our architecture style, which was necessary due to the distributed nature of data across a range of clients. We also utilized the Observer Design Pattern to send meal session steps to all of the host's friends. This pattern was necessary as their friends acted as observers, waiting for a set of meal session steps from the host (subject). This approach was necessary for our project's success, allowing us to effectively manage the real-time updates and notifications associated with our application.

Natural language processing (NLP) was a pivotal concept in our design that was integrated from MSCI 442 (Introduction to Machine Learning). For the input into recipe processing, recipe instructions were organized into a specific format to prepare for NLP. This involved tokenizing each word of a step and identifying relationships between words in a sentence using Python's spaCy library. These processes required advanced knowledge of NLP techniques and frameworks, as well as a deep understanding of the nuances of natural language processing.

5.3 Creativity, Novelty, Elegance

The creativity of this project is rooted in the concept of allowing multiple people to work on multiple recipes at once. This is a novel idea driven by our own struggles as university students in managing how to cook a meal in the shortest time possible, to allow time for studying and work. We chose to expand the scope of our original idea to allow multiple people to cook multiple recipes at once and had to come up with a creative way of allowing them to do so, landing on the idea of delegating steps of different recipes to different people.

The novelty of this project stems from the idea of gathering multiple people in a casual kitchen setting, to cook multiple recipes at once without a breakdown of communication. This is done by breaking down recipes into smaller tasks and redistributing these tasks among users to reduce the cooking time, with notifications to coordinate tasks that depend on each other.

This application provides a useful tool for people who are stressed, such as for the organization of large family dinners, because it removes some of the burden of coordination. Kitchen Buddies allows friends and family to focus on the enjoyable social aspect of cooking together, rather than the communication issues it can produce.

The elegance of the design is the graph database that generates a new way of examining the dependencies in each recipe, by encapsulating the details of each task in a node. The graph database cuts down on processing time since we can easily retrieve a previously-used recipe, already in graph form, and continue to use it for future meal sessions.

The simplicity of receiving real-time notifications in order to know when certain steps are urgent allows the product to be instantly approachable by new users. This was designed with ease of use in mind, encouraging more newbie chefs to use the platform.

5.3 Quality of Risk Assessment

Our risk assessment includes four risks from the 4A term that we took into account. Each table below provides a review of the risk and the method in which we mitigated the risk.

Table 6: Risk Assessment - Communication Errors

Risk: Communication between devices does not work consistently >10% of the time
Impact: Medium
Probability: Low
Situation: User would not be able to send notifications to friends which removes the ease of

use with our app by removing the communication between chefs
Remediation: By storing device tokens in Firestore, we can consistently access them and use FCM to provide reliable notifications. Using this trusted, standard service was beneficial because FCM will never drop a message even if someone accidentally turns off their phone; the message will only be delayed, which reduces the chance of communication errors [10].

Table 7: Risk Assessment - Task Parallelization Suboptimal

Risk: Task Parallelization Suboptimal
Impact: High
Probability: Medium
Situation: Parallelization of multiple recipes at once can result in the result: 1. Being slower than if each user made their own recipe 2. overcomplicating the set of recipes such as constantly switching someone between prepping food and cooking food
Remediation: Our algorithm focuses heavily on load balancing cumulative task time across users. This allows more work to be done in parallel, which decreases the total time to finish all recipes. Since we introduced time buffers for each task based on skill level, the total time improvement is only guaranteed if all chefs are set to expert skill level. In terms of over complicating the set of recipe steps, this risk was mitigated by our optimization algorithm. Within our algorithm, we begin assigning tasks from the end of the recipe to the beginning. This implicitly implies that the preparation steps will be assigned to users earlier on in their meal session rather than sporadically in between cook steps.

Table 8: Risk Assessment - Lack of Machine Learning Knowledge

Risk: Lack of Machine Learning Knowledge
Impact: High
Probability: Medium
Situation: Creators of the app may not have enough machine learning skills in natural language processing and image recognition to create certain essential components of the app
Remediation: Through MSCI446, one teammate reached out to Professor Lukasz Golab for guidance on potential solutions to our natural language processing (NLP) problem. He provided a framework that we used to extract components such as ingredients from a sentence. This framework utilized a Python package spaCy along with database queries, meaning no custom NLP required. Since we did not continue with image recognition, this risk was removed from our system.

Table 9: Risk Assessment - Real-world variation of devices

Risk: Real-world variation of devices
Impact: Medium
Probability: High
Situation: Real-life variations in devices and other miscellaneous cooking instruments that can result in steps taking longer or shorter periods of time than the app's algorithm (eg. stovetop being hotter than usual or heats up quickly)
Remediation: After refining our optimization algorithm, our model can handle overestimating these tasks. If preheating an oven takes 1 minute compared to 5 minutes, it will not affect the time dependencies of future tasks (i.e. bake chicken for 25 minutes). However, it does not work well for underestimating these types of tasks. To mediate the potential of underestimating the time needed for a device to heat up, we added a specific offset to ensure it would overestimate the time. This enabled us to mitigate this risk.

5.5 Student Workload

Throughout the past year the team has worked collaboratively to create this project. All members had equal involvement in the ideation and planning stages of the project. As we built the prototype certain people focused on different parts although everyone floated around when needed. Aurchon and Shadi created the frontend application. Aurchon and Maya worked on the Rest APIs. Marley and Caleb worked on Input Processing. Maya and Marley worked on Recipe Processing. Maya and Caleb worked on the Optimization algorithm. Shadi and Caleb created and managed the various databases. Some aspects required more effort and time and others required less effort and time in order to get that subsystem done. Since each of these aspects had varying levels of work which led to everyone handling equal amounts of work throughout the project.

References

[1] N. Farmer, K. Touchton-Leonard, and A. Ross, “Psychosocial benefits of cooking interventions: A systematic review,” *Health education & behavior : the official publication of the Society for Public Health Education*, Apr-2018. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5862744/. [Accessed: 17-May-2022].

[2] "Recipe Schema Markup | Documentation | Google Developers", *Google Developers*, 2022. [Online]. Available: https://developers.google.com/search/docs/advanced/structured-data/recipe. [Accessed: 21-Jun-2022].

[3] “Droplet Pricing,” *DigitalOcean*. [Online]. Available: https://www.digitalocean.com/pricing/droplets. [Accessed: 21-Mar-2023].

[4] “What you can't do with Neo4j”, *ArangoDB,* 2018. [Online]. Available: https://www.arangodb.com/solutions/comparisons/arangodb-vs-neo4j/. [Accessed: 20-Jun-2022].

[5] R. Bhardwaj, “Web Socket vs HTTP: What to choose for your next API Design”, *AAR Technosolutions*, 2020. [Online]. Available: https://ipwithease.com/web-socket-vs-http/. [Accessed: 20-Jun-2022].

[6] K. Barkved, “How to know if your machine learning model has good performance: Obviously ai,” *Data Science without Code*. [Online]. Available: https://www.obviously.ai/post/machine-learning-model-performance [Accessed: 02-Mar-2023].

[7] E. Bauer, “How to steam broccoli perfectly every time,” *Simply Recipes*, 19-Jun-2021. [Online]. Available: https://www.simplyrecipes.com/recipes/steamed_broccoli/. [Accessed: 21-Mar-2023].

[8] J. Cismaru, “Pork Schnitzel,” *Jo Cooks*, 28-Feb-2020. [Online]. Available: https://www.jocooks.com/recipes/pork-schnitzel/. [Accessed: 21-Mar-2023].

[9] J. Holmes, “The best cinnamon apple cake,” *Sweetest Menu*, 26-Aug-2020. [Online]. Available: https://www.sweetestmenu.com/cinnamon-apple-cake/. [Accessed: 21-Mar-2023].

[10] “About FCM messages,” *Google*. [Online]. Available: https://firebase.google.com/docs/cloud-messaging/concept-options. [Accessed: 21-Mar-2023].

Appendix A

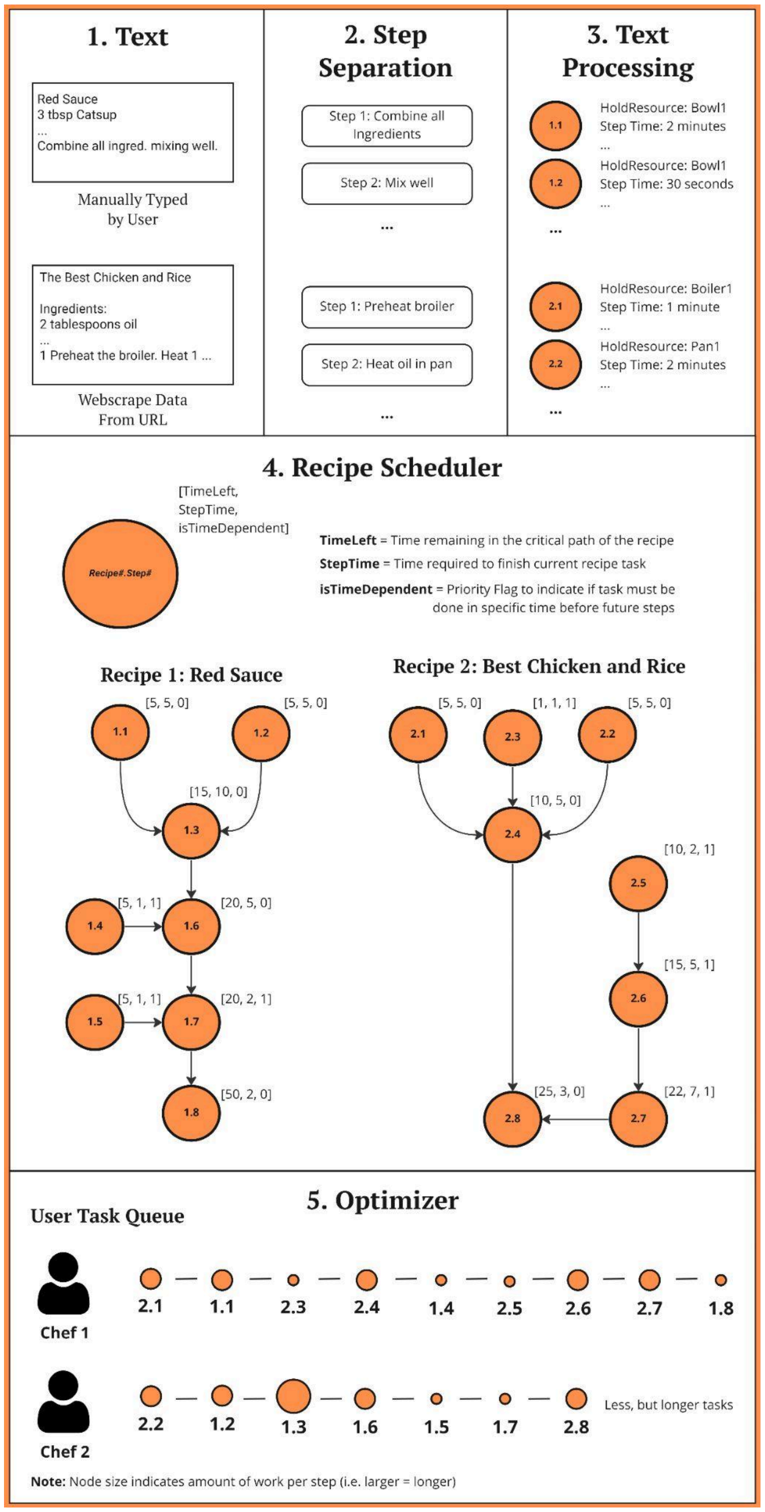
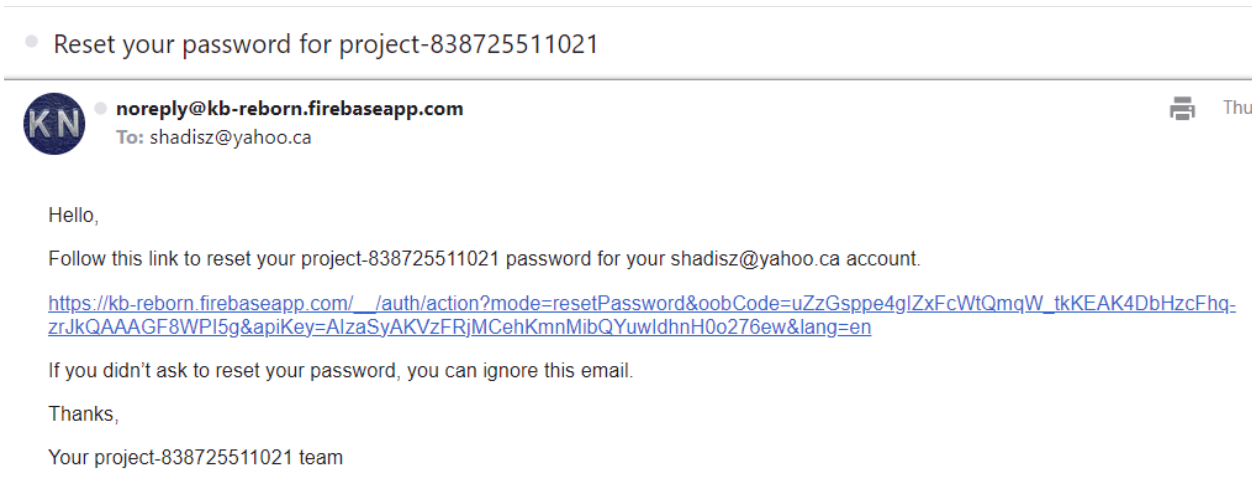


Figure 16 - Sample Workflow of 2 Recipes



Reset your password for project-838725511021

noreply@kb-reborn.firebaseapp.com
To: shadisz@yahoo.ca

Thu

Hello,

Follow this link to reset your project-838725511021 password for your shadisz@yahoo.ca account.

https://kb-reborn.firebaseapp.com/__/auth/action?mode=resetPassword&oobCode=uZzGsppe4glZxFcWtQmqW_tkKEAK4DbHzcFhq-zrJkQAAAGF8WPI5g&apiKey=AIzaSyAKVzFRjMCehKmnMibQYuwldhnH0o276ew&lang=en

If you didn’t ask to reset your password, you can ignore this email.

Thanks,

Your project-838725511021 team

Figure 17: Screenshot of an email with link to reset user password