GobbleUp Final Report

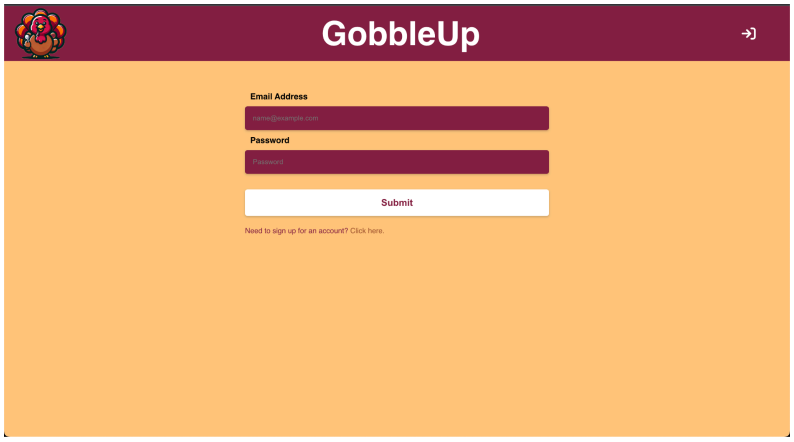
Kirk Knutsen, Daniel Hassler, Daniel Sabanov

Product Description

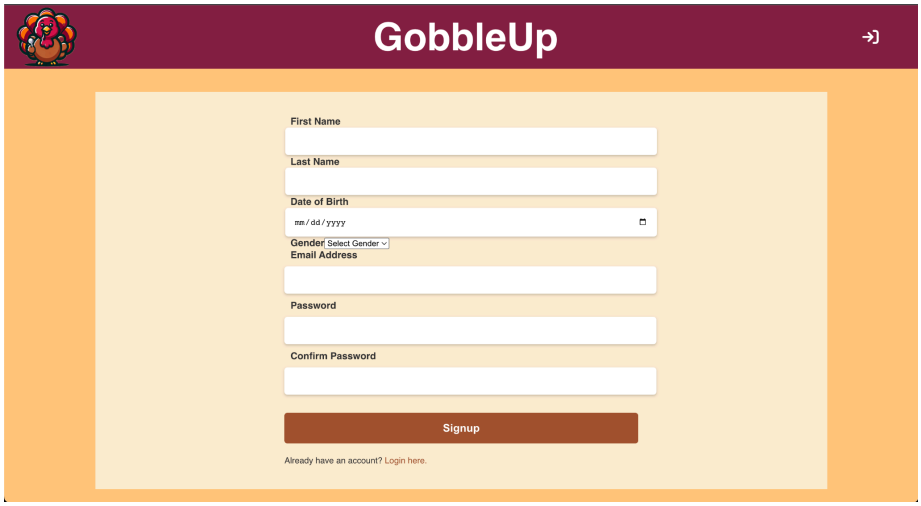
GobbleUp is a full-stack web application that aims to allow Virginia Tech (VT) students to view on-campus dining offerings, track and plan their meals for the day, and leverage AI-enhanced technology to suggest location and mealtime-specific meal plans. Additionally, our nutritional report offers a comprehensive summary of the user's dining activity in a readable manner, addressing one of our goals of assisting newly independent diners with better accountability and transparency. The application utilizes a simple UI with subtle VT theming and is accessible regardless of device or operating system.

Product Functionality

Upon visiting https://gobbleup.discovery.cs.vt.edu, our domain name hosted on the VT cloud, users are directed to the 'Login' page. Using the React framework router we ensure that logged out users are only ever able to reach the 'Login' and 'Register' pages. Assuming we are a new user, we'll first have to register, by clicking the 'Click Here' link.

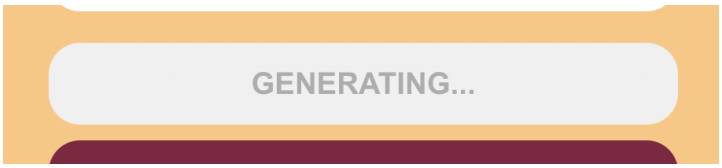


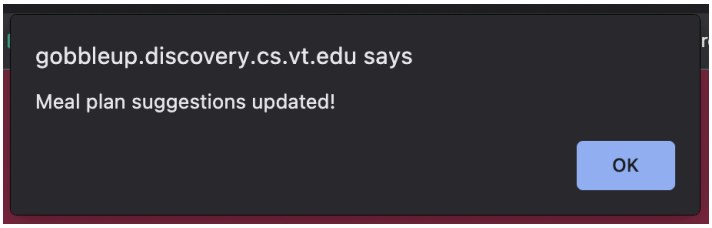
The registration page has users enter basic identifying information, as well as their login details. The email and password are both stored through FireBase, using oAuth to validate, while the biometric details and name are stored on our MySQL database. Basic password and email checks are done to prevent duplicate accounts or weak security, and then a confirmation pop-up appears letting users know their account has been created successfully and they are brought back to the 'Log In' screen.



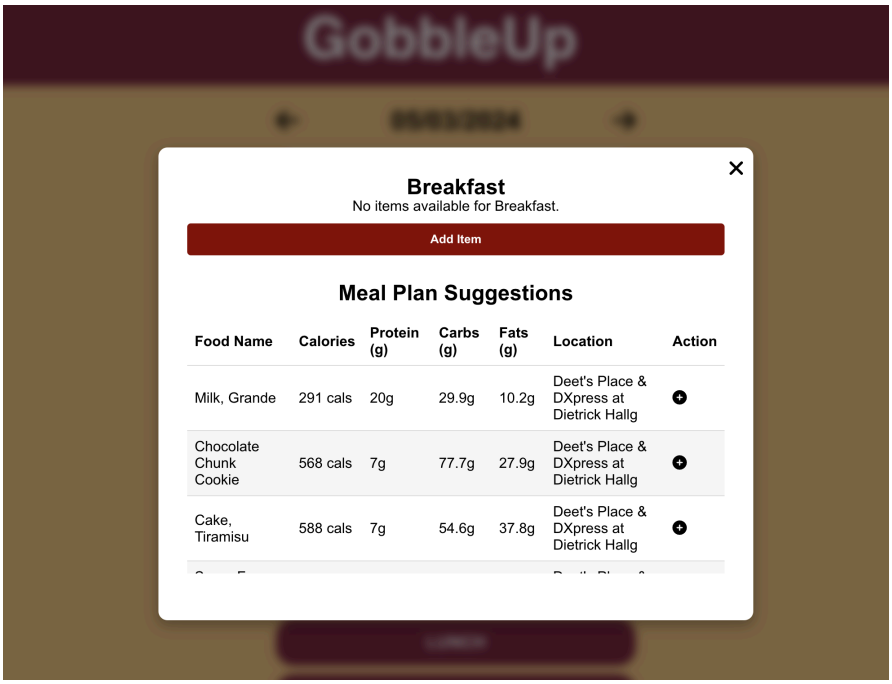
After the user logs in with their account details, they are brought to the ‘Dashboard’ screen, which acts much like a home screen. Their daily nutritional wrap-up is shown, and there are corresponding buttons for changing the date to view future or previous days, generating a suggested meal plan for the current date, and viewing their meal logs where they can add or remove items.

In this case, the user may be curious to see the suggested foods, so they click on generate meal plan. It changes to a shimmering gradient animation to indicate it’s generating, and then a pop-up appears to confirm upon completion.

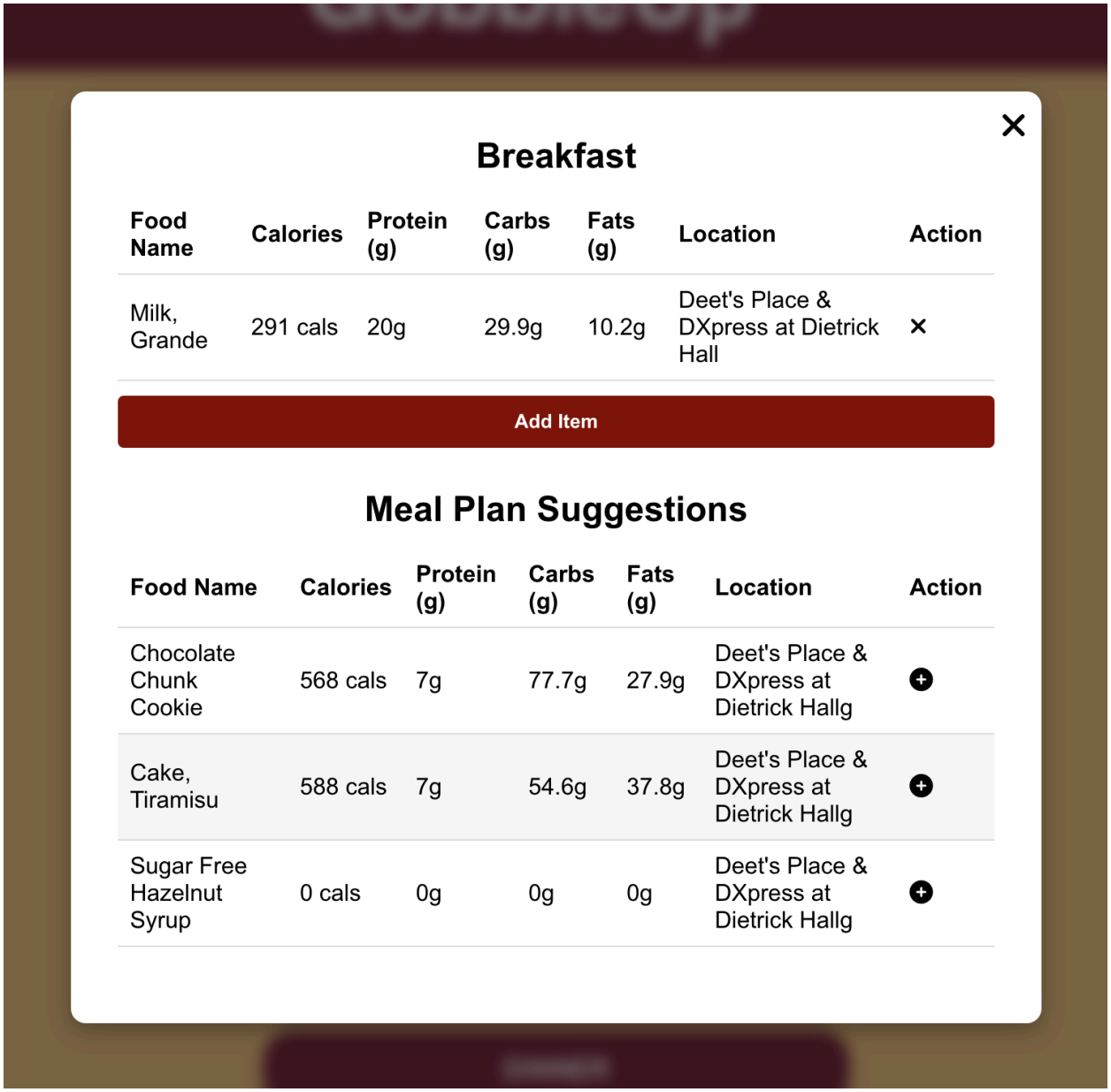




After a successful generation, you can see there are now scrollable suggestions in the breakfast table, mostly looking to suggest high-calorie items since this user has cardio as their current workout type, with a higher caloric intake. These items may be a bit sugary for us, so we'll just add milk for now by clicking on the '+' icon.



You can now see that the breakfast card reflects the milk item being added. Let's add another item not in our suggestions by using the 'Add Item' functionality.



Breakfast

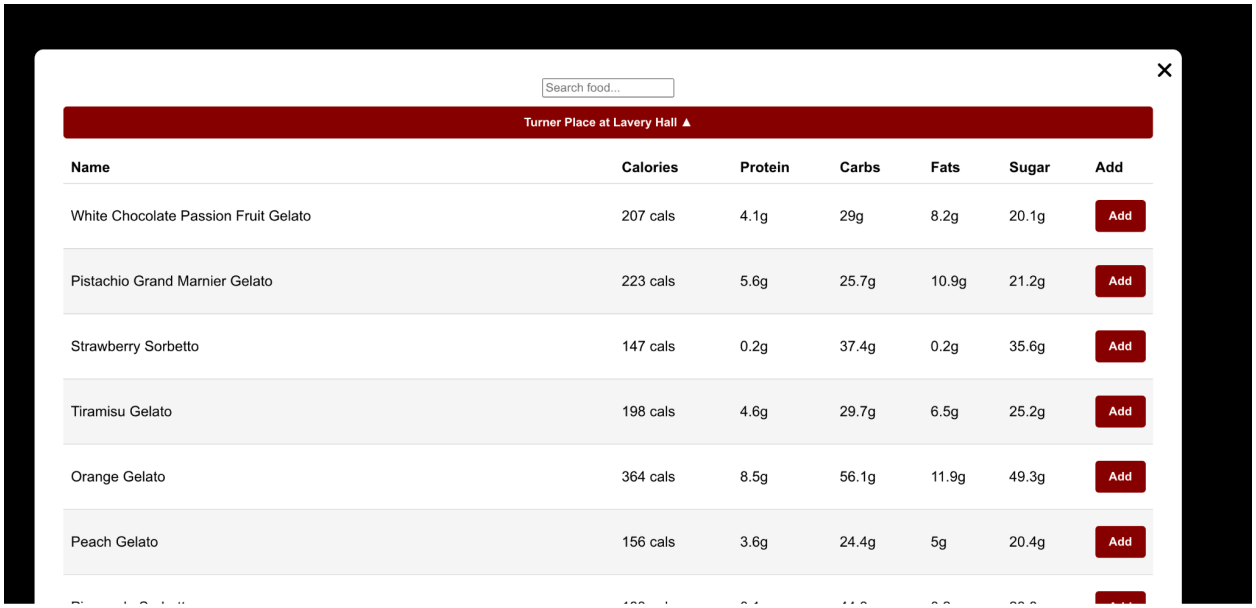
Food Name	Calories	Protein (g)	Carbs (g)	Fats (g)	Location	Action
Milk, Grande	291 cals	20g	29.9g	10.2g	Deet's Place & DXpress at Dietrick Hall	✕

Add Item

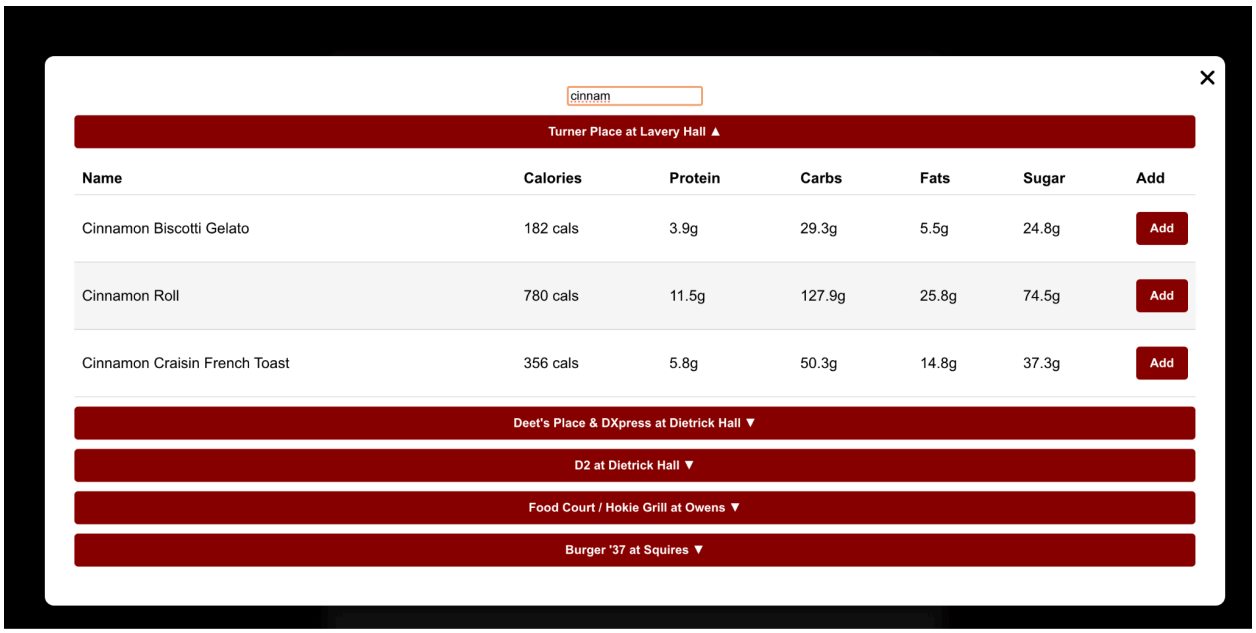
Meal Plan Suggestions

Food Name	Calories	Protein (g)	Carbs (g)	Fats (g)	Location	Action
Chocolate Chunk Cookie	568 cals	7g	77.7g	27.9g	Deet's Place & DXpress at Dietrick Hallg	⊕
Cake, Tiramisu	588 cals	7g	54.6g	37.8g	Deet's Place & DXpress at Dietrick Hallg	⊕
Sugar Free Hazelnut Syrup	0 cals	0g	0g	0g	Deet's Place & DXpress at Dietrick Hallg	⊕

On the ‘Add Item’ screen the food is sorted by both mealtime and location. This user had a friend bring them a cinnamon roll in class from Turner, so we’ll use the search functionality to find that item.



Now that it's appearing in the results, we'll simply click the add button and it will be added to the users meal.



Upon exiting both 'Add Item' and subsequently 'Breakfast' you can see that both items are reported in the nutritional summary. Clicking on the nutrition report allows us to see where calories are coming from by the three major energy forms. These are the most consistently reported nutritional values reported on the VT Dining site that we scrape from, but future work may expand the offerings.

GobbleUp

← 05/03/2024 →

Calories
473 kcal

Protein 23.9 g	Fats 15.7 g
Carbohydrates 59.2 g	Sodium 0.3 mg
Sugar 54.7 g	Cholesterol 0 mg

NUTRITION REPORT

GENERATE MEAL PLAN

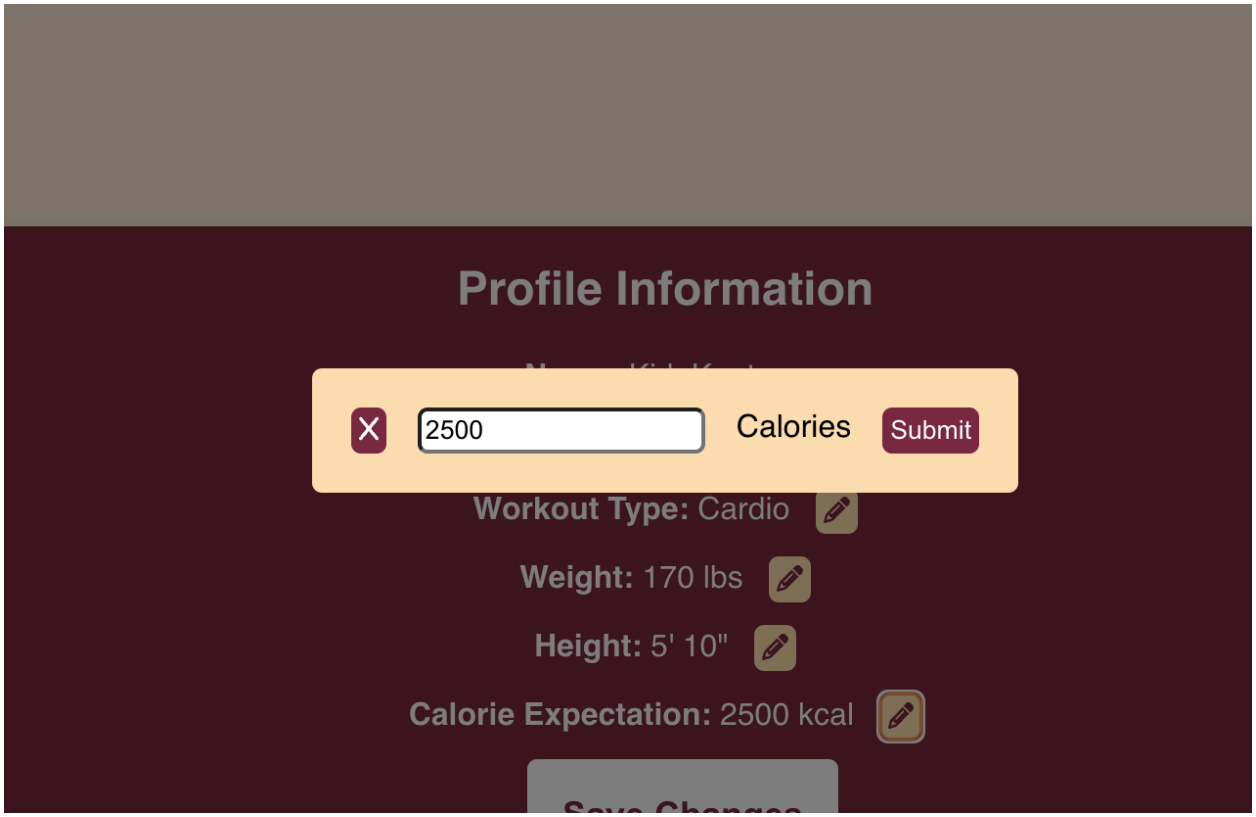
BREAKFAST

LUNCH

DINNER

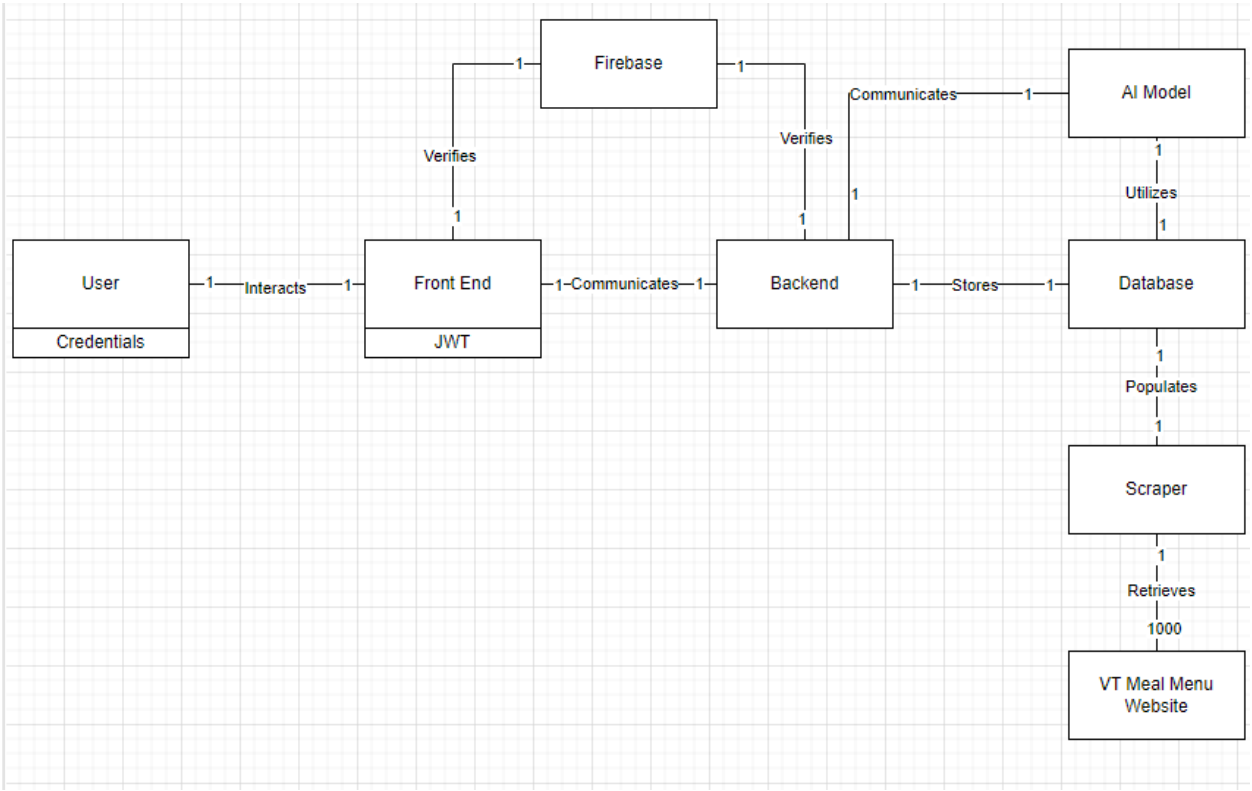
Finally, by clicking on the user profile icon in the header we can visit our user profile page. Here users can update their biometric info and workout type, which affects the meal plan generation requirements. As we continue to improve the generation by migrating to the GPT 3.5 model over Llama, these values will also have more impact on the suggested foods. All fields with a pencil icon are editable and will be changed in the database upon submission.


GobbleUp
Profile Information
Name: Kirk Knutsen
Date of Birth: 10/16/1996
Workout Type: Cardio
Weight: 170 lbs
Height: 5' 10"
Calorie Expectation: 2500 kcal
Save Changes


Profile Information
2500
Calories
Submit
Workout Type: Cardio
Weight: 170 lbs
Height: 5' 10"
Calorie Expectation: 2500 kcal

Product Design

Domain Model



The domain model depicts all of our microservices and their essential relationships, with each component uniquely isolated in our design implementation. In essence, the domain model describes the following interactions. The users interact with our front end and pass it their credentials. The front end then interacts with Firebase in order to verify the passed in credentials and receives back a JWT token that uniquely identifies the user. The front end passes the JWT token to the backend, which verifies it via Firebase. After verification, the backend either communicates with the AI model, or the database, depending on the nature of the request. The AI model also reaches out to our database in order to retrieve relevant data for meal plan generation. The database is populated with data from our scraper, which retrieves said data from the menu located on the VT dining website.

Application Hosting and Infrastructure

```
(base) grizzlord@grizzlord-VirtualBox:~$ k get pods
NAME                                        READY   STATUS      RESTARTS   AGE
gobbleup-aiml-backend                       1/1     Running     0          7d2h
gobbleup-backend                            1/1     Running     0          7d2h
gobbleup-bertopic-train                     0/1     Completed   0          18d
gobbleup-bertopic-train-1714266000-st8bg    0/1     Completed   0          6d
gobbleup-bertopic-train-1714611600-wz4pp    0/1     Completed   0          2d
gobbleup-frontend-68897df8b9-27lfk          1/1     Running     0          2d5h
gobbleup-frontend-68897df8b9-9s7ds          1/1     Running     0          2d5h
gobbleup-scraper-insert                     0/1     Completed   0          9d
gobbleup-scraper-insert-1714694400-8t4hd    0/1     Completed   0          25h
gobbleup-scraper-insert-1714780800-bl9vv    0/1     Completed   0          86m
llama2-7b-chat-7fc58cf564-pn487             1/1     Running     0          2d
minio-6b6d45bf65-zsdwm                      1/1     Running     0          37d
vtmealplandb-mysql-0                        1/1     Running     0          2d13h
(base) grizzlord@grizzlord-VirtualBox:~$ k get cj
NAME                      SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
gobbleup-bertopic-train   0 1 * * */4   False     0        2d              36d
gobbleup-scraper-insert   0 0 * * *     False     0        86m             9d
```

(Figure 0A: Gobbleup Pods)

```
(base) grizzlord@grizzlord-VirtualBox:~$ k get ingress
NAME                      CLASS    HOSTS                                ADDRESS                    PORTS   AGE
gobbleup-aiml-backend-ing <none>   aiml-gobbleup.discovery.cs.vt.edu    192.168.5.40,192.168.5.43  80      24d
gobbleup-backend-ing      <none>   backend-gobbleup.discovery.cs.vt.edu 192.168.5.40,192.168.5.43  80      24d
gobbleup-frontend-ing     <none>   gobbleup.discovery.cs.vt.edu         192.168.5.40,192.168.5.43  80      47d
gobbleup-llama-ing        <none>   llama-gobbleup.discovery.cs.vt.edu   192.168.5.40,192.168.5.43  80      23d
minio-ingress             <none>   minio-gobbleup.discovery.cs.vt.edu   192.168.5.40,192.168.5.43  80      37d
(base) grizzlord@grizzlord-VirtualBox:~$ k get svc
NAME                        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)            AGE
gobbleup-aiml-backend-svc   ClusterIP   10.43.73.94     <none>        5000/TCP           31d
gobbleup-backend-svc        ClusterIP   10.43.12.146    <none>        8080/TCP           30d
gobbleup-frontend-svc       ClusterIP   10.43.88.226    <none>        80/TCP             47d
llama2-7b-chat              ClusterIP   10.43.149.223   <none>        8000/TCP           23d
minio                       ClusterIP   10.43.46.251    <none>        9090/TCP,9000/TCP  37d
vtmealplandb-mysql          ClusterIP   10.43.174.206   <none>        3306/TCP           98d
vtmealplandb-mysql-headless ClusterIP   None            <none>        3306/TCP           98d
(base) grizzlord@grizzlord-VirtualBox:~$ k get pvc
NAME                        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-vtmealplandb-mysql-0   Bound    pvc-96a1affb-bb18-49a5-8e36-aac167022467   8Gi        RWO            ceph-rbd       98d
llama2-7b-chat-model        Bound    pvc-06c734d2-ae2f-414e-b86d-cdc64765fd22   5Gi        RWO            ceph-rbd       23d
(base) grizzlord@grizzlord-VirtualBox:~$ k get secret
NAME                                  TYPE                                  DATA   AGE
default-token-fdswh                   kubernetes.io/service-account-token   3      99d
firebase-config                       Opaque                                1      30d
firebase-file-name                    Opaque                                1      30d
minio-secret                          Opaque                                2      36d
sh.helm.release.v1.llama2-7b-chat.v1  helm.sh/release.v1                     1      23d
sh.helm.release.v1.vtmealplandb.v1    helm.sh/release.v1                     1      98d
vtmealplandb-mysql                    Opaque                                2      98d
vtmealplandb-mysql-token-lrkpv        kubernetes.io/service-account-token   3      98d
```

(Figure 0B: Gobbleup Service, Ingresses, PVCs, Secrets)

We decided to use Docker and Kubernetes provided by the VT discovery cluster. This decision was driven by a desire to improve application resilience and utilize powerful infrastructure, with intention to create a modern, cloud-driven application. This infrastructure also allows website access for external users via many Kubernetes service abstraction features and automatic DNS resolution.

Our choice was also influenced by the desire to harness more extensive benefits of Kubernetes as an application hosting platform, such as:

- Scalability: Increasing deployment sizes without downtime.
- Portability: Run applications consistently across various computing environments.

- Resiliency: High availability and failure toleration.
- Automation: Autonomously deliver application features without manual intervention.
- Load balancing: Distribute incoming network traffic efficiently.
- Management: Simplified application management and maintenance without entire application downtime.
- Service abstraction: Seamlessly connect services within the cluster.

By integrating these capabilities, we aim to create a robust, flexible, and efficient environment that supports our application's needs and enhances user experience.

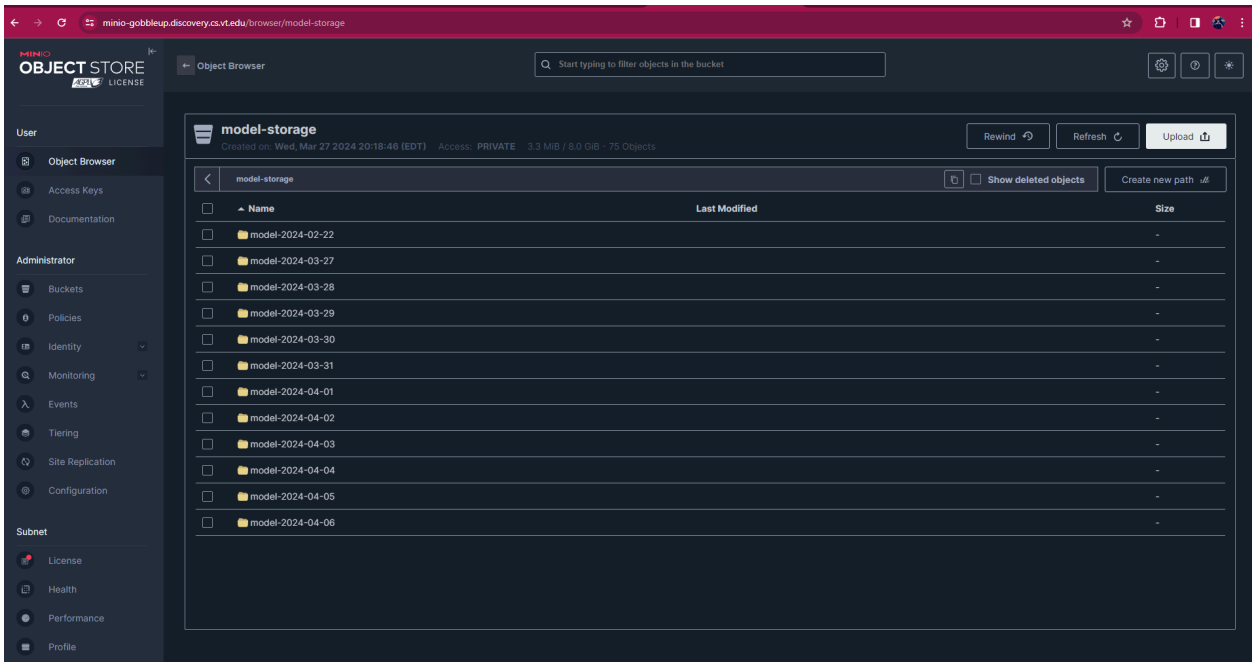
When it comes to integrating our application components into Kubernetes, we decided to approach our application design by ensuring strict isolation between components. This means separating features like backend into smaller parts like AI-backend, backend, database, scraper, etc. This allows us to leverage built-in Kubernetes features more efficiently.

We have multiple stateless and stateful pods that makeup our Gobbleup application depicted in Figure 0A:

- **gobbleup-aiml-backend**: a single pod that is hosted to run inference calls on our trained BERTopic models. First, it grabs model data by request and then generates meal plans based on predefined parameters and trained BERTopic clusters. The meal plans are then sent back via a POST request to any service that calls it. An example HTTP request would be formatted like this: "http://gobbleup-aiml-backend-svc:5000/mealplan" internally, or using our HTTPS route, https://aiml-gobbleup.discovery.cs.vt.edu/ defined by our ingress.
- **gobbleup-backend**: a single pod that represents the main backend for our entire application, being in charge of handling database requests directly from the frontend. An example internal HTTP request would be formatted like this: http://gobbleup-backend-svc:8080/", or using our HTTPS route, https://backend-gobbleup.discovery.cs.vt.edu .
- **gobbleup-bertopic-train**: AI model training is done through a Kubernetes CronJob, which runs a containerized workload to train several BERTopic models automatically every 4 days for each FoodDate entry in the database, and stores it in the Minio instance.
- **gobbleup-frontend**: two duplicate pods (for load balancing) for frontend access to our website. This is the pod that gets routed to by our main application link: https://gobbleup.discovery.cs.vt.edu/'
- **gobbleup-scraper**: VT dining website scraping is done through a Kubernetes CronJob as well, which runs a containerized workload to automatically scrape data once a day from the VT dining website, storing up to 6 days ahead in our MySQL database.
- **vtmealplandb-mysql-0**: This is our MySQL database holding all of our food data and location information. This service is internally accessible on port 3306 and accessed by other applications in our namespace by the service name. This database is leveraging PersistentVolumeClaims for storage, using 8 GB of disk space.

Supplementary components to our Gobbleup application that were essential for our application were:

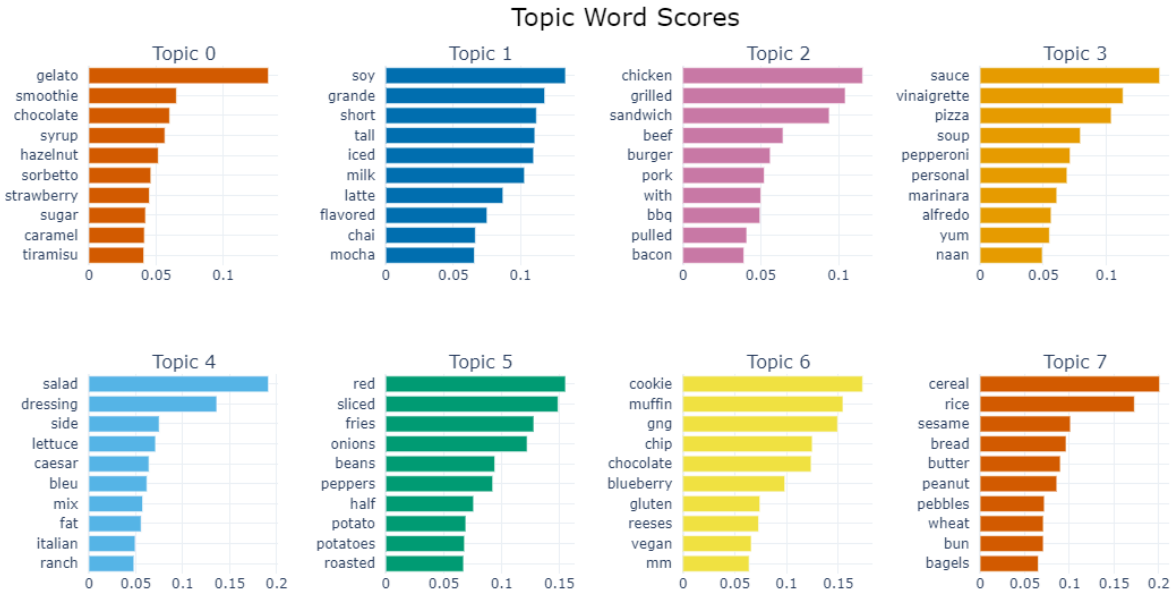
- **llama-2-7b-chat**: This is a hosted LLM pod on our namespace, using CPU for inference due to inaccessible GPU access on the discovery cluster. We are able to curl the model via a backend request in our other AI related applications to run Q&A prompting. This model is also leveraging a PVC, using 5 GB of storage. See AI section for more details.
- **minio**: We are running Minio as a modern AI model storage solution framework, which is an additional necessity to our application architecture. We adopted this architecture following the realization that our original intuition of having two applications share a single PVC was not feasible. The reason for this was that the discovery.cs.vt.edu cluster doesn't allow for "RWX" permissions for PVCs. Our implemented architecture, using Minio, both overcomes the "RWX" permission issue by exposing an endpoint and allowing for more features like storage administration, handling, and modifiability. The Minio console for our app is publicly accessible via https://minio-gobbleup.discovery.cs.vt.edu.



Pictured above is the Minio UI depicting our model storage configuration. Each model is stored in its folder with config files.

Here's a link to our application hosting demo: https://youtu.be/pm59Hwfa_2c

AI Meal Plan Generation Model



(Figure 1: BERTopic Clustering)

Related works with meal plan generation use LLMs or LSTM based approaches, which have downsides that we overcome. Our main advantage is the ability to adapt to our datasets, which offer different dining options every single day, in an automated, meaningful fashion. Most meal plan generation apps that incorporate AI are chatbots. The user usually prompts the LLM “What should I eat for dinner?” and the LLM responds with a good option, but whether that option is practical or location specific is not a guarantee. LSTMs on the other hand incorporate expensive design decisions like hyperparameter tuning, GPU access, and potential over/underfitting on a static dataset.

Our AI application aims to address challenging issues that most other meal plan generation algorithms do not have. Data persistence, up-to-date custom models for each food offering day in our database, on-demand fast training, dataset specific generation, and the ability to tailor user preferences and specific on-campus locations. To start this approach, our goal is to group food items together in a meaningful way, so that we can create diverse, semantically rich clusters of food items given our unique dataset. These clusters would allow us to generate diverse meal plans with more refined search criterias, like "Give me the seafood options" or "give me the vegan options". The intuition of this approach stems from the idea **Items that are semantically similiar hold similar nutritional information**. For example, Chicken and Pork should be in the same cluster, whereas Chicken and Wine are more unlikely to be in a cluster. For the AI portion of our project, we present a **novel approach to meal planning generation**, different from many other AI meal planning applications out there.

We are applying BERTopic, based on the paper BERTopic: Neural topic modeling with a class-based TF-IDF procedure (https://arxiv.org/abs/2203.05794) with LLM guided cluster labeling via Q&A prompting. The BERTopic model is stacked into many different components such as SBERT embeddings for semantically rich text embeddings, UMAP for dimensionality reduction on the large embedding space, HDBSCAN on the reduced embeddings, and a TF-IDF procedure for class based topic modeling. Though we had a few ideas regarding the approach of smart meal plan generation, we decided to approach the problem by grouping semantically similar food items together into clusters, which BERTopic achieves through its architecture. The intuition here is to group food's by their purpose, like high protein foods into one cluster or drinks in another cluster. For example, by our visualization for topic 2 (see Figure 1), it appears that the model satisfied our objective by predicting a "protein" cluster, as foods like "chicken", "beef", "pork" are in the same cluster. Furthermore, another example cluster we have represents "salads" (topic 4), with topics like "caesar", "lettuce", "dressing" in the same cluster. These clusters are semantically meaningful and provide a solid grouping for meal planning and organization, tailored to our specific dataset. Though clusters have strong, semantically rich groupings, we noticed that they don't have overall meaningful cluster labels directly from BERTopic. For example, the "chicken", "beef", "pork" cluster is labeled as "Topic 2"; to overcome this issue, we leveraged the power of LLMs (large language models) to directly label a topic in an automated fashion via Q&A prompting. Below is an example of our Q&A prompt we pass into our LLM.

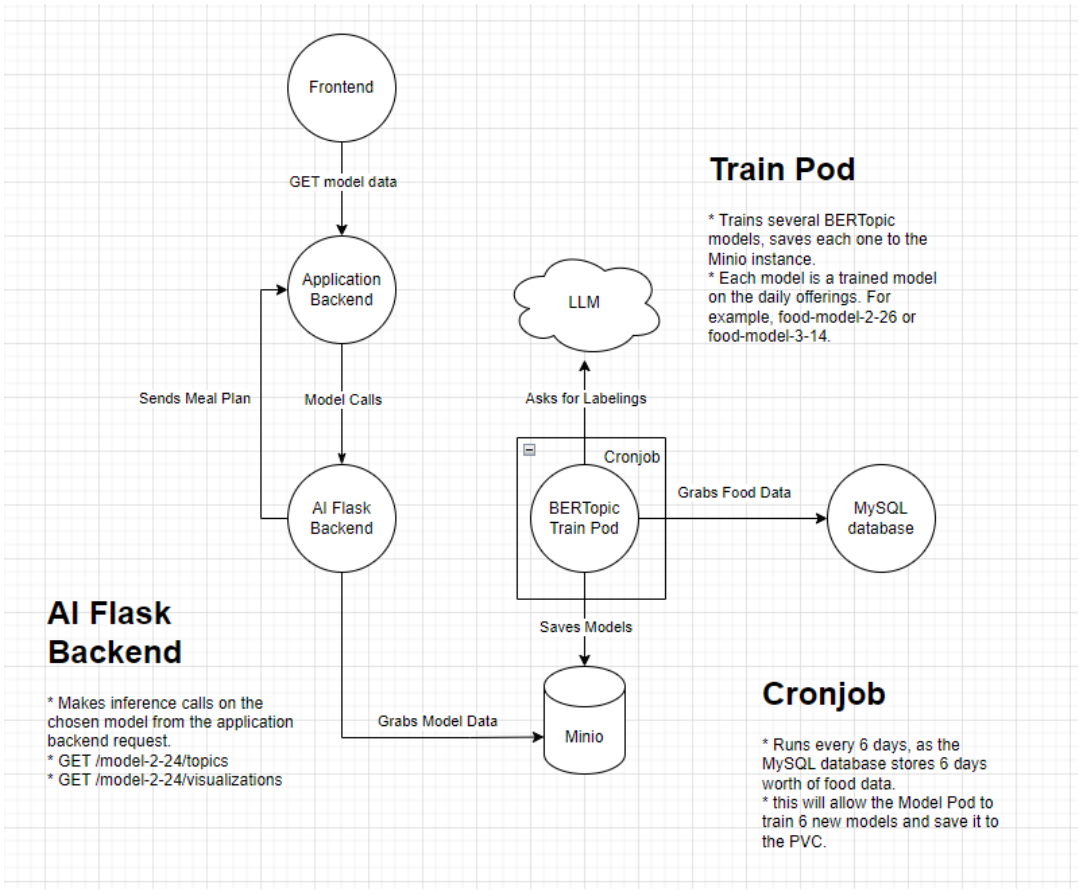
```
Given this cluster topic: ['danish', 'tall', 'bread', 'aulait', 'cafe', 'rice', 'fresh', 'pork', 'chai', 'apple'], choose the best category:
A. Coffee
B. Protein
C. Drink
D. Carb Heavy
E. Seafood
F. Fast Food
G. Sugary
You must respond with ONLY the letter associated with the correct option (A-G). NOTHING MORE.
```

(Figure 2: LLM Q&A Prompt)

To take advantage of free software and department resources, we decided to deploy a LLaMa-2-7B LLM on the VT discovery cluster. This decision allows our entire meal planning application to be directly hosted on VT hardware, making it VT proprietary and avoiding vendor lock-in like using a GPT service. Though it's important to note here that there are trade offs with this decision. Because we don't have NVIDIA GPU access on the discovery cluster, this LLM is running inference on CPU, making this a slower alternative. Also, we do acknowledge inherent limitations with LLaMa such as the ability to generate thoughtful and correct responses. For our future work in this space, we plan on obtaining an OpenAI access key and running inference on a GPT-3.5 model, as the monetary cost associated with each query has significantly dropped since the forthcoming of GPT-4. Regardless, our LLaMa model, though not always accurate, produces a conceptually good approach for using LLM guided cluster labeling and does produce meaningful results in automated fashion for meal plan generation.

One big advantage of using BERTopic + LLM guided labeling over other alternatives is the ability to create custom topic labels. For example, the AI designer is not locked into any specific labels. We can modify these labels to be “Seafood”, “Chicken”, “Soft Drinks”, “Alcohol”, “Condiments”, anything we want, without changing the underlying dataset, continuously allowing data persistence and accurate offerings. All this would require is a change to the Q&A prompting answers, and the LLM will do its best to fit certain labels onto certain clusters. This allows us to create “smarter”, more diverse meal plans, tailored to on-campus dining dataset and other preferences.

When it comes to design complexity and resource tradeoffs, we present our cloud-based hosting architecture for our AI components:



(Figure 3: AI Training Infrastructure)

Depicted in Figure 3, another advantage to our approach is the training and resource time complexity of the model. Because our dataset is rather small for each day in our database, being around 1,100 daily food offerings, training this clustering model is fairly quick at around a minute per model on a Kubernetes pod without a GPU. Our specific implementation is designed to be as resource-efficient as possible for this methodology, utilizing PyTorch for CPU to avoid installing superfluous NVIDIA drivers. This focus on resource efficiency and modeling choices

enables us to construct and deploy containers rapidly, as an alternative to this approach would be deep learning or strong LLMs, which are either too much overhead, or incur a monetary cost.

This interaction diagram in Figure 3 depicts how our frontend interacts with the AI components in our application. First, the frontend makes a request to the application backend, then the application backend asks the AI backend to make model calls; the model calls that are sent back to the application backend include “/mealplan”, which is a JSON list formatted generated meal plan for the day. An important aspect of our application's infrastructure is the implementation of a Minio instance for model data storage. Minio is a modern storage management solution commonly utilized in cloud-based machine learning applications for Kubernetes. The model training is done autonomously using Kubernetes built-in automation jobs called CronJobs. Every 6 days, the BERTopic-train-pod will run to train several different BERTopic models (one for each day in our FoodDate table). These models are saved into the Minio instance so the AI Flask backend can grab the data efficiently.

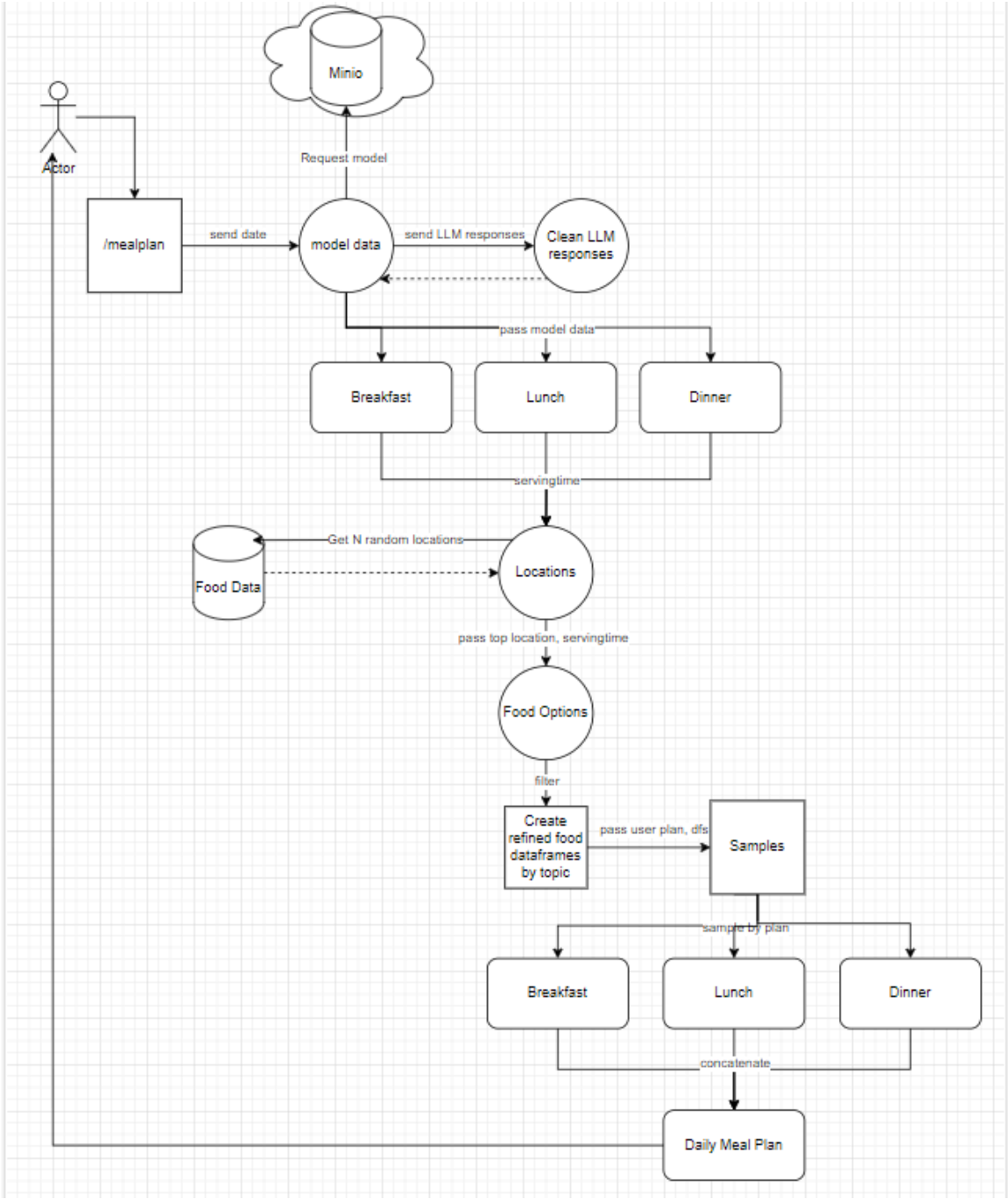
With this infrastructure we are able to leverage the power of AI and Kubernetes in a unique way with key features like automation, resiliency, load balancing, and storage management, directly enhancing the performance of our AI training and hosting applications. Our lightweight, yet powerful AI model for meal-plan generation encapsulates the idea of resilient, dynamic, and featureful meal plan generation.

Here is a link to our AI/ML demo: https://youtu.be/WG1a-bWmDUY

AI Flask Backend

In order to retrieve meaningful information from our AI model, we need to be able to make inference calls, which involves making a separate AI backend component coined as our “AI Flask Backend”.

The key functionality with our AI-backend Flask application is to generate meal plans and make model calls tailored to the user’s preferences. This is essentially running inference calls on our BERTopic + LLM guided model based on user preferences. Because we are using Kubernetes as the backbone to all of our applications, we are trying to ensure isolation between each separate backend component in the event of application failure and better load balancing, which is why the Gobbleup AI backend is separate from the Gobbleup main application backend.



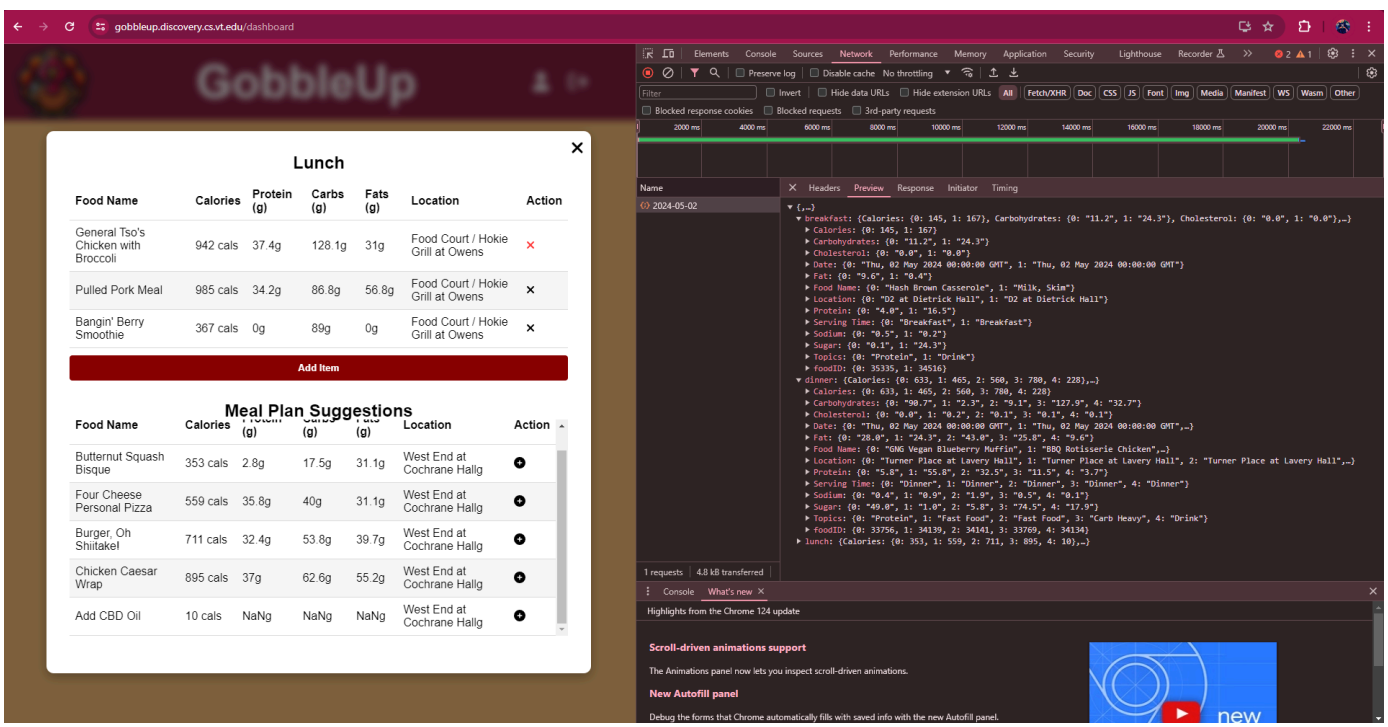
(Figure 4: /mealplan route)

This interaction diagram, Figure 4, depicts the workflow of the “/mealplan” route in the AI-backend, demonstrating a top-level overview of how generated meal plans are made and sent to the user. First, the actor sends a GET request to “/mealplan/<user>/<date>”, which goes through these steps in order:

1. Given the date, grab all model data from Minio and populate the variables.
2. Filter the noisy LLM responses into just the selection it chose.

3. Pass all model data into a loop, which loops over the serving times “Breakfast”, “Lunch”, and “Dinner”.
4. In the loop, given the serving time and date, select a random location from the database that fits the criteria.
5. In the loop, then given the location, serving time, and date, select all the foods offered for that query. Save this into a dataframe.
6. In the loop, refine the food data into logical categories (separate dataframes), from custom labels selected by the LLM (Example: Topic = “seafood”) and other refined queries (Example: Protein > 50).
7. In the loop, grab samples from each of the refined dataframes by the user’s workout plan.
8. Append each serving time result into its own dataframe (Example: breakfast generated foods).
9. Send JSON to user detailing all the generated plans, one plan for each serving time.

The /mealplan route is built to be very customizable, flexible, and have room for tuning and nutritional improvement. Because we are not nutritional experts, we employed our own interpretation of a healthy meal plan for each workout type. With more assistance from VT Nutrition or Fitness, we can improve our nutritional output via inference tuning. This would be a very easy change on our end if necessary.



(Figure 5: /mealplan example)

Figure 5 depicts our /mealplan route, in action, in our Gobbleup application. We are able to send breakfast, lunch, and dinner options to the user in an automated, intelligent way, responding back with plenty of dining information including location, food IDs, food types, food information, etc.

Go Backend API

Our backend API serves as a connection point to our database. The API consists of the following routes:

POST /auth/register

Responsible for registering new users. When a new user is signed up, the front end first registers the user with Firebase. Firebase, returns a JWT token that identifies this user. The front end then forwards said token to the backend API along with the user's first name, last name, date of birth, gender. This data is then registered in the database along with a unique user ID token that is received from firebase API when the JWT token is decoded. This allows us to recognize each user on subsequent logins. This route requires a JWT token, but is not considered a protected route.

GET /meals/food/?date={YYYYmmdd}

Responsible for fetching all the food items that are served on the specified date. The year is passed as "YYYY", month is passed as "mm" and day is passed as "dd". The route returns all the food items as a list of JSON objects where each item consists of the food id of the item, food name, number of calories, amount of fat, amount of cholesterol, amount of sodium, amount of carbs, amount of sugar, amount of protein, serving time, and location - meaning dinning hall.

GET /meals/history/?date={YYYYmmdd}

Responsible for fetching the user's meal history. The front end specifies the date in YYYYmmdd format as a URL parameter. The backend then returns a list of JSON objects that consist of a meal id, and a nested object of food items that mimics the structure of objects described in the "GET /meals/food/?date={YYYYmmdd}." This route helps to keep track of the user's eating habits. This is a protected route.

POST /meals/history/

Responsible for saving the user's previous meals in the database. The front end specifies a food ID and the date on which the food was eaten in the request's body. The backend API then stores that data. This food will then later show up when the front end uses the "GET /meals/history/?date={YYYYmmdd}" route. This is a protected route.

DELETE /meals/history/?foodId={mealId}

Responsible for removing a meal from the user's history. The front end specifies the meal ID as {mealId}. The backend API then removes the meal with the matching meal ID from the user's eating history. This is a protected route.

POST /user/biometrics
Responsible for updating the user's biometric data. The front end specified the user's workout type - the workout type can be "cardio", "bulking" and "aerobics". Additionally, the front end also specifies the user's calorie expectation, user's weight, and user's height. These all are stored in the database. The body may specify only one value out of these fields, in which case only the said value will be updated. This is a protected route.

GET /user/biometrics
Responsible for providing the font end with the user's biometric data. This route returns a JSON object that contains the user's workout type, calorie expectation, weight, height, first name, last name, and date of birth. This is a protected route.

The routes in the backend can be separated into two types: protected and unprotected. Unprotected routes can be queried by anyone without requiring authentication first. Protected routes require the client to first authenticate with firebase and pass a JWT token in the request header under "X-Firebase-AppCheck". The token is then decoded into a user ID that is used to store and fetch all the data that is relevant to the specific user.

The decision to use Firebase for authentication was reached due to the complexity of implementing proper authentication. As Firebase is a large platform that is specifically designed to provide authentication, we have decided that it would be best to use this solution rather than trying to implement our own authentication, which could be less safe for the user's data. Therefore, we use Firebase for user registration and authentication, as it would provide much better authentication capabilities and thus would provide greater safety for our users' data.

Scraper

The role of the scraper is to retrieve the data about the foods that are served in the VT dining halls. It does so by scraping the VT dining services menu.
We had to strike a tricky balance between retrieving the food data as quickly as possible and also minimizing the disturbance we would be creating with the additional load on the VT dining website. To strike this balance we have implemented two strategies: caching and throttling. Each food nutritional data is cached to guarantee quicker retrievals and avoid making additional requests. Additionally, we implemented throttling in order to reduce the total number of requests per second and thus reduce the load on the VT dining website. Currently, we are limiting ourselves to ten requests per four seconds.

The process of scraping the VT dining menu is as follows. First we retrieve all the food item names along with the time at which each item is served and a link from which we can access the nutritional data from each dining hall. The number of requests here is limited to the number of dining halls, as each dining hall lists all of the foods that it serves on a single page. The scraper then retrieves nutritional data for ten items at a time. This is the step when the food data is cached. The scraper then uploads the ten items to our database, providing some time for the VT dining website to recover from the load we placed on it, while also not spending time just

idling. The scraper then waits until four seconds have passed since the last request, and requests new data. The cycle then repeats until all the food data is scraped.

Database

The database is responsible for storing all the foods that are served at the dining halls at specific dates, which is collected by the scraper, as well as the data about our user's. The database consists of the following tables:

Food

Responsible for storing all the foods that are fetched by our scraper. Each row in the table consists of the food name, food calorie, amount of fat, amount of cholesterol, amount of sodium, amount of carbs, amount of sugar, food serving time - meaning "breakfast," "lunch," or "dinner." Additionally, it also stores the location where the food is served and a unique ID identifier within the database for easy access.

FoodDates

This table is responsible for monitoring the dates at which each food item is served. The table contains a food ID that links the entry to the Food table as well as a date at which the food item is stored. Each food item may appear in this table multiple times - once for each date that it is served on.

User

This table is responsible for storing the user data. This table contains the user's name, email, date of birth, gender, and a unique id token that links it to the corresponding user within Firebase.

WorkoutPlan

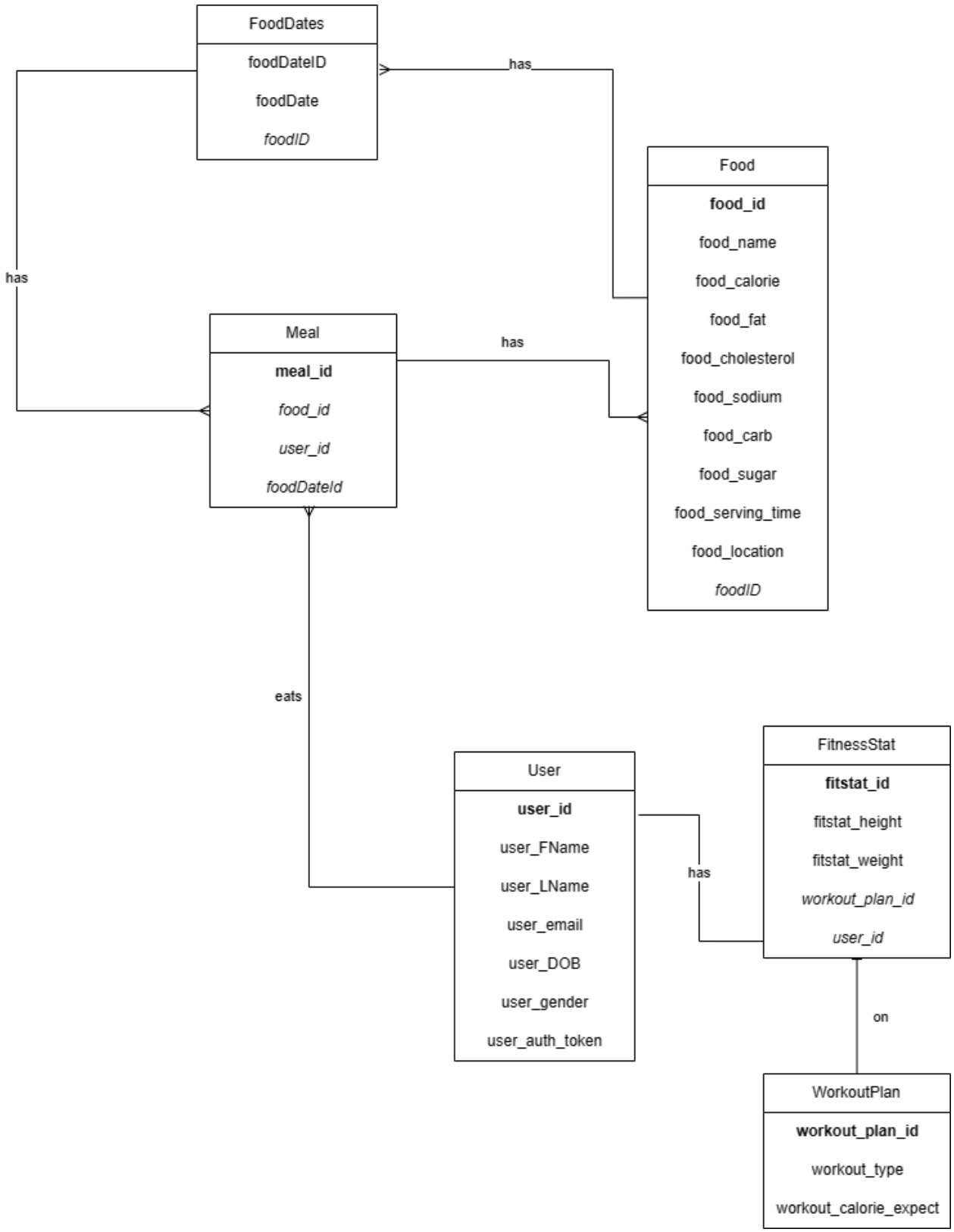
This table is responsible for storing the workout related data about each user, such as the user's workout type and the user's expected calorie intake. However, this table does not link to the user table directly. Instead, it links to the FitnessStat table, which then links to the user.

FitnessStat

This table monitor's the user's fitness statistics. In our case, that primarily refers to the user's weight, and height. It also serves as a link between the User table and the workout plan table.

Meal

This table is responsible for storing the user's previous meals. It essentially serves as a link between the user table and the food table. Each entry here references the Food table, the ser table, and the FoodDate table. This is done to indicate what food was eaten by what user, and when was it eaten. It is necessary so we could provide the user access to their eating history.


FoodDates
foodDateID
foodDate
foodID
has
Food
food_id
food_name
food_calorie
food_fat
food_cholesterol
food_sodium
food_carb
food_sugar
food_serving_time
food_location
foodID
has
Meal
meal_id
food_id
user_id
foodDateId
has
eats
User
user_id
user_FName
user_LName
user_email
user_DOB
user_gender
user_auth_token
has
FitnessStat
fitstat_id
fitstat_height
fitstat_weight
workout_plan_id
user_id
on
WorkoutPlan
workout_plan_id
workout_type
workout_calorie_expect

The database was implemented in MySQL as it is a simple relational database that is very common in the industry, and therefore has a large community of users for support. We have decided against using PostgreSQL as it is marginally more complex than MySQL and therefore would result in more additional work setting it up, and not focusing on the project as a whole.

Retrospective

We were able to successfully achieve all aspects and functionality of our VT centric meal planning application including: a modern, simple frontend design, creative hosting architecture that enhances user experience, intelligent meal plan generation utilizing AI, and food offering consistency. These points, and many more, allow VT students to take advantage of dining service options more effectively, greatly benefiting the VT dining community.

All in all we accomplished our goals within a tight timeframe and with a small team, so we're very pleased with the result. Our communication and teamwork was definitely our strongest aspect, and it allowed us to overcome issues and roadblocks without stress or conflict. We were always readily available to meet over Discord or in person, which ensured accountability.

Our hiccups were mostly in the form of time management, full stack integration, and testing. We were all taking on a fairly high volume workload classes, and if another project took one teammate's time, that took out a third of our development team temporarily. In turn, if one aspect of development was reliant upon another for any dependencies, development could be halted for a bit. While all of the functioning parts turned out great, it took a good amount of work linking them all together, especially with the complexity of Kubernetes. Finally, we would have liked to spend much more time on test driven development, but our timeframe didn't allow us to do so

For future work, we still plan on making improvements to our application, especially in the frontend and AI portion of our application. For the AI components, one immediate improvement we plan on making is swapping the LLM with a GPT based model, as monetary costs for legacy versions have plummeted due to GPT-4's introduction to the public. This would allow our topic labels to be more accurate, as GPT-3 performance is much better than our current LLaMa implementation. This would also reduce hosting demands on our current namespace and free up some resources on the discovery cluster.