Team 31–Final Paper–CS144 -Sp’25

## Documentation

### **a.** Frontend Layer

* React SPA (Port 3001)
  + Written in TypeScript with Tailwind CSS
  + Single-page routing via React Router
* WebAssembly Module
  + AssemblyScript implementation of the Haversine formula
  + Calculates walking distances and estimated calories burned
* Service Worker
  + PWA offline support (asset caching, graceful fallback)
  + Registers via Workbox for installability and background sync
  + Data flow: User’s browser ⟶ HTTPS ⟶ Express API

### **b.** Backend Layer

* Express.js API (Port 8080)
* Node.js 18+, TypeScript, middleware (Helmet, CORS, body-parsers)

**API Routes**

* /recommendations → personalized diet-based suggestions
* /menu → today’s menu & nutrition data
* /scraper → scrape triggers, status checks

### **c.** Database Layer

* MongoDB Atlas
  + Stores MenuItem, Restaurant, UserSession, RecipeMaster, WeeklyMenu  
    Mongoose ODM schemas enforce validation & full-text indexes
* Redis Cloud
  + Caches hot queries (menu data, recommendation results)
  + Key TTLs, session caching for anonymous tracking

### **d.** Business Logic and Servers

* **Recommendation Service**
  + Ranks foods by protein-to-calorie ratio
  + Applies distance calculation & user goal filters
* **Menu Service**
  + Aggregates today’s menu, search by keyword
  + Summarizes nutrition & filters by dietary tags
* **Restaurant Service**
  + Geolocation queries for nearest dining halls
  + Returns hours, real-time status, and cuisine type filters
* **Cache Service**
  + Encapsulates Redis integration
  + Manages cache keys & TTL for menus and sessions

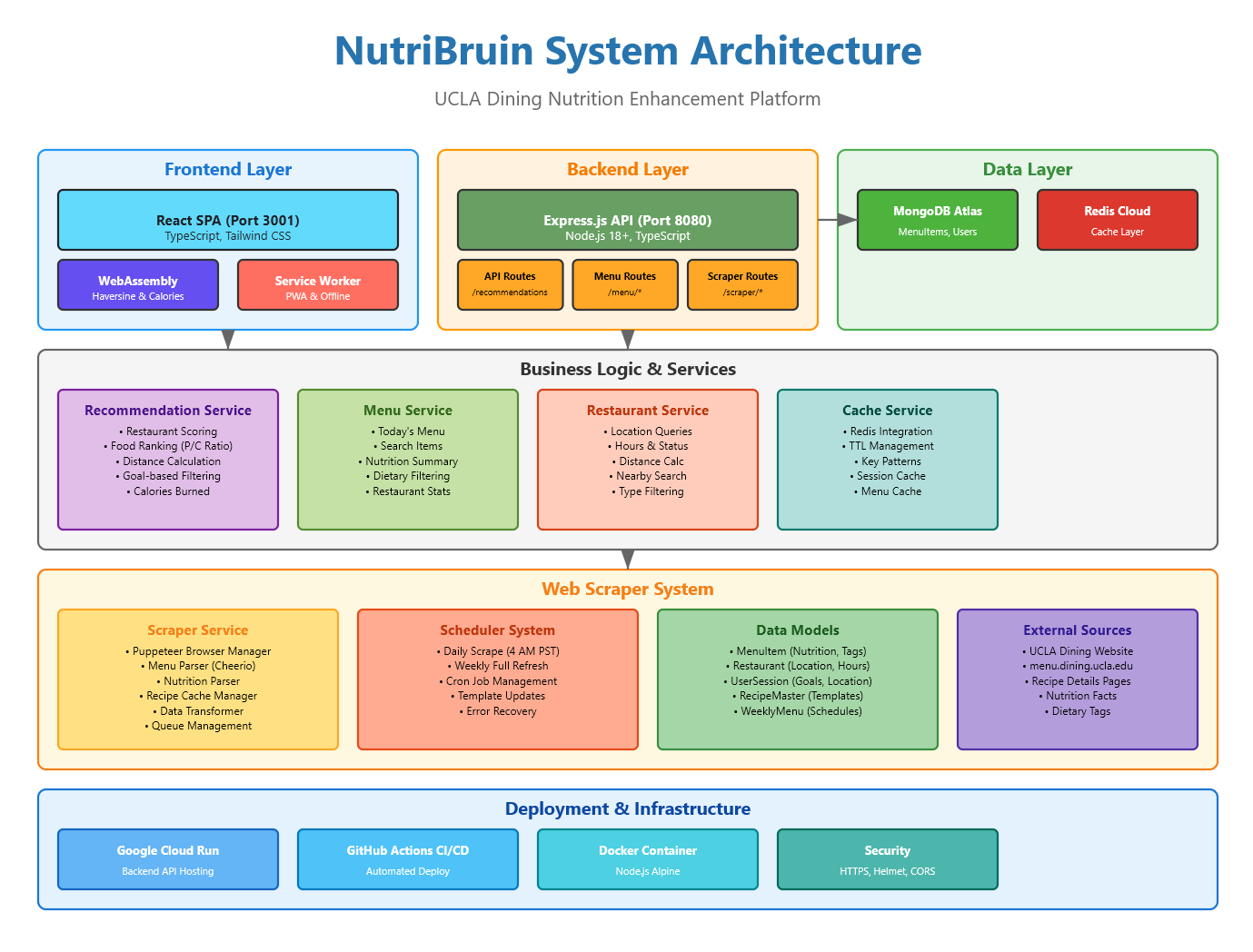
#### **5. Web Scraper System**

* Scraper Service  
  + Puppeteer browser manager + Cheerio parsers
  + Extracts menu items, nutrition facts, dietary icons
  + Caches recipe templates, transforms raw data
* Scheduler System
  + Cron jobs: daily scrape (4 AM PST), weekly full refresh
  + Template updates, error recovery & retry logic
* Data Models
  + Ingested into MongoDB via Mongoose
  + Includes menu schedules, nutrition tags, user goals
* External Sources
  + UCLA Dining website (menu-dining.ucla.edu)
  + Recipe detail pages for micronutrients & tags

### e. Deployment & Infrastructure

* Google Cloud Run
  + Hosts backend API in Dockerized Node.js Alpine container
  + Automatic scaling, managed SSL
* GitHub Actions CI/CD
  + Runs tests on PRs, builds Docker image, and deploys on merge to main
* Security
  + HTTPS enforced by App Engine SSL
  + Helmet sets CSP/HSTS headers
  + CORS restricted to known frontend origin

#### **i. System Architecture Diagram:**



## Requirement Satisfaction

1. Must use semantic HTML5 elements where appropriate (e.g., <header>, <nav>, <article>), and avoid overuse of generic tags like <div> and <span> unless justified and must use one of the following APIs:

a. Canvas, b. Geolocation, c. Drag and Drop, d. Use of camera and microphone.

* For requirement 1 we used appropriate HTML5 elements (such as section to divide our project into 3 distinct sections– title, best restaurants, and best foods). We used the Geolocation API to calculate the distance from the current user to set restaurant locations.

2. Must be aesthetically pleasing on screens as small as 320px . In other words, it must be responsive.

* Handwrote CSS to fulfill this requirement. Example: the .phone-sim container uses width: calc(100vh \* 0.563) and height: 100vh to mimic the aspect ratio of a typical phone screen. At the same time, we also had a centered layout for best functionality that also scales to larger screens.

3. The app must be available offline when there is no Internet connection. The app may not display any data, but the general layout should be available with perhaps a progress indicator showing the user "we are trying to fetch the data." (PWA)

* Progress indicator implemented when loading WASM modules and Geolocation. Also implemented the offline functionality with our service worker, where a service worker (service-worker.js) caches assets and displays a loading indicator when offline

4/5. Must use HTTPS. The app MUST be a single page application. No page scrolling horizontally or vertically.

* The app is a single-page application with a fixed viewport; all content fits within the screen without any horizontal or vertical scrolling. Set overflow: hidden and height: 100% on the html and body elements. React Router handles in-app navigation; all traffic is over HTTPS

6. It is more important that your app satisfies the requirements than look "beautiful" as this is not art class, but please do not hesitate to make it look nice.

* I think it looks quite OK :)

7. Should use a production CSS processor like Tailwind or Sass, but this is not required. If you wish to write the CSS yourself, this is fine.

* Handwrote CSS in NutritionDashboard.css

8. Must authenticate users using cookies with JWT, SSO or two-factor authentication. If you use cookies at all (you probably will have to), you must present a banner at the bottom of the page informing users of their rights.

* Secure, HTTP-only JWT cookies for sessions; CookieBanner component prompts consent

9. Must be impervious to the most common web security vulnerabilities (e.g. SQL injection, CSRF, XSS). There are technologies that handle this for you.

* Helmet (CSP, HSTS), CORS, Joi validation, and Mongoose schemas guard against XSS, CSRF, and injections

10. Must use a caching layer and a database. The database does not necessarily have to be MongoDB or relational (e.g. MySQL, PostgreSQL) but must make sense for your project. The database work must use an ORM or ODM.

* MongoDB Atlas with Mongoose ODM for persistence; Redis Cloud for menu and recommendation caching

11. Should use Node.js and Express; however, if you wish to use something else, please include your plans in the spec

* Backend on Node.js 18 + Express 4.18 in TypeScript (Controller→Service)

12. The app must function as a PWA with a service worker. If data is unavailable offline, the app should still render its structure and display a message or progress indicator (e.g., "Attempting to fetch data..."). The app must be installable on desktop and mobile

* Accomplished [service-worker.js](http://service-worker.js) file that handles offline actions.

13. [Extra Credit] Must use a WebAssembly module. a. Strong preference for AssemblyScript for Typescript but can use other languages supported by WebAssembly.

* A WebAssembly module was used to implement the Haversine formula for calculating distance between coordinates, which was then used to estimate calories burned from walking.

14. Must create/use an API to (pick one of these three): a. Expose data to an end-user (e.g. providing cafe menus to outsiders)

* Exposed JSON endpoints for recommendations (POST /api/recommendations), menus (GET /api/menu/today), and scraper health (/api/scraper/\*)

15. Must use a significant front-end framework such as React, Angular, Vue,js, Svelte, SolidJS, Quik etc.

* Used React 18 as the front-end framework for the project.

16. Must comply with basic accessibility principles (e.g., color contrast, tab navigation, semantic HTML). ARIA attributes should be used where applicable.

* Respected light/dark mode user preference, no tab navigation, had semantic HTML, and used ARIA attributes to ignore emojis.

17. Must be deployed onto Google Cloud: Google App Engine (single and pairs) or Google Kubernetes Engine (groups of 3 or 4) and use CI/CD principles via Github Actions. a. We may allow other deployment strategies as long as they provide scalability and fault tolerance. You must describe it in your proposal. b. The deployment must be triggered via CI/CD (not manual kubectl apply). You must include GitHub Actions logs showing build, test, and deploy stages.

* Deployed on Google App Engine with automatic scaling; GitHub Actions runs build, test, and automatic deployment on merges to main with no manual steps

## CI/CD Log and Build Triggers

We use Github Actions as a build trigger for CI/CD – upon a successful merge with main, the github workflow we set up has 3 build triggers:

* Automatic: Push to `main` branch triggers full deployment pipeline
* Pull Requests: Automated testing without deployment
* Manual: On-demand deployment via workflow\_dispatch

Build Stage (5-7 minutes)

1. Backend: TypeScript compilation, dependency installation, unit tests

2. Frontend: React build, WebAssembly compilation, asset optimization

3. Integration: Frontend bundled into backend/public directory

Test Stage

- Backend: Jest unit tests with 80%+ coverage requirement

- Frontend: React component tests

- Build verification: TypeScript type checking

Deploy Stage (3-5 minutes)

- Platform: Google App Engine (Standard Environment)

- GCP Project ID: cs144-webapps-ryanrosario-sp25

- Scaling: Automatic (0-10 instances)

GitHub Actions Logs:

