Verification & Validation Plan

Sophros

A Health Planning App

ECE 49595SD – Software Section – Fall 2025

Team 2

Emerson Maddock, emaddock@purdue.edu

Evan Stonehurst, estonest@purdue.edu

Janis Mikits, jmikits@purdue.edu

Eduard Tanase, etanase@purdue.edu

Table of Contents

1. Objectives 3
2. Prioritization 3
3. Requirements Verification and Traceability Matrix (RVTM) 4
4. Test Approach 5
5. Validation Plan 5
6. Tools, Environments & Test Data Sets 7

1. Objectives

Objective 1: User Authentication - New accounts are updated in the Sophros database, and users can log in within 5 minutes after signing up.

Objective 2: Recipe API Connection - Verify that at least 95% of API calls return valid recipes that match the given parameters.

Objective 3: LLM Integration – At least 90% of responses about recipe creation are contextually accurate, readable, and match intended functions.

Objective 4: Data cache – Verify that API calls retrieve at least 90% of data from the cache instead of an API call.

Objective 5: Nutrition API Connection – At least 95% of responses about food queries from the USDA FoodData Central API return valid nutrition data including macronutrient values.

2. Prioritization

Product Capability	Priority (Low, Medium, High)	Brief Justification
User Authentication	High	Integral to accessing the app, the first step in user experience, and is essential for profiling a user
Recipe API Connection	Med	Method for gathering recipes, important to the meal plan portion of our app
LLM Integration	Med	LLM is used to process user meal descriptions and make recipe suggestions, quite important
Data cache	Low	Drawing from cache is important but only relevant for optimization / saving costs at later points in development
Nutrition API Connection	High	Performing nutrition calculations is foundational to our forecasting model and recipe suggestion

3. Requirements Verification and Traceability Matrix (RVTM)

Note: Req. stands for Requirement

Product Capability	Req ID	Requirement	Assumptions & Context	Test ID	Test Description	Impacted Components
User Profile	FR-1	The system shall allow a user to input biological data including gender, age, weight, height, and physical activity.	This requirement assumes the user already has an account created. Detailed information is defined as the data used by the USDA for DRI (Dietary Reference Intake) calculations.	TR-1	Have 10 users complete go through a full user-flow of creating a profile and updating it, track relevant metrics such as percentage of data recorded, ease of use, ability to update, etc.	User Database
User Authentication	FR-2	The system shall allow a user to create an account or login via email or an OAuth provider through a Clerk interface.	This requirement assumes Clerk is running at time of testing.	TR-2	Use GUI auto interaction to test creating a new account from scratch. System test that this gets updated in our database and that users can log in again after sign up	UI, Database
Schedule forecaster	FR-3	The system shall provide the user with a personalized schedule of meals, workouts, and sleep based on the user's biological data, goals, and preferences.	Forecasting is defined as AI-driven generation of weekly plans integrating nutritional and scheduling data.	TR-3	Provide 20 diverse user profiles with different goals and schedules. Verify each forecast includes meal, exercise, and rest recommendations consistent with a nutritionist's suggestions for the user. Component top-level test	Algorithm
Personalized insights	FR-4	The system shall generate adaptive insights based on the user's tracked performance and historical data to recommend behavior adjustments.	Insights refer to auto-generated summaries and recommendations informed by stored biological and behavioral data.	TR-4	Run 10 test users for a simulated two-week period. Confirm that each user receives at least one adaptive recommendation aligned with tracked data changes.	Everything
Progress tracking system (gamified UI)	FR-5	The system shall display user progress through gamified elements such	Gamified progress tracking includes elements like streak counters, badge unlocks, and leaderboards.	TR-5	Simulate user activity logs over a 4-week period and verify that badges and streak counters update	UI, Database

		as streaks, badges, and leaderboards based on activity adherence.			correctly 95% of the time.	
Recipe API Connection	FR-6	The system shall connect to a recipe database API to retrieve meals that meet users' macronutrient and dietary preferences.	"Valid recipe" means one that includes full nutritional data and ingredient lists compatible with USDA nutritional models.	TR-6	Query the API 100 times with different nutritional constraints. Requirement passes if 95% return valid recipes matching parameters.	Algorithm

4. Test Approach

For unit testing, we will write unit tests for individual backend components in FastAPI (route handlers, services, data access layers, and utility functions) based on each component's defined task. These tests will be fully automated using pytest and run in CI on every commit or pull request. For integration testing, we will verify the interactions between our application and our user database, authentication provider (Clerk), and external API's (USDA API, recipe APIs). These tests will be mostly automated in a Python test suite, but we will perform manual exploratory tests when endpoints or workflows change significantly. For system testing, we will focus on end-to-end user flows. We will define tests mirroring the user flows in Cypress to test an entire flow. These tests will run in a staging environment that mirrors production. This will be automatic, although the test flows will have to be defined manually.

For testing performance, we will use a load testing tool such a Locust against the staging stack to ensure p95 latency is within our NFR's (<2 second latency) under normal load (50 concurrent users). For testing reliability, we will monitor uptime and error rates via AWS CloudWatch, Sentry, and Neon metrics. We will verify that we meet or exceed the 99.5% backend uptime requirement. For testing security, we will run periodic scans such as ZAP to confirm TLS 1.3 in transit and encryption at rest.

In terms of regression testing, we will set up GitHub Actions to automatically run all unit and integration tests on every push and pull request, preventing regressions in core logic or database interactions. We will maintain a regression test suite of high-priority system tests that we rerun at the end of each sprint or before a release. If a bug is found, we will add a test that reproduces it, fix the code, and keep the test in the testing suite so the issue cannot reappear.

5. Validation Plan

For the User Acceptance Testing (UAT) approach, during each major project milestone, we will download a beta version of the app and ask at least 5 people to perform tasks (like adding a

schedule item or selecting a meal) on the app to see their performance. If we see user friction with the feature, we will make tickets to fix the issues raised, as well as surveying their feedback for usefulness to their life. We will measure the time-to-action (i.e. the amount of time it takes for them to complete the given action), the number of misclicks, and survey the user frustration with the action pathway. We will also use the app for a week or until the next engineering drop (whichever comes first) to ensure that usability and user acceptance are improving over time. We will utilize the added features as well as the whole suite of implemented features. We will record our app usage time and time per open instance to understand if we are developing an easy-to-use, lightweight solution as intended. This sort of incremental improvement and validation of design will ensure continued development towards usability and user acceptance.

A core element of our UAT is prototype testing. Our procedure is relatively simple; we use the application, covering the core features over time and seeing how the system complements the users self-development journey. We will be the primary testers due to budgetary and time constraints, so we will begin by inputting our personal data into the system and making accounts. In order to better understand friction points, we will be sure to store debug information and meta information on a dev user profile so that we can trace the system execution better. With this framework, our prototype testing will be broken into 2 types: formal and informal testing. As developers, we will follow an informal testing procedure: we will use the application and note down issues, etc. as appropriate. For the formal testing procedure, we will follow steps as follows:

1. Launch the application on the target platform with a screen recording enabled to capture bugs and friction points for review.
2. Make a new account, ensuring that it is marked as a development account.
3. Perform the setup to make an account in a specific state – this could be automated with internal scripts
4. Perform the actions needed to test a certain feature.
5. Record results.

We will record the time it takes to perform the tested action and the rate of error for the feature, along with qualitative user feedback on the feature. Combining a small amount of formal testing (ideally with new users who haven't been involved in the development process) with a large amount of informal testing will ensure that our prototype is well tested.

6. Tools, Environments & Test Data Sets

We will use a combination of automated testing tools, environments, and datasets to support verification and validation.

- Backend and unit/integration testing:
  - Pytest/unittest: Our primary frameworks for unit and integration testing on the FastAPI backend
  - FastAPI TestClient: An API in FastAPI to simulate HTTP requests to endpoints without deploying
- API/system testing:
  - Postman: A tool for manual and automated API testing
  - Locust: For performance and load testing for latency/throughput
- UI/E2E Testing:
  - Selenium: For scripted UI flows in a near-real environment
- Security/Quality:
  - ZAP (OWASP): For security scanning of API endpoints and authentication flows
  - GitHub Actions: CI/CD pipeline for our test suites and regression testing
- Environments:
  - Local development environment: Run by each developer for development and occasional unit/integration testing
  - Staging environment: A deployed copy of our production environment for integration, system, and performance tests. This will be deployed using the same tools as the production environment and a staging branch of our database.
  - Production environment: A deployed version of our application, with strict access controls. This will not be used for testing.
- Data sets:
  - We will generate synthetic user profiles (age, sex, weight, height, activity level, etc.) to cover edge cases and typical behaviors, stored in our repository.
  - We will save sample responses from recipe and nutrition API's for deterministic testing.
- Version control and test management:
  - All code, tests, and test data will live in our GitHub repository. Tests will be organized by level (ex. /tests/unit, /tests/integration)
  - GitHub issues will track tests, test failures, and regressions. Each major test is mapped to one or more automated test files or scripts
  - GitHub Actions will run a fully automated test suite on every push and pull request and store artifacts such as coverage reports. For performance and security tests, we will store summary reports as CI artifacts. Manual test results will be captured in a document in a /docs/validation folder.