**Software Requirements Specification (SRS)**

**Project:** Australian Dietary Supplements (ADS) – web application  
**Version:** 1.4  
**Date:** 3 October 2025

**1. Introduction**

**1.1 Purpose**

This document presents the Software Requirements Specification (SRS) for the Australian Dietary Supplements (ADS) project. The purpose of this document is to define the objectives, scope, functional and non-functional requirements, and constraints of the system. It provides a reference for stakeholders and serves as a development guideline throughout the project lifecycle.

**1.2 Scope**

The ADS application is a web-based platform designed to help consumers compare dietary supplements and wellbeing products across multiple Australian retailers. It delivers four primary functions:

|  |  |  |
| --- | --- | --- |
| **Reference** | **High-level** | **Detailed** |
| 1 | Price comparison | Aggregates product and price data (vitamins, supplements, over-the-counter medication, etc.) from major retailers to compare prices, reviews, and product information. |
| 2 | Save product | Enables users to save searched items into a personalised list for later review or to monitor price changes over time. |
| 3 | AI chatbot | Provides an interactive chatbot to answer user queries about products, categories, and general information. |
| 4 | Direct user to retailer | Allows users to navigate directly to the cheapest available retailer’s product page for purchase. |

The system architecture includes:

* Database: MongoDB deployed in a Docker container for data storage and persistence.
* Backend: Node.js/Express REST API to serve application functionality.
* Frontend: Materialize CSS-based user interface for search, browsing, My List, and chatbot interaction.

**1.3 Definitions, acronyms, and abbreviations**

|  |  |
| --- | --- |
| **Term** | **Definition** |
| ADS | Australian Dietary Supplements (project name) |
| API | Application programming interface |
| ETL | Extract, transform, and load process |
| MVP | Minimum viable product |
| MVC | Model-View-Controller |
| UI | User interface |

**2. Overall description**

**2.1 Product perspective**

The ADS system is a new, standalone web application. It includes:

* Scraping modules to extract and transform data from retailer websites.
* Database (MongoDB via Docker) for secure, persistent data storage and access.
* Web API (Node.js/Express) for communication between the database and frontend.
* Frontend (Materialize CSS) for user interaction and display.

**2.2 Product functions**

The system provides the following core functions:

* Searchable product listings with price and details.
* Personalised "My List" feature for saved items.
* Links directing users to retailer product pages for purchase.
* Chatbot-based product information and guidance.

**2.3 User characteristics**

* Primary users: general consumers seeking to compare supplement prices and receive basic information.
* Users are expected to have basic computer and web navigation skills.

**2.4 Constraints**

* Timeframe: ~5 weeks of development, split into two sprints. Sprint 1 emphasises project setup and core functionality; Sprint 2 focuses on advanced features and UI/UX refinement.
* Technology: limited to MongoDB, Node.js/Express, Materialize CSS, and Python (mostly consistent with coursework scope).
* Deployment: The application requires Docker Desktop to run the database container.
* Persistence: data is retained using Docker volumes; deleting volumes causes data loss without backups.
* Limitations: limited scraped sources reduce the strength of cross-retailer comparison.

**2.5 Assumptions and dependencies**

* Internet connectivity is required for scraping and external retailer access.
* MongoDB container remains active and accessible to the web application.
* Retailer websites will allow scraping and remain available.
* Project backlog is managed using Trello (<https://trello.com/b/hRBjFHW9/sit725-group-project-australian-diatery-supplements-website>).

**3. Specific requirements**

**3.1 Functional requirements**

|  |  |
| --- | --- |
| **Reference** | **Detail** |
| FR1 | System shall deploy a MongoDB server in a Docker container, with persistent data stored in a Docker volume accessible via internal Docker networking. |
| FR2 | System shall store product details (name, description, rating, price history). |
| FR3 | System shall scrape product data from at least two Australian retailers. |
| FR4 | System shall provide a product search function. |
| FR5 | System shall display comparison views across retailers. |
| FR6 | System shall allow users to save items to their personal “My List”. |
| FR7 | Chatbot shall answer queries about products and comparisons. |
| FR8 | System shall provide links to retailer product pages, prioritising the cheapest option for purchase. |

**3.2 Non-functional requirements**

|  |  |
| --- | --- |
| **Reference** | **Detail** |
| NFR1 | Performance: API responses within 300 ms for typical queries. |
| NFR2 | Reliability: scraper jobs succeed ≥ 90% of attempts. |
| NFR3 | Usability: UI must be responsive on desktop. |
| NFR4 | Persistence: data retained via Docker volumes across restarts. |

**3.3 Use cases**

|  |  |  |
| --- | --- | --- |
| **Reference** | **Use case** | **Description** |
| UC1 | Account creation/login | User can create an account and sign in. |
| UC2 | Product search | User searches for a product and views results. |
| UC3 | View details | User clicks “VIEW DETAILS” to access product information. |
| UC4 | Save to My List | User saves items to “My List” for later. |
| UC5 | Purchase link | User clicks “PURCHASE” to be directed to the retailer site. |
| UC6 | Chatbot query | User asks chatbot about a product; chatbot responds logically. |
| UC6 | Logout | User logs out and system requests confirmation. |

**3.4 Data requirements**

* Products stored in MongoDB with retailer references.
* Persistent storage via Docker volumes.
* Backup/Restore supported through mongodump and mongorestore.

**3.5 Testing**

**3.5.1 Automated Testing**

* Unit tests: individual functions and data validation.
* Integration tests: API endpoints and module interactions.
* UI/UX tests: de-duplication, pagination, formatting, labelling, and truncation.
* Simulation tests: error handling and user flow resilience.
* Performance tests: response time and scalability under load.

**3.5.2 Manual testing**

* Performed during development and acceptance testing.
* Verify all features: search, comparison, chatbot, and scraping.
* Confirm error handling and input validation.
* Assess usability, responsiveness, and accessibility.

**3.5.3 Test cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reference** | **Description** | **Precondition** | **Steps** | **Expected result** |
| **TC1** | Search | Products exist with multiple brands and prices | Go to search →  Enter “Vitamin” →  Click search | Products containing “Vitamin” displayed (20 per page). |
| **TC2** | Product comparison | Product available from ≥ 2 retailers | Select product →  View prices | Retailer and prices displayed with “PURCHASE” link |
| **TC3** | Search de-duplication | Dataset with duplicates (\_id and name+img) | Access the container → Run npm test →  Execute dedupe logic | Items deduped; retailer data merged. |
| **TC4** | Pagination | Dataset ≥ 31 items | Access the container → Run npm test →  Paginate results | Page 1 = 20 items, last page = 11, requests beyond max clamp to last. |
| **TC5** | Retailer extraction | Dataset with multiple retailer entries | Access the container → Run npm test →  Call extraction helper | Price/URL merged, sorted ascending. |
| **TC6** | Review labels | Dataset with mixed/null review values | Access the container → Run npm test →  Pass review values | Consistent labels: “Not yet reviewed” or <n> reviews. |
| **TC7** | Title truncation | Long/short titles | Access the container → Run npm test →  Call truncate() | Short titles unchanged; long titles end with ellipsis, cut at word boundary. |
| **TC8** | Number of items in the database | Ensure the number of items in the database is > 15000 | Access the container → Run npm test → DB test | The minimum number of items in the database is 15000 |
| **TC9** | Always ensure certain collections are in databases | Default collections when initializing the database | Access the container → Run npm test → DB test | Collections 'Items' and 'Items\_li' are always present |
| **TC10** | Minimum information of an item | All items must have certain properties. | Access the container → Run npm test → DB test | Attributes like item name, price, image path, and item ID must always appear |
| **TC11** | Similar items | Similar products must share the same categories and then be sorted by brand overlap. | Access the container → Run npm test → Item test | Similar products with id '2728073' must have category 'medicines' and have the same brand name 'Panadol' |
| **TC12** | Search item by ID | URL: "items?id=<item\_id>" must return information about the exact item | Access the container → Run npm test → Item test | Item ID '2728073' must return the product as 'Panadol Rapid Paracetamol Pain Relief 48 Caplets' |
| **TC13** |  |  |  |  |
| **TC14** |  |  |  |  |
| **TC15** |  |  |  |  |

**3.6 UI/UX requirements**

The interface shall be clean, consistent, and accessible. Materialize CSS provides styling, with emphasis on clarity and readability.

**3.6.1 Design**

* Homepage: navbar (Home, Tips, Chatbot, Item detail, My List), search bar, quick links.
* Search results: grid view with price, rating, comparison, details, and save to My List option.
* Product detail: full product info and retailer comparison, with purchase links to retailers.
* My List page: saved items list with thumbnail, retailer, price, remove button, and quick “VIEW DETAILS” / “PURCHASE” actions.
* Chatbot panel: dedicated interface for queries, with conversation history based on each user.

**3.6.2 Core screens and components**

* Homepage / Search results / Item detail / My List / Chatbot.

**3.6.3 Interaction**

* Top navigation bar for quick access.
* Prominent search bar.
* Save-to-list buttons, purchase links, and My List.
* Chatbot panel for interaction. Add and delete conversations

**4. Lessons Learned**

The ADS project was developed within a highly compressed timeframe and by a team with differing levels of technical capability. Several important lessons were identified:

* Differing skill sets and capability gaps: team members entered the project with varied experience in coding, database design, and front-end development. This sometimes slowed progress as additional time was needed for peer support and knowledge sharing. However, it also encouraged stronger collaboration and cross-learning.
* Overscoping within a 5-week turnaround: the initial scope was ambitious, including multiple advanced features such as AI chatbot functionality, data analysis and visualisation, and administration user/functionality. Delivering all of this within a short academic sprint required prioritisation. Some features had to be simplified or partially implemented to ensure a working MVP.
* Code conflicts and version control challenges: with multiple contributors working on different modules, merge conflicts in GitHub occurred. This emphasised the importance of frequent commits, branch discipline, and clear communication. Over time, the team developed better practices for pull requests and conflict resolution.

These lessons highlight the value of careful scope management, alignment of tasks to individual strengths, and consistent collaboration practices. They will directly inform how the team approaches future projects with regard to planning, communication, and technical execution.

**5. Final Project Report**

The ADS project presented both challenges and successes. Midway through development, the team faced a setback when one member resigned from the project. This required rapid reallocation of workloads and rebalancing of responsibilities among the remaining members. As a result, some features were descoped to ensure delivery of a functional minimum viable product within the five-week timeframe.

Despite this challenge, the team maintained strong collaboration. Tasks were redistributed fairly; communication improved; and collective focus shifted toward essential features such as product comparison, My List and retailer links. While some advanced features had to be scaled back, the final product reflects a solid and usable application.

Key reflections include:

* The importance of adaptability when project circumstances change.
* The benefit of prioritising core features over “nice-to-haves” under tight deadlines.
* The resilience of the team when faced with unexpected setbacks.

Ultimately, the project delivered a working application that aligns with its stated objectives. More importantly, it served as a valuable learning experience in project management, teamwork, and technical development. Each team member left the project with stronger technical skills and a deeper appreciation of collaborative software engineering practices.

The team is confident that, with further development and time, the ADS application could be expanded to include additional data sources, stronger chatbot intelligence, and richer user profiles.

**6. Supporting information**

**6.1 References**

* Chemist Warehouse and Chemist Outlet websites.
* Deakin University SIT725 coursework guidelines.
* Team’s Trello board for backlog and sprint planning (<https://trello.com/b/hRBjFHW9/sit725-group-project-australian-diatery-supplements-website)>.