

---

# **SOFTWARE REQUIREMENTS SPECIFICATION**

**For**

## **Rice Retail Inventory Management and POS System**

---

**Prepared by:**  
**Equilibrium Syntax**

**Jhon Cries A. Adlao**

**Arth A. Dablo**

**Nathaniel A. Piraman**

**Submitted in fulfillment requirements for Integrative  
Programming and Technologies 2 (IT305)**

**October 18, 2025**

# Table of Contents

<b>Table of Contents .....</b>	<b>2</b>
<b>1. Introduction.....</b>	
1.1 Purpose .....	
1.2 Intended Audience .....	
1.3 Product Scope.....	
1.4 Definitions, Acronyms, and Abbreviations .....	
<b>2. Overall Description.....</b>	
2.1 User Characteristics .....	2
2.2 Constraints .....	3
2.3 Assumptions and Dependencies .....	3
<b>3. Requirements Specifications.....</b>	<b>4</b>
3.1 Functional Requirements .....	4
3.2 Non-Functional Requirements.....	4
3.3 External Interface Requirements .....	6
3.4 System Models.....	6
3.5 Database Design.....	7
3.6 Implementation.....	7

## **1. Introduction**

The *Rice Retail Inventory Management and POS System* is designed to help rice sellers efficiently manage their inventory, sales, and customer transactions through a digital platform. It automates tasks such as stock monitoring, product listing, and order processing while ensuring accuracy in weighing and compliance with tax regulations. The system also allows buyers to browse products, place orders, and choose pickup or delivery options. Overall, it improves efficiency, reduces manual errors, and connects rice retailers and customers through a convenient and reliable online marketplace.

### **1.1 Purpose**

The Rice POS System is a web-based platform designed to streamline the buying and selling of rice in local markets. It enables buyers to browse, add to cart, and place orders; sellers to manage store information, list rice products, and fulfill orders; and administrators to monitor all transactions and maintain system integrity.

### **1.2 Intended Audience**

The primary users of the *Rice Retail Inventory Management and POS System* are rice store owners, store staff or cashiers, and regular customers. Store owners and staff use the system to manage inventory, record sales, and process customer transactions efficiently. Meanwhile, customers benefit from a more convenient shopping experience, allowing them to browse rice products, place orders, and receive accurate order updates through the platform.

### **1.3 Product Scope**

The system provides:

- User authentication (login, signup, logout)
- Product catalog with image upload
- Shopping cart and order placement with delivery/pickup options
- Order tracking (pending → completed)
- Seller store management
- Admin oversight (view/delete products, view completed sales)

**Boundaries:** Does not include payment processing, real-time chat, or multi-currency support.

### **1.4 Definitions, Acronyms, and Abbreviations**

<b>Term</b>	<b>Definition</b>
-------------	-------------------

- POS Point of Sale
- SPA Single Page Application
- CRUD Create, Read, Update, Delete
- API Application Programming Interface
- JSON JavaScript Object Notation
- 25kg Standard rice sack size used in pricing

## **2. Overall Description**

### **2.1 User Characteristics**

- Buyers (Customers): Can browse/search rice products, add to cart, choose pickup or delivery, and track orders. Basic computer or mobile device knowledge required.
- Sellers (Store Owners): Can register/login, create store profiles, add/manage products, configure fulfillment options (Pickup Only or Delivery), confirm orders, and update order statuses. Requires moderate knowledge of online selling platforms.
- System Administrator (Optional Role): Manages platform-wide settings, ensures system reliability, and supports both buyers and sellers.

### **2.2 Constraints**

- Must comply with local taxation and digital receipt issuance laws.
- System must support integration with weighing scales, and receipt printers (for sellers with physical stores).
- Data precision requirement: weights up to three decimal places.
- Must run on modern web browsers and mobile devices (Android/iOS).

### **2.3 Assumptions and Dependencies**

- Rice is sold in both packaged form (bags of varying sizes) and by weight.
- The system operates in a single base currency.
- Internet connectivity may not always be available; sellers must have offline POS functionality with automatic sync.
- Delivery availability depends on the seller's store settings (Pickup Only or Pickup + Delivery).
- External accounting system integration may be required for advanced reporting.

### **3. Specific Requirements**

#### **User Requirements**

##### **Buyers (Customers)**

- Create an account and log in securely.
- Browse/search rice products across different sellers.
- Add products to the cart and proceed to checkout.
- Choose order fulfillment option: Pickup or Delivery (if available).
- Make payments through multiple methods (cash on delivery, card, e-wallets).
- Track order status until completion.

##### **Sellers (Store Owners)**

- Register/Login as sellers.
- Create a store profile (business name, address, contact info).
- Add/manage rice products (type, grade, packaging, price).
- Choose fulfillment capability: Pickup Only or Pickup + Delivery.
- Receive and confirm orders from buyers.
- Update order status (Pending → Preparing → Ready for Pickup / Out for Delivery → Completed).
- View sales history and track completed transactions.

### **3.1 Functional Requirements**

#### **3.1.1 Authentication & User Management**

**User Signup:** The system shall allow a new user to create an account by providing a unique username, a valid email, and a password. The user must select a role (Buyer or Seller). The system shall validate that the password and confirm password fields match.

**User Login:** The system shall allow registered users (Buyers, Sellers, Admins) to log in using their username/email and password. The system shall redirect the user to their respective dashboard upon successful authentication.

**Password Visibility Toggle:** The system shall provide an icon on password fields to allow users to toggle the visibility of the password characters.

**User Logout:** The system shall allow a logged-in user to log out, which will terminate their session and redirect them to the public landing page.

### **3.1.2 Buyer Functions**

**View Products:** A buyer shall be able to view a grid of all available rice products from all sellers. Each product card shall display its image, name, price, and the seller's name.

**View Product Details:** A buyer shall be able to click on a product to view a detailed modal containing more information, including the seller's full store details and product description.

**Add to Cart:** A buyer shall be able to add products to a shopping cart.

**View and Manage Cart:** A buyer shall be able to view the contents of their cart, see the total price, and remove items from the cart.

**Place Order:** A buyer shall be able to check out from their cart by providing a delivery address (City and Purok). Upon checkout, a new order with a 'pending' status shall be created.

**View Order History:** A buyer shall have a dedicated section on their dashboard to view their pending orders and their order history (completed and cancelled orders).

Order & Fulfillment Management

### **3.1.3 Seller Functions**

**Manage Store Information:** A seller shall be able to add and update their store information, including Store Name, Owner Name, Contact Number, Email, and a short description.

**Add New Product:** A seller shall be able to add a new rice product to their inventory by providing a name, price (per 25kg and per kilo), description, delivery/pickup option, and an image.

**View Own Products:** A seller shall be able to see a list of all the products they have added to the system.

**View and Process Orders:** A seller's dashboard shall display a list of pending orders containing their products. The seller shall be able to mark a pending order as "Done" (completed).

**View Completed Orders:** A seller shall be able to view a history of their fulfilled/completed orders.

### **3.1.4 Administrator Functions**

**View All Stores:** An administrator shall be able to view a list of all registered seller stores and their information.

**View All Products:** An administrator shall be able to view all products listed on the platform from all sellers.

**Delete Products:** An administrator shall have the ability to remove any product from the system.

**View All Completed Orders:** An administrator shall be able to view a comprehensive history of all completed orders across the entire system.

### **3.2 Non-functional Requirements**

Performance: Page load time  $\leq$  2 seconds on 3G

Reliability: System uptime  $\geq$  99%

Security: Passwords must be hashed using `password_hash()` in production

Usability: Intuitive UI with modals, responsive on mobile ( $\geq 320\text{px}$ )

Scalability: Support up to 100 concurrent users

Maintainability: Code is modular, commented, and follows naming conventions

File Security: Validate image type (JPG/PNG/GIF), size  $\leq$  5MB

Data Integrity: Use transactions for order placement

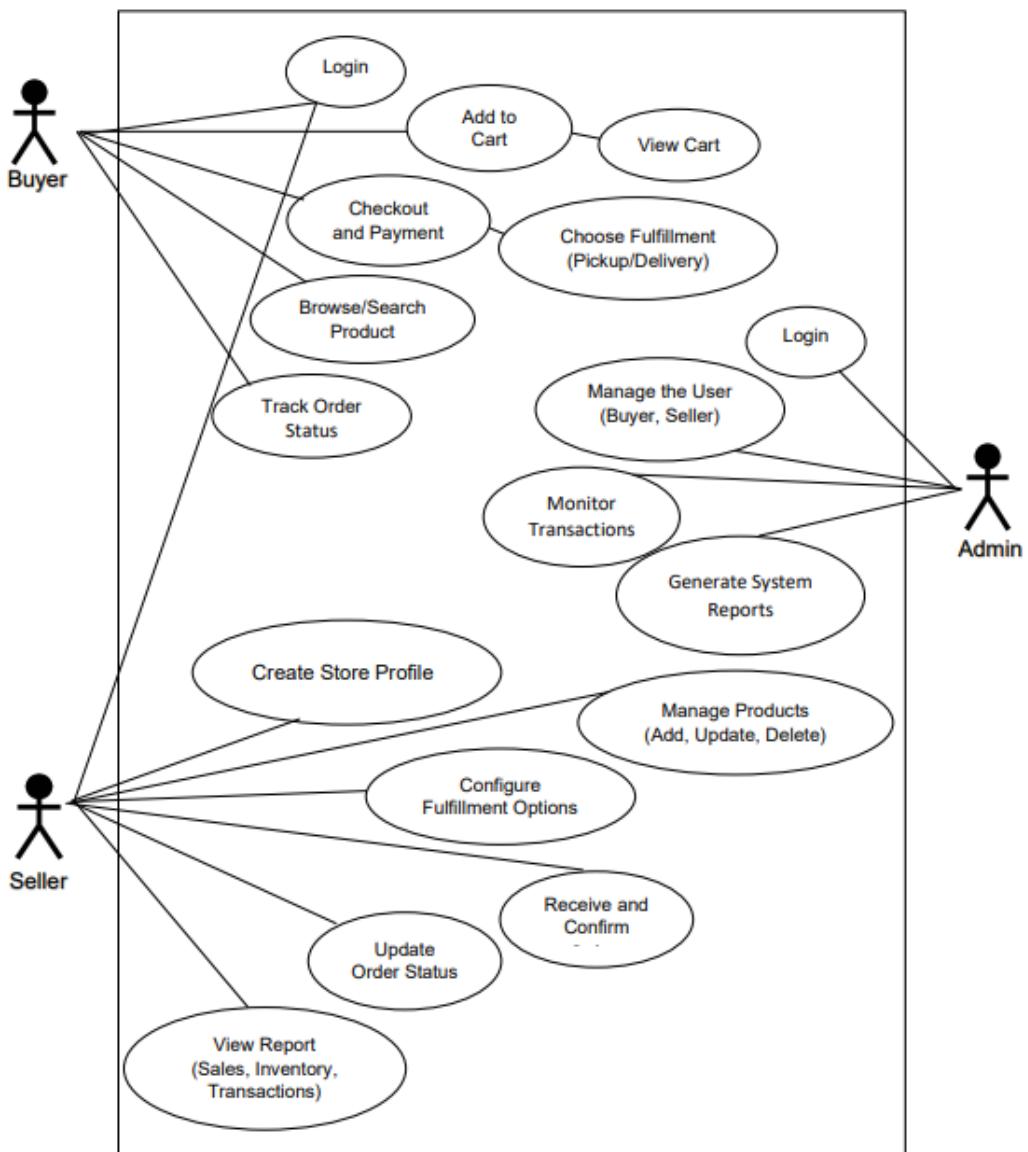
### **3.3 External Interface Requirements**

**User Interface:** The application presents a graphical user interface (GUI) through a web browser. The design is clean, modern, and utilizes a consistent color scheme.

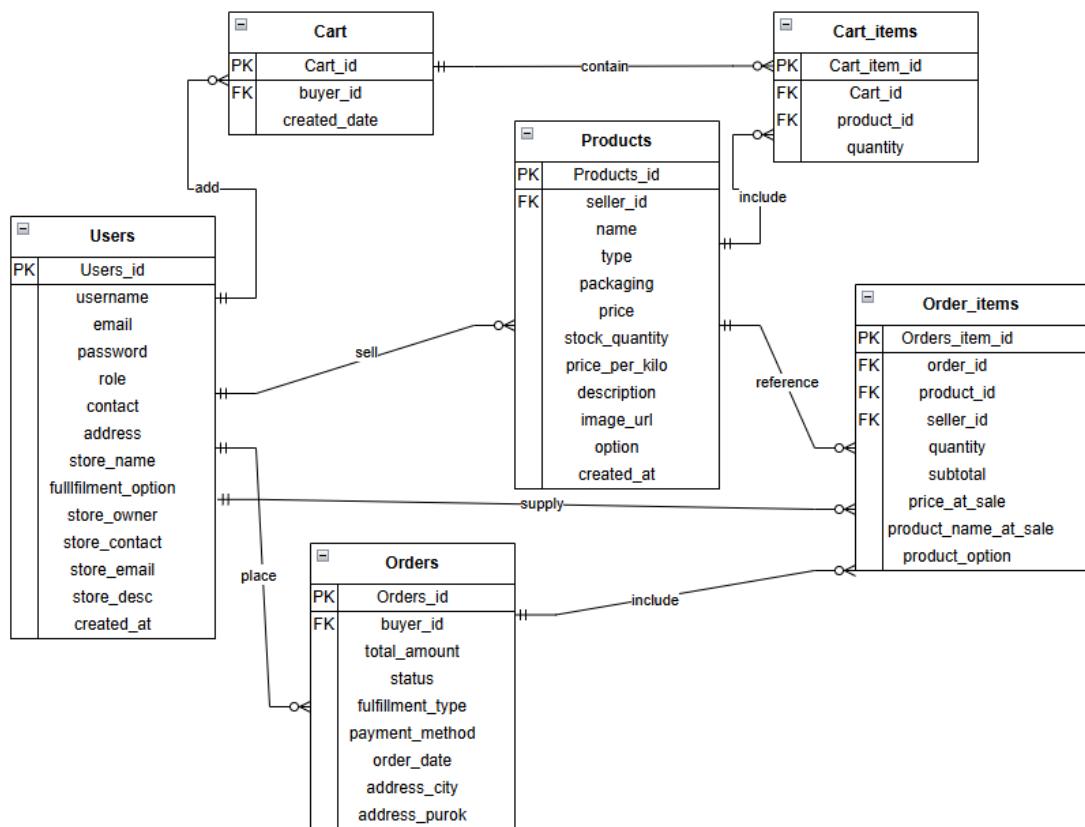
**Software Interface:** The frontend communicates with the backend via a RESTful API built in PHP. Data is exchanged in JSON format. The backend interfaces with a MySQL database to persist data.

**Hardware Interface:** The system requires no specialized hardware. It is accessible on any device with a modern web browser and internet connectivity, including desktops, laptops, tablets, and smartphones.

### 3.4 System Models



### 3.5 Database Design



### **3.6 Implementation**

**Frontend:** The client-side is built with standard HTML, CSS, and JavaScript. The code is structured into modular JavaScript files (api.js, auth.js, buyer.js, etc.) to separate concerns and improve maintainability. Asynchronous communication with the backend is handled via the fetch API.

**Backend:** The server-side API is built with PHP. It follows a modular approach where each endpoint (e.g., login.php, product\_add.php) is a separate file responsible for a single action. The backend is stateless and communicates exclusively through JSON.

**Database:** A MySQL relational database is used for all data persistence. The schema is normalized to ensure data integrity.

**Architecture:** The system follows a classic Client-Server architecture. The browser (client) makes requests to the PHP backend (server), which then processes the request, interacts with the MySQL database, and returns a JSON response.

**Tools:** VS Code, phpMyAdmin, Browser DevTools.

## Load Test Results

System context: The **Rice Retail Inventory Management and POS System** is a web-based platform designed to streamline the buying and selling of rice. The system allows **Buyers (Customers)** to browse and search for rice products across different sellers. This load test specifically simulates concurrent user activity on the **product retrieval** functions: **View Products** and **Browse Product**.

**Table 1: Load Test Performance Metrics**

Concurrency Level(C)	Average Response Time (milliseconds)	System Performance Observation
50 users	5081	<b>Stable but Slow.</b> No requests failed ( <b>0 Failed requests</b> ), confirming the system is <b>stable</b> at C=50.
100 users	12295	<b>High Load Stability Confirmed.</b> Only <b>1 Failed request</b> (0.02% failure rate), indicating the system is fundamentally stable at C=100.
125 users	12,186	<b>Load Test Failure.</b> <b>21.74% of requests failed</b> (1087 out of 5000). The system is crashing due to overload.
150 users	12,031	<b>Critical Failure.</b> 35.88% of requests failed.

The load test reveals a clear and concerning trend in system performance under increasing load. At a low concurrency of **50 users**, the system demonstrates **stability** with no failed requests, but the **Average Response Time (ART) is already slow at 5,081 milliseconds** (5.1 seconds), indicating an inherent performance bottleneck. As the load increases to **100 users**, the system maintains its **stability** (only 1 failed request, 0.02% failure rate), but the ART **more than doubles to 12,295 milliseconds** (12.3 seconds). This severe degradation suggests that the system is experiencing significant **resource contention** and is not scalable, rendering the user experience unusable. Critically, the system reaches its breaking point between 100

and 125 users; at **125 users**, the load test **fails**, with **21.74% of requests failing**, confirming the system is crashing due to overload. This capacity failure worsens at **150 users**, resulting in a **critical failure rate of 35.88%**. The data strongly suggests that the system's operational capacity is below 125 concurrent users, and fundamental architectural or code optimizations are required to address the high baseline latency and poor performance scaling.

## Stress Test Results

System Context: The **Rice Retail Inventory Management and POS System** is a web-based platform designed to manage inventory and sales for rice sellers. The system allows **Buyers (Customers)** to add products to a shopping cart, and subsequently view and manage the cart contents. The stress test evaluates the system's behavior and breaking point when the application is subjected to excessive concurrent load on the **Cart Retrieval** functionality, simulating a worst-case scenario where many Buyers are simultaneously accessing and modifying their shopping carts during peak sales.

**Table 1: Stress Test Performance and Breaking Point Analysis**

Database Load	Concurrent Users & Actions	System Response	Observation & Failure Mode
50,000 records	250 users <ul style="list-style-type: none"><li>• 5000 requests</li></ul>	Response Time: 0.526 seconds	Avg response time is acceptable, but the <b>99% percentile hit 5seconds</b> , and <b>1.04% failed</b> .
50,000 records	500 users <ul style="list-style-type: none"><li>• 5000 requests</li></ul>	Response Time: 1.558 seconds	Avg response time is 1.55 seconds. <b>3.64% failed</b> . Mass bottleneck under high contention.
50,000 records	750 users <ul style="list-style-type: none"><li>• 7500 requests</li></ul>	Response Time: 2.121 seconds	Avg response time exceeds 2 seconds. <b>4% failed</b> . Unacceptable load level.

50,000 records	1000 users <ul style="list-style-type: none"><li>• 10000 requests</li></ul>	Response Time: 2.785 seconds	Avg response time is 2.78 seconds. <b>4.26% failed</b> (426 out of 10000). System is unstable; maximum latency exceeded 27seconds.
50,000 records	10,000 users <ul style="list-style-type: none"><li>• 10,000 requests</li></ul>	Unresponsive	The operating system actively refused the connection instantly. This is the absolute breaking point.

The system performs well for **250 users**, showing a fast average response time of **0.526 seconds** and a low **1.04% failure rate**. However, this acceptable performance masks a vulnerability, as **1% of requests** suffer high latency, reaching **5 seconds**. When the load is doubled to **500 users**, performance degrades significantly: the average response time nearly triples to **1.558 seconds**, and the failure rate climbs to **3.64%**. This sharp decline is caused by a "**Mass bottleneck under high contention**," indicating that the system's capacity for handling simultaneous processes, likely in the database or transaction layer, is the core limiting factor.

The stress test on the **50,000-record database load** demonstrates that performance rapidly degrades beyond **250 concurrent users** and hits a critical "**Mass bottleneck under high contention**" at **500 users** (where failures reach 3.64%). Performance becomes **unacceptable** at **750 users** (4% failed, 2.121s response time), confirming the system cannot reliably support moderate user growth. The system's **absolute breaking point** is reached at **10,000 concurrent users**, where it completely refuses connections. The core problem lies in the system's concurrency handling, likely in the database layer.