# Project Requirements: Automated Warehouse Management

## 1 Project Overview

The goal of this project is to create a multi-threaded, concurrent simulation of an automated warehouse. The simulation will manage autonomous robots, charging stations, a parts inventory, and a queue of part requests, all running in parallel. The system must be resilient, manage shared resources in a thread-safe manner, handle errors gracefully using specific Java exception handling patterns, and provide a live-monitoring GUI.

## 2 Functional Requirements

### 2.1 FR-1: Core Simulation & System Components

- The simulation must be **multi-threaded**, with a central `ExecutorService` managing the lifecycle of all active components.
- A central **`Warehouse`** class shall contain and manage all other components and shared resources.
- **Storage Management System:** An **`Inventory`** class must manage a collection of `Part` objects and their quantities. It must support fast part lookups by `partID` string.
- **Task Management System:** A **`PartRequestManager`** must manage a thread-safe queue of incoming tasks (`PartRequest`).

### 2.2 FR-2: Entities & State Models

- **`Robot` (Runnable):** Must run on its own thread.
  - Must have a `batteryLevel` (0-100).
  - Must have a `RobotStatus` state machine (IDLE, WORKING, LOW_BATTERY, WAITING_FOR_CHARGE, CHARGING).
  - Battery must drain after `WORKING` and recharge during `CHARGING`.
  - Must independently pull tasks from the `PartRequestManager`.
- **`ChargingStation` (Runnable):** Must run on its own thread.
  - Must pull `Robots` from a shared, thread-safe `BlockingQueue`.
  - Must hold a `Robot` and call its `charge()` method until `isFullyCharged()` returns true.

- **PartRequest (Record):**
  - Must be an immutable record.
  - Must have a RequestStatus (PENDING, IN_PROGRESS, COMPLETED, FAILED).
  - Must provide a withStatus() method to create a new, updated copy.

### 2.3 FR-3: Data Exchange (I/O Streams)

- **Character Streams (Read):** The PartRequestManager must use BufferedReader to read and parse the pending_requests.txt file.
- **Character Streams (Write):** The LoggerUtil must use BufferedWriter to write all text-based log files.
- **Byte Streams (Write):** The Warehouse must use DataOutputStream to write a final binary report file (completed_report.dat) upon shutdown.

### 2.4 FR-4: Logging & Reporting

- **Event Logging:** A LoggerUtil class must be used to log timestamped events for all major components.
- **Granular Logs:** Logs must be generated for each individual robot (e.g., Robot-R-001.txt), charging station (e.g., ChargingStation-CS-A.txt), and system component (e.g., Warehouse-WH-01.txt).
- **Log Metadata Management:** The LoggerUtil must, upon initialization, find all previous logs of its type in the Logs/ directory and move them to an Logs/Archive/ directory.

### 2.5 FR-5: Graphical User Interface (GUI)

- The system must provide a **Swing-based GUI** (WarehouseSystemGUI) for control and monitoring.
- The GUI must allow the user to **configure** (robot/station count) and **Start/Stop** the simulation.
- The GUI must allow users to **manually add new tasks** to the PartRequestManager queue at runtime.
- The GUI must display **live-updating JTables** for:
  - All Robots (ID, Status, Task, Battery).
  - All ChargingStations (ID, Status, Charging Robot).
  - All Inventory items (ID, Name, Stock).
  - The active PartRequest queue (both PENDING and IN_PROGRESS tasks).
- All file I/O operations for the GUI's log viewer must be performed on a **background thread** (e.g., SwingWorker) to prevent freezing the UI.

# 3 Non-Functional Requirements

## 3.1 NFR-1: Concurrency & Thread Safety

- The system must use an `ExecutorService` to manage a thread pool for all `Runnable` components.
- Shared collections must be thread-safe:
  - **Inventory:** The main stock map must use `ConcurrentHashMap`.
  - **Task Queue:** The request queue must use `ConcurrentLinkedQueue`.
  - **Charging Queue:** The queue for robots awaiting a charge must be a `BlockingQueue` (e.g., `LinkedBlockingQueue`) to handle producer-consumer logic between robots and stations.
- **Atomicity:** Critical operations on shared resources (e.g., checking for and taking a task, checking and removing stock) must be atomic, using `synchronized` blocks.
- **Inter-thread Communication:**
  - The `PartRequestManager` must use `wait()` and `notifyAll()` to efficiently pause and resume `Robot` threads when the task queue is empty.
  - Fields read by the GUI thread but written by component threads (e.g., `Robot.status`, `Robot.batteryLevel`, `ChargingStation.currentRobot`) must be marked as `volatile` to ensure visibility.

## 3.2 NFR-2: Exception Handling

The project must correctly implement four specific exception handling patterns:

**a. Handling Multiple Exceptions:** A single `catch` block must be used to handle at least two distinct exception types.

- **Implementation:** `PartRequestManager.loadRequestsFromFile` uses `catch (IOException | NumberFormatException e)` to handle file-reading and parsing errors in one block.

**b. Re-throwing Exceptions (Business Error):** A method must throw a custom, checked exception to signal a specific business rule failure.

- **Implementation:** `Inventory.removeStock` throws `InsufficientStockException` if the requested quantity is not available. This is caught by the `Robot`'s `findAndSecureTask` method, which then fails the task.

**c. Resource Management (try-with-resources):** All file and I/O streams must be managed using the `try-with-resources` statement to prevent resource leaks.

- **Implementation:** Used in `LoggerUtil` (for `BufferedWriter`), `PartRequestManager` (for `BufferedReader`), and `Warehouse` (for `DataOutputStream`).

**d. Chaining Exceptions:** Low-level exceptions (e.g., `IOException`) must be caught and wrapped in a custom, high-level exception, preserving the original `cause`.

- **Implementation:** `PartRequestManager.loadRequestsFromFile` catches `(IOException | NumberFormatException e)` and throws `new RequestProcessingException("Error reading request file...", e)`.

### 3.3 NFR-3: Unit Testing

*(No unit test files were provided, but this would be the requirement)*

- **Test Coverage:** Unit tests should be created for each major class (e.g., `Robot`, `Inventory`, `PartRequestManager`, `ChargingStation`).
- **Test Suite:** All unit tests should be combined into a single test suite for regression testing.
- **Test Execution:** Tests must be run and results captured as proof of completion.