Final Project: Automated Warehouse Inventory, Verification, and Analytics System

Songgun Lee
Physics 129

December 9, 2025

1 Software Installation Instructions

This project relies on specific system-level libraries for the Graphical User Interface (GUI) and data visualization, but in my python files I have all the necessary imports such as Tkinter or Matplotlib. Also, the system may fail for python versions prior to 3.7 due to a datetime object that I use that was introduced in Python 3.7. All other libraries used (such as `json`, `os`, `datetime`) are also part of the standard Python 3 standard library and require no separate installation.

2 External Hardware

The following external hardware is required to operate the Warehouse Inventory System. This device serves as the primary input mechanism for the barcode-driven logic.

- **Item:** Symcode Wireless Barcode Scanner Versatile 2-in-1
- **Make/Model:** Symcode Handheld Automatic 1D Laser Barcode Scanner
- **Specifications:** Dual-mode connectivity (2.4GHz Wireless transmission up to 328ft + USB 2.0 Wired); Plug-and-Play driverless installation.
- **Place of Purchase:** Amazon
- **Purpose:** Functions as a Human Interface Device (HID) keyboard emulator. It scans 1D barcodes (such as EAN-13 on products and Code-128 on waybills) and inputs the decoded string directly into the active Python application window on the Raspberry Pi 5. The barcodes are also necessary, but you can generate them for free anywhere online (which I did) or just use existing databases.

2.1 Overview

The goal of this project was originally to construct a computer-vision-based Quality Control (QC) station for Dragon Diffusion leather goods using a Convolutional Neural Network (CNN). However, due to the hardware constraints of the Raspberry Pi for real-time neural network training and inference, the project scope was pivoted to a data-driven **Warehouse Inventory & Verification System**. The core input mechanism was shifted from computer vision to a USB Barcode Scanner to prioritize 100% process accuracy, system stability, and industrial reliability over AI vision, which by nature is probabilistic (and may be faulty). I also want to acknowledge that I adapted the core logic for the barcode interface and the GUI from pre-existing codes and tutorials found online. All these resources are listed in the **References** section. Honestly, this project was not "impossible level" challenging thanks to the abundance of helpful information provided on YouTube and the internet. These resources allowed me to focus on integrating the different parts rather than writing complex drivers from scratch.

2.2 Project File Structure

The project codebase consists of five Python modules. The primary executable(key program) is `warehouse_gui.py`, which serves as the main entry point. When executed, it initializes the Kiosk interface and orchestrates logic imported from the supporting modules (`inventory_stock.py`, `sales_analytics.py`, etc.). The supporting files must reside in the same directory for the import logic to function correctly.

- `warehouse_gui.py`: The primary executable of the project. It launches the full-screen Tkinter Kiosk interface, manages the real-time scanning event loop, and integrates the backend logic to provide visual Pass/Fail feedback to the user.
- `inventory_stock.py`: The backend logic handler for state persistence. It manages the reading and writing of JSON database files, handles stock deduction mathematics, and ensures that inventory levels are updated only after a successful order verification.
- `sales_analytics.py`: A data visualization module that parses the transaction logs. It utilizes the `matplotlib` library to generate high-resolution bar charts (Sales Velocity) and line graphs (Daily Throughput) for business intelligence reporting.
- `waybill_database.py`: A static dictionary module acting as the "Customer Database." It maps unique Waybill barcodes to specific customer orders, addresses, and required item lists, based on real data from *Maison de Celine*.
- `barcode_info.py`: A static dictionary module acting as the "Product Database." It maps individual item barcodes (SKUs) to their respective product names, colors, and categories to enable the scanner to identify physical items.
- `warehouse_scanner.py`: A lightweight Command Line Interface (CLI) version of the system. It serves as a fail-safe backup to the GUI, allowing the warehouse station to continue operating via text commands if the graphical environment fails.

Generated Data Files:

- `current_stock.json`: A persistent storage file that tracks the live inventory count for every item. It is automatically updated by the system every time an order is completed, ensuring stock levels are saved even if the Raspberry Pi is rebooted.
- `sales_history.json`: A cumulative log file that records the timestamp, order ID, and items sold for every completed transaction. This file serves as the raw dataset used by the analytics module to generate reports.

2.3 System Architecture

The final system is a full-stack logistics solution running on a Raspberry Pi 5. It integrates hardware input, a relational database structure, a graphical user interface, and backend analytics into a cohesive application.

- **Input Mechanism:** A USB Barcode Scanner acts as the primary interface, emulating keyboard input to feed data into the Python application.
- **Database Logic:** The system relies on two static database modules. `barcode_info.py` maps specific item barcodes to product details (Stock Keeping Unit, Color, Category), while `waybill_database.py` links shipping waybill numbers to specific customer orders. These databases were populated using real-world data samples from the South Korean company *Maison de Celine* to ensure realistic testing.
- **Graphical User Interface (GUI):** Built with `Tkinter`, the application runs in a full-screen "Kiosk Mode." It provides the packer with a dual-column display: one column listing the *Required Items* from the waybill, and a second interactive column tracking *Scanned Progress*. The UI offers immediate visual feedback (Green for Pass, Red for Error) to enforce validation without audio dependency.
- **Inventory Management:** The `inventory_stock.py` module handles persistent state. Upon the successful completion of an order, stock levels are automatically deducted from a JSON-based inventory file, and the transaction is logged to a sales history file.
- **Analytics:** To provide business intelligence, the `sales_analytics.py` module processes the transaction logs to generate visual plots (using `Matplotlib`) of sales velocity and daily throughput.

This system effectively simulates a production-grade warehouse station, preventing shipping errors by enforcing a digital "checklist" that must be satisfied before an order is marked complete.

3 Results

To demonstrate the system's functionality, the software was tested using the sample database provided by *Maison de Celine*. (Special thanks to our CEOs Mr. and Mrs. Kim, and my ex-manager Mr. Ha). The test procedure verified the end-to-end workflow: Waybill Scanning → Item Verification → Inventory Deduction → Analytics Generation.

3.1 Operational Workflow: One Example

I will be explaining an example action that would be made by this program. I run `warehouse_gui.py` via LXTerminal. Immediately after launching `warehouse_gui.py`, the system enters full-screen Kiosk mode. The following successful test sequence was performed:

1. **Initialization:** The system loaded the `current_stock.json` file, confirming 200 units for "Minkmore Rabbit Keyring Black".
2. **Waybill Scan:** Waybill `3372-6467-0943` was scanned. The GUI successfully retrieved the order for customer "Rachel Morrow" and populated the "Required" column with:
   - Mini Flat Gora Tan (x2)
   - Minkmore Rabbit Keyring Powder (x1)
3. **Item Verification:** The barcode for the "Mini Flat Gora Tan" was scanned. The system updated the interface with a green checkmark and incremented the counter. A subsequent scan of a wrong item ("Mini Flat Gora Black") triggered a red visual alert, preventing the packing error.
4. **Completion:** Once all items were verified, the system displayed a "COMPLETE" popup and automatically deducted the items from the persistent inventory file.

Video Demonstration: A full video recording of this operational process has been included in the submitted tar file for reference.

3.2 Analytics Output

The `sales_analytics.py` module was run to visualize the transaction history generated by the scanner. The program successfully parsed the JSON logs and produced plots.

Figure 1 illustrates the sales velocity for the last 30 days. The logic highlights the top three performing products in Gold, Silver, and Bronze colors respectively, providing immediate visual hierarchy for inventory managers. They will be the top 3 "winner" items, so the next time they order items from Dragon Diffusion they can order more "winner items". The y-axis must be only using integers, so I edited my code to only use integers for my y-axis, but it is not reflected on this image, yet it is reflected in the submitted file and video.

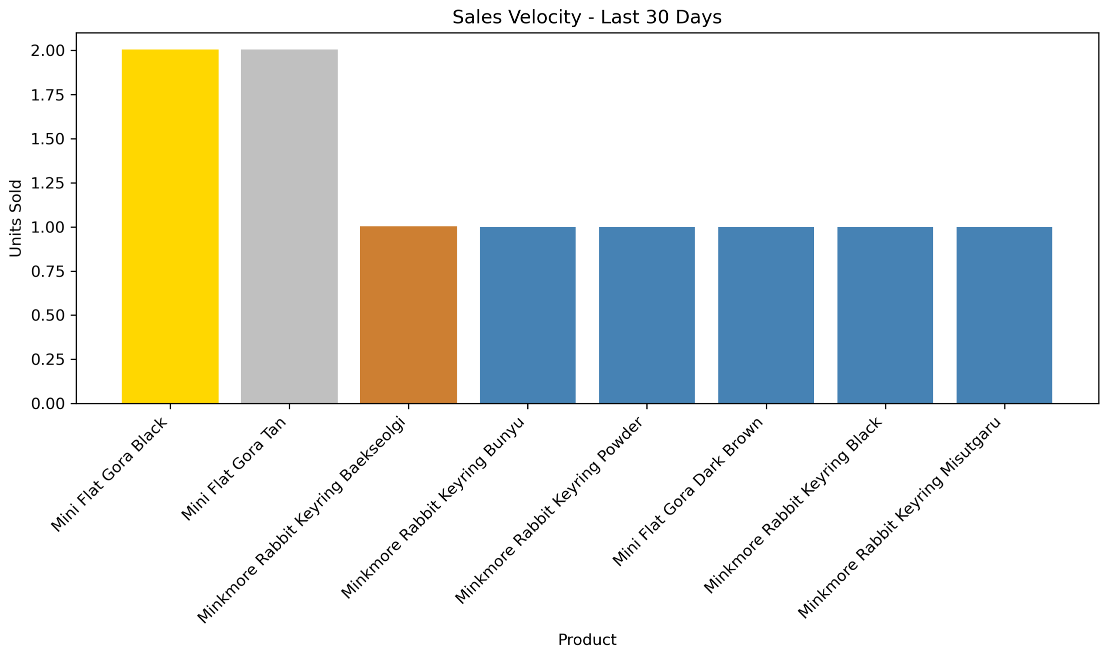

Figure 1: Sales Velocity Chart generated by Matplotlib, showing the distribution of item popularity over the last 30 days. The top 3 items are highlighted automatically.

Figure 2 shows the daily throughput. This plot utilizes the `datetime` library to aggregate timestamps from the sales history log, demonstrating the system's ability to track activity spikes.

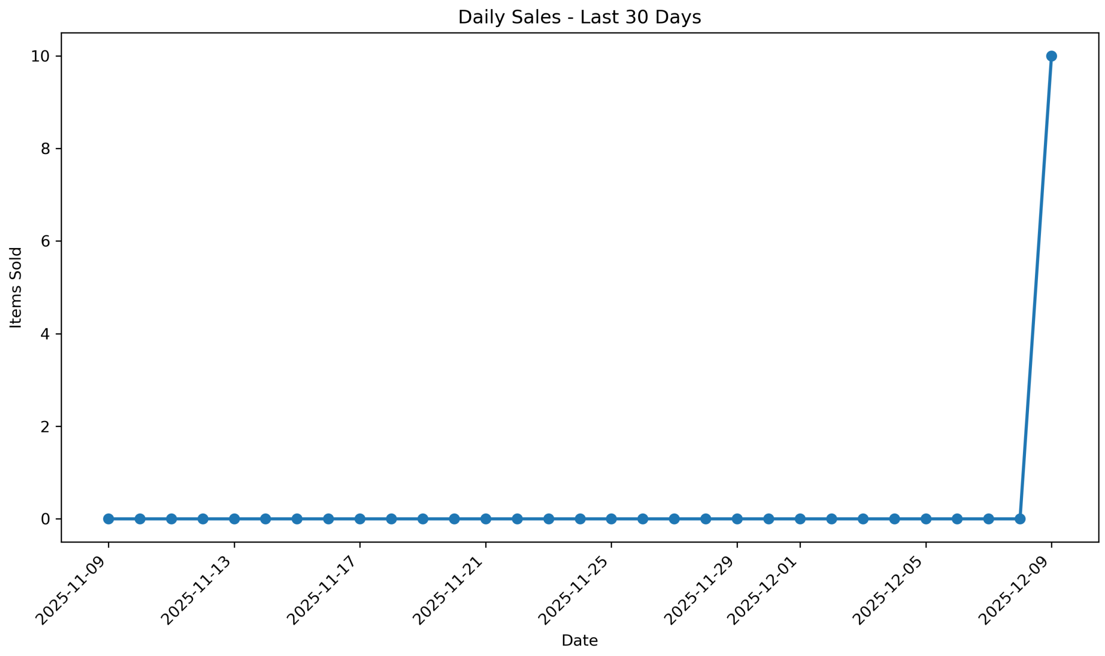

Figure 2: Daily Sales Volume line chart. The sharp increase at the end represents the batch of test orders processed during the final verification run.

The execution of these modules confirms that the system meets the core requirements of inventory verification, error prevention, and data persistence without reliance on unstable audio hardware (especially in places like warehouses) or heavy neural network processing.

4 References

The development of this system referenced several standard libraries and technical tutorials to implement the GUI and hardware integration.

- **Barcode Integration:** *Raspberry Pi Barcode Scanner with Python*, Learn Robotics (`https://youtu.be/BEsUpozvJCU`). Referenced for understanding HID input handling and `input()` stream logic.
- **GUI Design:** *Python Tkinter GUI Tutorial Playlist*, Codemy.com (`http://bit.ly/2UFLKgj`). Consulted for constructing the dual-column, kiosk-mode, general GUI instructions.
- **Data Visualization Logic:** *Active Product Sales Analysis using Matplotlib*, GeeksforGeeks (`https://www.geeksforgeeks.org/data-science/active-product-sales-analysis-using-matplotlib-in-python/`). Referenced for the methodology of aggregating sales data and the logic for visualizing comparative metrics, adapted to use standard Python structures for lightweight performance on the Raspberry Pi. I actually decided not to use numpy and decided to go for using standard Python lists because the dataset size for this project is relatively small (hundreds of records),

making the lightweight, built-in list structures faster and more memory-efficient for the Raspberry Pi than loading the heavy NumPy library. Maison de Celine has a couple hundred orders maximum per day.

- **Data Persistence:** *Inventory Management with JSON in Python*, TutorialsPoint (`https://www.tutorialspoint.com/inventory-management-with-json-in-python`). Provided the architectural basis for the `inventory_stock.py` module,and using json for inventory.

- **System Logic:** *An introduction to finite state machines and the state pattern for game development*, The Shaggy Dev (`https://youtu.be/-ZP2Xm-mY4E`). Used for finding how to make the RPI know if I am in the "waybill" barcode scan mode or "item" barcode scan mode.

- **Event-Driven Architecture:** *Events and Bindings in Tkinter*, GeeksforGeeks (`https://www.geeksforgeeks.org/python-binding-function-in-tkinter/`). Used for structuring the GUI to wait asynchronously for the "Return" key signal from the scanner (via `root.mainloop()`) rather than running a linear script that blocks user input.

- **Hash Map Optimization ($O(1)$):** *TimeComplexity*, Python Wiki (`https://wiki.python.org/moin/TimeComplexity`). Consulted to confirm that using Python dictionaries for the product and waybill databases provides $O(1)$ constant-time lookup performance, ensuring the Raspberry Pi processes scans instantly regardless of database size.

- **Separation of Concerns (Modularity):** *Modules and Packages*, Python 3.11 Documentation (`https://docs.python.org/3/tutorial/modules.html`). Used as a guide for organizing the project into distinct files (logic, database, and presentation layers) to ensure that changes to the stock mathematics in `inventory_stock.py` do not accidentally break the scanning loop in `warehouse_gui.py`.

- **Official Documentation:** Python 3.11 Standard Library Documentation (specifically modules: `datetime`, `json`, `tkinter`, and `matplotlib`).