SmartStock Inventory Optimization for Retail Stores

Smart Stock Inventory System – Overview

Introduction

A Smart Stock Inventory System is a digital solution designed to help retail stores track their products automatically. It replaces manual stock counting with technology that updates inventory in real time, reduces errors, and ensures that important items are never out of stock.

Main Objective

- To monitor stock levels accurately and automatically
- To alert store owners about low stock, overstock, or expired items
- To make daily operations faster, smarter, and more organized

<u>Python basics</u>

1. Python Casting

- Casting = data type conversion
- Common conversions:
    - int() → integer
    - float() → floating number
    - str() → string
- Example: float(5) = 5.0

---

2. Function Initialization (def)

- Syntax:
- def function_name():

- # code
- Function can have parameters and return values.

---

3. f-string

- f-string =Used to insert variables directly into a string using {}.
- Fastest & cleanest string formatting.
- name = "Mannat"
- print(f"Hello {name}")

---

4. NumPy

- NumPy = Numerical Python (fast math library).

5. Pandas

- Pandas = data analysis and data cleaning library.
- DataFrame .
- Important operations:
    - df.head()
    - df.describe()

---

6. Matplotlib

- graph/visualization library.
- Graph types:
    - Line → plt.plot()
    - Bar → plt.bar()
    - Pie → plt.pie()
    - Scatter → plt.scatter()
- Basic functions:
    - plt.title(), plt.xlabel(), plt.ylabel(), plt.show()

AI MODELS

AI models are computer programs that **learn patterns from data** and then use that learning to **predict, understand, or generate** things on their own.

Company	**Famous Model Family**
Google	**Gemini**
OpenAI	**GPT** (Generative Pre-trained Transformer)
Groq	**Llama**

What is an LLM?

LLM = Large Language Model

It is an AI model trained on huge amounts of text to understand and generate human-like language.

Examples of LLMs

These are all LLMs:

OpenAI

- GPT-5
- GPT-4o
- GPT-3.5

Google

- Gemini
- Gemma

Meta

- Llama 3
- Llama 3.1

LLMs are trained using large datasets.

Chat Completion means using an AI model (LLM) to generate the next response in a conversation.
This is how GPT, Gemini, Groq (Llama), Claude, etc. work.

What is Chat Completion?

It is an API style where you send:

- messages (user + assistant history)
- The model replies with the next message

What is PostgreSQL (Postgres)?

PostgreSQL is an **open-source relational database management system (RDBMS)**.

In simple words:
It stores your data in **tables** (rows & columns)
Helps you **query**, **insert**, **update**, **delete**, and **manage data**
Uses **SQL** language

1. Type of Database

- **PostgreSQL** → *Object–Relational Database* (supports advanced data types, complex queries).
- **MySQL** → *Relational Database* (simple structure, very fast for basic queries).

---

2. Speed

- **PostgreSQL** → Slower for simple reads but *faster for complex queries*.
- **MySQL** → Very *fast for simple read-heavy operations*.

Why do we need Vector Databases?

Normal databases (**MySQL, PostgreSQL**) can't search:

- similar meaning sentences
- similar images

- related documents

They only match *exact words*, not *meaning*.

Vector DBs search using **similarity**, like:

- “Find documents similar to this one”
- “Find images that look like this picture”
- “Find product recommendations”

3 Main Types of Machine Learning

Supervised Learning

Learning with labeled data
The model is given **input + correct output**, and it learns the mapping.

Used for:

- Predicting prices
- Classifying emails (spam/not spam)
- Face recognition

Unsupervised Learning

Learning without labeled data
The model finds **patterns and groups** on its own.

Used for:

- Market segmentation
- Anomaly detection
- Recommendation systems

Reinforcement Learning (RL)

Learning by trial and error
The model gets **rewards** for good actions and **punishments** for bad actions.

Semi-Supervised Learning

Some data → labeled
Most data → unlabeled

Used when labeling is expensive.
Examples:

- Google Photos
- Medical images

1. API (Application Programming Interface)

Definition:
An API is a set of rules that allows two software applications to communicate and exchange data.

Simple meaning:
API works like a **messenger** between two programs. One program sends a request, API forwards it, and returns the response.

Real-life example:
A weather app requests today's temperature using a weather API and displays the result.

Benefits:

- Easy communication between software
- Reuse features without rewriting code
- Saves development time
- Secure, fast, structured communication

---

2. API Request Methods

GET Method

- Used to **fetch data** from the server.
- No request body/payload.
- Example: Getting user details.

POST Method

- Used to **send or create data** on the server.
- Contains payload (JSON data).

- Example: Creating a new user.

---

3. Payload

Definition:
Payload is the **actual data** sent inside an API request or response.

Example:
POST Request Payload:

---

4. Endpoint

Definition:
Endpoint is a **specific URL path** of an API where requests are made.

Examples:

- /api/users – GET all users
- /api/users/1 – GET a user by ID
- /api/users – POST to create a user

---

5. FastAPI vs Flask

Flask

- Lightweight web framework
- Beginner-friendly
- Manually handle validation
- Good for small/medium projects

Example:

```
from flask import Flask, jsonify


app = Flask(__name__)


@app.route("/hello")
def hello():
    return jsonify({"message": "Hello World"})
```

FastAPI

- Modern & high-performance

Example:

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/hello")
def hello():
    return {"message": "Hello World"}
```

---

6. Python :

Create a virtual environment

```
python -m venv venv
```

or

```
python -m venv tenv
```

Activate environment

Windows:

```
venv\Scripts\activate
```

Deactivate

```
deactivate
```

---

7. requirements.txt

Create requirements file

 requirements.txt

Install from requirements

```
pip install -r requirements.txt
```

Purpose:
Helps install the same packages on any system.

---

8. Git Commands

git add

- Adds files to staging area.

Add all files:

```
git add .
```

Add specific file:

```
git add filename.py
```

---

git commit

Saves staged changes:

```
git commit -m "message"
```

---

git push

Sends your commits to remote repository:

```
git push
```

---

git branch

View branches:

```
git branch
```

View all branches (local + remote):

```
git branch --all
```

Switch to another branch

```
git checkout branch_name
```

Or newer way:

```
git switch branch_name
```

Create and switch to a new branch

```
git checkout -b new_branch
```

or

```
git switch -c new_branch
```

---

git pull

Fetches latest changes and merges them into your branch:

```
git pull
```

---

git fetch --all

Downloads all changes from remote **without merging**:

```
git fetch –all
```

Database

A **database** is an organized collection of data that allows easy **storage, retrieval, updating, and management** of information. Databases help keep data structured and accessible for applications like banking systems, websites, social media platforms, and school management systems.

Why Do We Use a Database?

- To store large amounts of data safely
- To avoid mistakes and inconsistency
- To access data quickly
- To allow multiple users to work at the same time
- To keep data organized and structured

Types of Databases

1. Relational Databases (SQL)

Data is stored in **tables** with rows and columns. Uses **SQL language**.
Examples: MySQL, PostgreSQL, Oracle.

2. Non-Relational Databases (NoSQL)

Data stored in **documents, key-value pairs, graphs, or wide columns**.
Examples: MongoDB, Firebase, Cassandra.

Important Database Terms

- **Table:** Collection of rows and columns
- **Row (Record):** One entry of data
- **Column (Field):** Attribute of the data
- **Primary Key:** Unique identifier for each record
- **Foreign Key:** Connects two tables
- **Query:** Command to retrieve or update data
- **Index:** Speeds up searching

Normalization

Normalization is a database process used to **organize data, reduce redundancy, and improve data consistency**.
It breaks a large table into smaller linked tables so that data becomes clean and easy to manage.

Why Normalization Is Important

- Removes duplicate data
- Saves storage
- Ensures consistency
- Avoids update/delete anomalies

- Improves performance
- Makes database structure logical

Normal Forms (1NF, 2NF, 3NF)

First Normal Form (1NF)

Rules:

- No repeating columns
- Values must be **atomic** (single values only)
- Each table must have a **primary key**

Meaning:
Break multi-valued and repeating data into separate rows.

Second Normal Form (2NF)

Rules:

- Table must be in **1NF**
- No **partial dependency**
  (A non-key attribute should not depend on part of a composite key)

Meaning:
Non-key attributes must depend on the **whole primary key**.

Third Normal Form (3NF)

Rules:

- Table must be in **2NF**
- No **transitive dependency**
  (Non-key attribute should not depend on another non-key attribute)

Meaning:
Every column must depend **only** on the primary key.

Summary Table

Normal Form	**What It Fixes**	**Meaning**
1NF	Repeating / multi-valued data	Only single values per cell
2NF	Partial dependency	Non-key depends on whole primary key
3NF	Transitive dependency	Non-key depends only on primary key

Git vs GitHub

Git	**GitHub**
Git is a **software/tool**.	GitHub is a **website/platform**.
Used for **version control** (tracks code changes).	Used for **hosting Git repositories online**.
Works **offline** on your computer.	Works **online** with internet.
Helps you **manage project versions**.	Helps you **share, store, and collaborate** on projects.
Only version control.	Provides extra features: issues, pull requests, teams, actions.

git --version is a command used to **check which version of Git is installed** on your computer.

Git Repository

A **Git repository** is a **storage place** where Git keeps all your project files **and all the history of changes**.

It remembers:

- every file
- every update
- every version

- who changed what

Agentic AI Workflow (Simple Explanation)

Agentic AI is an AI system that can **take actions on its own** to achieve goals, not just answer questions.
It works like a **digital agent** that can plan, decide, and act.

Workflow Steps

1. Goal Definition

- The AI is given a **goal or task** to achieve.
  Example: “Book a flight from Delhi to Jaipur.”

---

2. Environment Perception

- The AI **observes the environment** (data, APIs, or system states).
- It collects the **information it needs** to act.

Example: Checking flight availability or prices.

---

3. Planning

- The AI **decides a sequence of actions** to reach the goal.
- It may **consider multiple options** and choose the best one.

Example: Compare flights → choose the cheapest → check schedule.

---

4. Action / Execution

- AI **performs the action** in the environment.
- Can be **automatic tasks** like sending emails, booking tickets, or running scripts.

---

5. Monitoring / Feedback

- The AI **checks if the action worked**.
- If not, it **adjusts** its plan and tries again.

Example: Flight not available → pick next cheapest option.

Agent:

1. check stock levels automatically.
2. Predict when items will be out-of-stock
3. send alert to manager
4. auto-generate purchase orders
5. update demand forecasts daily

Reactive Agent

A **Reactive Agent** is an agent that:

- **Responds immediately** to the current situation.
- **Does not use memory** or past experience.

Robot vacuum → Changes direction when it hits an obstacle

Goal-Driven Agent

A **Goal-Driven Agent** is an agent that:

- Acts to **achieve a specific goal**
- Can **plan its actions** instead of just reacting

A **Multi-Agent System** is a system where **multiple agents work together** to achieve goals.

- Each agent is **independent**
- Agents can **communicate, cooperate, or compete**

How AI Agents Plan, Think, and Act

1. Perception (See/Observe)

- Agent **perceives the environment** using sensors or data.

---

2. Thinking / Reasoning

- Agent **analyzes the information** it perceives.
- Uses **logic, rules, or models** to decide what to do next.
- Can **predict outcomes** of different actions.

---

3. Planning

- Agent **makes a plan or sequence of actions** to achieve a goal.
- Considers **multiple options** and chooses the **best path**.

Example: Self-driving car plans the route to reach the destination safely and fast.

---

4. Acting / Execution

- Agent **performs the chosen actions** in the environment.
- The action is designed to **move closer to the goal**.

Example: Robot moves forward, orders items, or sends alerts.

---

5. Monitoring / Feedback

- Agent **monitors the result** of its actions.
- If the goal is not achieved, it **replans** or adjusts actions.

Example: Stock agent sees a product is still low → sends another alert.

AutoGen Agents

AutoGen frameworks often use **multiple agents** to work together. Two important ones are:

---

1. User Proxy Agent

- Acts as a **representative of the user**.
- **Receives instructions** or goals from the user.
- Communicates the **user's intent** to the assistant agent.

---

2. Assistant Agent

- Acts as the **worker or executor**.
- **Plans, reasons, and performs tasks** to achieve the user's goal.
- Can **use tools, APIs, or other agents** to complete the task.
- **pip install openai**

- `pip install openai` is a **command** used to install the **OpenAI Python library** on your computer.

```
resp = client.chat.completions.create(
    model="gpt-4o-mini",  # replace with an available model like "gpt-4o-mini" or "gpt-4o" or "gpt-5" etc.
    messages=[
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "Write a two-line poem about chai."}
    ],
    max_tokens=120
  )
```

What This Code Does

This code sends a message to an OpenAI model and asks it to generate a reply.

---

1. resp = client.chat.completions.create(...)

This line **creates a chat completion request**.
Means: "Send my messages to the AI model and give me a response."

---

2. model="gpt-4o-mini"

You are choosing which AI model to use.
Example options:

- gpt-4o-mini
- gpt-4o
- gpt-5

Here it means: **Use the gpt-4o-mini model.**

---

3. messages=[...]

These are the messages we send to the AI.

a) System message

```
{"role": "system", "content": "You are a helpful assistant."}
```

This tells the AI:
“Behave like a helpful assistant.”

b) User message

```
{"role": "user", "content": "Write a two-line poem about chai."}
```

This is the actual question we want AI to answer.

---

4. max_tokens=120

This limits how long the AI’s answer can be.
Max 120 tokens (roughly words/parts of words).

from openai import OpenAI mean?

This line is a **Python import statement**.

“From the **openai** library, import the **OpenAI** class.”

client = OpenAI() mean?

This line **creates a connection** between your Python program and the **OpenAI API**.

The word **client** means a program that talks to a server (OpenAI).

pip install google-generativeai

This installs Google's AI library (Gemini models).

```
import google.generativeai as genai

# 1. Configure API KEY

genai.configure(api_key="YOUR_API_KEY")

# 2. Create Model

model = genai.GenerativeModel("gemini-1.5-flash")

# 3. Send user message

response = model.generate_content(

   "Write a short poem about coffee."

)
```

```
# 4. Get text output
print(response.text)
```

LLM Text Processing Workflow

Text → Tokens

- Input text is **split into small pieces called tokens**.
- Tokens can be:
    - Words → "Hello"
    - Word parts → "playing" → "play" + "ing"
    - Punctuation → "."
- Purpose: LLM cannot understand raw text directly, so we break it into manageable units.

Tokens → Embeddings

- Each token is converted into a **vector of numbers** called an embedding.
- Embeddings **capture the meaning of the token** in a numerical form.
- Example: "chai" → [0.12, 0.88, 0.45, ...]

Embeddings → Transformer Layers

- The embeddings are passed through **transformer layers**.
- Transformer layers **analyze the relationships** between all tokens in the sequence.
- They contain **multiple sub-layers** like:
    - Feed-forward layers
    - Normalization layers

Attention Mechanism

- Attention lets the model **focus on important tokens** in the input.
- Example: In "I love chai because it is warm", the model understands "chai" is related to "warm".
- Helps the model **capture context and meaning** across the sentence.

Output Generation

- After transformer processing, the model predicts the **next token**.
- Tokens are generated one by one → converted back to text → final response.

Prompt Engineering

What is Prompt Engineering?

- **Prompt engineering** is the process of **designing and refining the input (prompt) given to an AI model** so that it produces the desired output.
- A **prompt** is the text you give the model, e.g., a question, instruction, or request.
- • **Rate Limit:** Maximum requests per short period (e.g., per minute).
- • **Quota:** Maximum total usage allowed over a long period (e.g., per day/month).

Logistic Regression

- Used for **binary classification**: 0/1, Yes/No, Spam/Not Spam
- Linear equation:

$$z = w_1x_1 + w_2x_2 + \ldots + b$$

- Sigmoid function converts z to probability:

$$p = \frac{1}{1 + e^{-z}}$$

- **Prediction rule**:
  - If **$p > 0.5$ → class 1**
  - Else → class 0
- **Uses**: Fraud detection, Medical diagnosis, Customer churn, Email spam
- **Code**:

```
from sklearn.linear_model

 import LogisticRegression

model = LogisticRegression()
```

Decision Tree

- Makes decisions like a **flowchart** (if–else questions)
- **Root node** → first question
- **Branches** → answers (Yes/No)
- **Leaf nodes** → final prediction (Class 0/1, Yes/No)

```
Is it Sunny?

  /       \

 Yes       No → Play = No

  |

Is Temperature Hot?

 /   \

Yes   No

|     |

No    Yes
```

Code:

```
from sklearn.tree

 import DecisionTreeClassifier

model = DecisionTreeClassifier()
```

Random Forest

- Ensemble of **many decision trees**
- Uses **random rows + random features** for each tree
- **Classification** → majority vote
- **Regression** → average of outputs
- **Advantages**: Accurate, reduces overfitting, works with large datasets
- **Uses**: Fraud detection, Loan approval, Medical prediction, Stock/Price prediction
- **Code**:

```
from sklearn.ensemble

import RandomForestClassifier

rf = RandomForestClassifier()
```

KNN – K-Nearest Neighbors

- KNN is a **supervised machine learning algorithm**.
- It can be used for **classification** (class labels) or **regression** (numbers).
- It predicts the output of a **new data point** based on **the closest K neighbors** from the training data

How KNN Works

- Choose a value of **K** (number of neighbors).
- Calculate the **distance** between the new point and all training points (common: Euclidean distance).
- Pick the **K nearest points**.
- For **classification**: Take the **majority class** among neighbors.
- For **regression**: Take the **average value** of neighbors.
-

For two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- d → distance between the points
- x_1, y_1 → coordinates of first point
- x_2, y_2 → coordinates of second point

-

K-Means Clustering

It is an **unsupervised machine learning algorithm** that groups data into **K clusters**.

- Each cluster has a **centroid** (mean of points in the cluster).
- The algorithm **minimizes the distance** between points and their assigned cluster centroids.

Steps

1. Initialize K **centroids** $(\mu_1, \mu_2, \ldots, \mu_K)$
2. Assign each point x_i to nearest centroid $(\min ||x_i - \mu_j||^2)$
3. Recalculate centroid:

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

4. Repeat until centroids **don't change**

Linear Regression

- A **supervised machine learning algorithm**.
- Used for **predicting a continuous numerical value** (regression problem).
- Predicts output y from input x using a **linear relationship**.
- Formula:

  $Y=mx+c$

  m → slope, c → intercept

1. IN / NOT IN

IN

Used to match a value with **multiple values**.

Syntax:

```
SELECT column_name

FROM table_name

WHERE column_name IN (value1, value2, value3);
```

Example:

```
SELECT name

FROM students

WHERE city IN ('Delhi', 'Mumbai', 'Chennai');
```

Meaning → students from **either** Delhi, Mumbai, or Chennai.

NOT IN

Used to exclude multiple values.

Example:

```
SELECT name

FROM students

WHERE city NOT IN ('Delhi', 'Mumbai');
```

Meaning → show students who are **not** from Delhi or Mumbai.

2. JOINS

Joins are used to combine rows from two tables based on a common column.

Suppose:

Table 1: Students

std_id name address

Table 2: Subjects

| subject_id | std_id | subject_name |

A. INNER JOIN

Returns only those rows where **matching data exists in both tables**.

Example:

SELECT students.std_id, students.name, subjects.subject_name

FROM students

INNER JOIN subjects

ON students.std_id = subjects.std_id;

Shows only students who have subjects.

B. LEFT JOIN (LEFT OUTER JOIN)

Returns **all rows from the left table** + matching rows from right.
If no match → NULL.

Example:

SELECT students.std_id, students.name, subjects.subject_name

FROM students

LEFT JOIN subjects

ON students.std_id = subjects.std_id;

Shows *all students*, even those who don’t have subjects.

C. RIGHT JOIN (RIGHT OUTER JOIN)

Returns **all rows from the right table** + matching from left.

Example:

SELECT students.name, subjects.subject_name

FROM students

RIGHT JOIN subjects

ON students.std_id = subjects.std_id;

Shows *all subjects*, even if some subjects are not assigned to any student.

D. FULL JOIN (FULL OUTER JOIN)

Returns **all rows from both tables**, matching or not.

Example:

SELECT *

FROM students

FULL OUTER JOIN subjects

ON students.std_id = subjects.std_id;

Every student + every subject, match or no match.

---

3. GROUP BY

Used to group rows that have the **same values**.

Example: Count students in each city

SELECT city, COUNT(*)

FROM students

GROUP BY city;

Output example:

city	count
Delhi	10
Mumbai	8

---

4. HAVING Clause

- WHERE → used before grouping (filters **rows**)
- HAVING → used after grouping (filters **groups**)

Example: Show cities having more than 5 students

SELECT city, COUNT(*)

FROM students

GROUP BY city

HAVING COUNT(*) > 5;

HAVING applies on **aggregate functions** like COUNT(), SUM(), AVG().