---

1. Cover Page 📜

SQLite Order Management System (OMS)

Command-Line Interface (CLI)

Author: [Prakeet Kumar]

Date: November 2025

Version: 1.0

---

2. Introduction 🎯

This document serves as the official project guide for our Command-Line Interface **Order Management System (OMS).** We built this system using **Python** for the brains and **SQLite** for simple, file-based data storage. Our core mission was to create a reliable, **transactional** platform that handles orders, products, and status tracking, with a critical focus on **data integrity**, especially for financial records.

---

3. Problem Statement 💡

We are solving a common business problem: how to accurately record sales without risking data corruption when prices change. Manual processes are error-prone, and simple databases often link order history to the current price, causing massive inconsistencies in reporting.

Our solution focuses on three critical guarantees:

- **Transactional Integrity:** An order is an "all or nothing" event. If anything goes wrong during creation, the entire order is canceled to prevent partial or junk data.
- **Historical Price Accuracy:** Orders must preserve the price agreed upon at the time of sale.
- **Clear Tracking:** Establishing a persistent, auditable trail for every order's lifecycle.

---

4. Functional Requirements (What It Does)

The system delivers the following core features:

ID	Requirement	Description	Status

FR1	**Product Management**	View a list of all products, prices, and SKUs.	Implemented
FR3/FR4	**Order Creation**	**Transactionally** create new orders, automatically capturing the current price (Snapshot Pricing).	Implemented
FR5/FR6	**Order Visibility**	List recent orders and retrieve a complete, detailed view of any single order.	Implemented
FR7	**Status Update**	Change the order's workflow status (e.g., pending $\rightarrow$ shipped).	Implemented
FR8	**Order Deletion**	Safely delete an order; all associated items are automatically removed (Cascade).	Implemented

---

5. Non-functional Requirements (How Well It Works)

These requirements focus on the quality and performance of the system:

ID	Requirement	Description	Priority
NFR1	**Data Integrity**	Uses **Foreign Keys** and **Transactions** to prevent bad data.	High
NFR3	**Maintainability**	Uses the **Repository Pattern** to separate data logic from the CLI interface.	High
NFR2	**Portability**	Runs easily anywhere Python and SQLite are available (everywhere!).	High
NFR5	**Idempotency**	Seeding the initial demo data can run repeatedly without duplicating records.	Medium

---

6. System Architecture 🏗️

The system is built on a reliable **three-tier structure**, designed for clarity and maintainability.

- **Presentation Layer (CLI):** Handled by the **main()** function. This is the user's interface—it only handles menus, input, and printing output. It knows nothing about SQL.
- **Application/Business Layer (The Brains):** Handled by the **OrderRepository** class. This class contains all the business logic: calculating totals, running validation, and managing transactions.
- **Data Persistence Layer (The Vault):** **orders.db** (SQLite). This file stores the data and is only accessed by the OrderRepository.

Key Pattern: We use the **Repository Pattern** to make the CLI code clean and database-agnostic. The CLI asks the Repository, "Please create an order," not "Run this INSERT SQL command."

---

7. Design Diagrams

ER Diagram (Entity-Relationship)

This is the blueprint for how our data is linked.

- **Customer** $\leftarrow$ **Order** (A Customer can have many Orders)
- **Order** $\leftarrow$ **Order Item** (An Order contains many Items)
- **Product** $\leftarrow$ **Order Item** (An Item links back to a Product)

The `unit_price` in **Order Item** is the specific field that implements **Snapshot Pricing** (FR4).

Workflow Diagram: Creating a Transactional Order

The order creation process is protected by a single, atomic transaction:

1. **Preparation:** The system gets the customer ID and fetches the current prices for all requested products (`get_product_price_map`).
2. **START TRANSACTION:** The system opens a secure block (`with self.get_conn() as conn:`).
3. **Insert Header:** The order summary (Total, Status, Customer ID) is saved.
4. **Insert Items:** Each line item, **including its snapshot price**, is saved.
5. **COMMIT:** If successful, the entire order is permanently saved. If any step fails (e.g., database error), the transaction **ROLLS BACK**, and the database remains unchanged (Data Integrity: NFR1).

Decision	Rationale (The Why)

8.

Snapshot Pricing	**Crucial Business Requirement (FR4).** This ensures that every transaction is financially immutable, protecting historical reports from price updates.
Repository Pattern	**Maintainability (NFR3).** This keeps our application logic clean, and if we ever migrate from SQLite to PostgreSQL, we only have to change the `OrderRepository` class, not the entire `main.py` file.
SQLite DB Selection	**Portability and Simplicity (NFR2).** Zero setup required, ideal for a lightweight CLI tool.
Foreign Key Constraints	**Data Integrity (NFR1).** We instruct the database to manage relationships. For example, if we delete an Order (FR8), the database automatically uses `ON DELETE CASCADE` to clean up all associated `Order Items`.

Design Decisions & Rationale

---

9. Implementation Details

We built the system using **Python 3**, relying primarily on the robust **standard library** (`sqlite3`, `pathlib`).

- **Transaction Handling:** Python's context manager (`with conn:`) makes transactions easy and safe, ensuring an automatic rollback if an exception is raised during the order creation.
- **Clear Data Access:** By setting the connection's `row_factory = sqlite3.Row`, we can access database columns by name (e.g., `row['name']`) rather than complex index numbers. This dramatically improves code readability and acts as a "poor-man's ORM."
- **Input Robustness:** Utilities like `prompt_int()` force the user to provide valid numbers for IDs, preventing crashes and improving the user experience.

---

10. Screenshots / Results (Mock-up)

Feature	Expected User Interaction and Output
List Products (1)	Shows a clear, dollar-formatted table of all items (e.g., `$29.99 Laptop Stand`).
Create Order (3)	User enters `PID, QTY: 1, 5`. System returns: `>> Order created successfully! ID: 101`
View Order (4)	Displays a complete invoice-like summary: Customer Name, Status (`SHIPPED`), Line item breakdown, and a final **GRAND TOTAL**.
Error Handling	Attempting to use an invalid status: `Validation Error: Invalid status. Must be one of: {'pending', ...}`

---

11. Testing Approach 🧪

To ensure reliability, we recommend a testing strategy that leverages SQLite's flexibility:

- **In-Memory Testing:** All unit tests should run using `sqlite3.connect(':memory:')`. This creates a database entirely in RAM, making tests **lightning-fast** and perfectly **isolated** (the test database vanishes after the test finishes).
- **Transactional Tests:** We specifically write tests to verify that if an exception occurs mid-way through `create_order`, the system correctly executes a **ROLLBACK** (NFR1).
- **Data Integrity Tests:** Confirmation tests ensure that foreign key rules work as expected, particularly that `delete_order` successfully removes all related line items.

---

12. Challenges Faced 🚧

1. **Concurrency (Future Scaling):** While SQLite is perfect for this project, its file-based nature means it struggles with high-volume, concurrent writes from multiple users. If the system scales up, we would face a challenge needing migration to a multi-user database like PostgreSQL.
2. **User Input:** Crafting the CLI to handle comma-separated inputs (`PID, QTY`) required careful error handling to prevent the program from crashing if the user mistyped the format.
3. **Data Retrieval Complexity:** The `get_order` function had to be carefully designed to pull related data from the **three separate tables** (`orders`, `order_items`, `products`) and stitch it together into a single, clean dictionary for presentation.

---

13. Learnings & Key Takeaways 🏆

1. **Transactions are Non-Negotiable:** The `with conn:` block is the most critical feature learned. In any business system, guaranteeing that data is either **100% complete or 0% recorded** is essential.

2. **Data Modeling Solves Business Problems:** The decision to implement **Snapshot Pricing** was a data modeling solution to a crucial business problem (historical accuracy), demonstrating that the schema is often the foundation of the best solutions.
3. **Clean Code Structure:** The **Repository Pattern** was an immense win for **maintainability**. It proved that abstracting the database details makes the main application logic far simpler and more readable.