INVENTORY CORE

System Design & Technical Specification

Project: Autonomous Mart for Robots (AMR)
Subject: Inventory Microservice v2.0.4
Architecture: Distributed Microservices
Protocols: gRPC, ZMQ, Flatbuffers, Redis, PostgreSQL

Confidential Engineering Document

This document details the internal orchestration logic of the Inventory Service, including the asynchronous robot dispatch system, atomic stock reservation, and distributed state management.

February 13, 2026

Contents

1. Executive Summary

The Inventory Microservice is the authoritative source of truth for all physical entities within the warehouse. Its primary function is to synchronize digital order requests with physical robot movements. It acts as a state machine that transitions orders from `RESERVED` to `PICKING` and finally to `COMPLETED`.

2. Infrastructure Stack & Dependencies

2.1 Persistence Layer (PostgreSQL)

The service utilizes PostgreSQL for long-term storage of stock levels and aisle mappings.

- **Isolation Level:** Read Committed (with SELECT FOR UPDATE for atomicity).
- **Migrations:** Versioned SQL files handled by `golang-migrate`.

2.2 Distributed Caching (Redis)

Redis is utilized for transient state tracking. Specifically, it stores the completion counter for the 5-robot squadron picking logic.

2.3 Messaging (ZeroMQ & Flatbuffers)

To ensure sub-millisecond dispatching to the robot fleet, the service uses **ZeroMQ (PUB/-SUB)**. Unlike JSON/REST, messages are serialized using **Google Flatbuffers**, allowing robots to access pick data without expensive deserialization overhead.

3. Database Architecture & Data Integrity

3.1 Table Schema: available_stock

The `available_stock` table is the core of the service.

Column	Type	Constraint	Descri
`sku`	VARCHAR	PRIMARY KEY	Unique
`name`	TEXT	NOT NULL	Produc
`aisle_type`	VARCHAR	NOT NULL	Robot
`quantity`	INT	CHECK >= 0	Preven
`unit_cost`	NUMERIC	NOT NULL	Inward
`expiry`	TIMESTAMP	NULL	Used fo

3.2 Atomic Reservation Logic

To prevent "Ghost Inventory" (where two clients buy the last item simultaneously), the service implements a mutex-locked decrement.

```
-- Transactional Stock Reservation
BEGIN;
SELECT quantity FROM available_stock WHERE sku = 'APPLE' FOR UPDATE;
UPDATE available_stock SET quantity = quantity - 5 WHERE sku = 'APPLE';
COMMIT;
```

4. gRPC API: Method Specifications

The service exposes a high-performance gRPC interface on port `50051`.

4.1 AssignRobots(OrderID, ItemList)

Called By: Ordering Service.
Logic Flow:

1. Validates order existence in Redis.
2. For each item, retrieves the `aisle_type` from the SQL DB.
3. Groups items by aisle to prevent redundant robot movement.
4. Constructs a Flatbuffers pick-list.
5. Broadcasts to ZMQ port `5556`.

4.2 ReportJobStatus(OrderID, RobotID)

Called By: Autonomous Robots.
Logic Flow:

1. Increments a Redis key: `order:{id}:count`.
2. If the key value equals **5**, triggers the `finalizeOrder` sequence.

5. Robot Orchestration Publishing Mechanism

The service employs a "Broadcast and Filter" strategy.

5.1 ZeroMQ Publishing

The service maintains a persistent ZMQ `PUB` socket.

```
// Internal mq/publisher.go logic
publisher, _ := zmq.NewSocket(zmq.PUB)
publisher.Bind("tcp://*:5556")
// Broadcaster sends the Flatbuffers binary blob
publisher.SendBytes(binaryBlob, 0)
```

5.2 Flatbuffers Payload Structure

The `.fbs` schema defines an `OrderBroadcast` which contains an array of `Item` objects. Each item contains:

- `SKU`: string
- `Quantity`: int
- `Aisle`: string (Filtering criteria for robots)

6. The Order Finalization Sequence

Once the final robot reports completion, the service orchestrates a multi-service handshake.

6.1 Billing Integration

The service calls the **Pricing Service** gRPC method `CalculateBill`. It sends the final pick-list to generate a billing manifest.

6.2 Webhook Notification

The service generates a JSON payload and performs a `POST` request to the Ordering Service's internal webhook:

```
{
  "order_id": "ORD-123",
  "status": "COMPLETED",
  "final_price": 45.99
}
```

7. Error Handling & Edge Cases

7.1 Partial Pick Logic

If a robot finds that an item is physically damaged or missing (despite DB status), it reports a `PARTIAL_PICK`. The service logs this to an `inventory_anomalies` table for manual audit.

7.2 Network Partitions

If Redis is down, the service falls back to an in-memory map to track robot progress. This ensures the warehouse remains operational even if the cache layer fails.

8. Truck Restock Logic

The `RestockItems` method is the entry point for supply-chain data. It is an **Upsert** operation.

1. Checks if SKU exists in `available_stock`.
2. If yes: `Update quantity = existing + new`.
3. If no: `Insert new record`.
4. Notifies the Pricing Service to trigger an immediate re-price event due to stock increase.

9. Conclusion

The Inventory Microservice is designed for extreme reliability. By separating the synchronous gRPC requests from the asynchronous robot broadcasts, it ensures that high-volume orders do not cause congestion in the physical picking process.