FIRINGup Auto-Reorder System

Intelligent Inventory Replenishment Planning Document

Core Algorithm: When to Reorder

```
Reorder Point = (Daily Consumption Rate x Lead Time) + Safety Stock
```

1. Consumption Rate Calculation

Two primary data sources to work with:

Invoices (receiving): Know when X quantity arrived

Counts (depletion): Know when quantity dropped

Calculation approaches (simple to complex):

Simple: (qty_received) / (days_until_next_receive)

Better: Track count-to-count deltas with dates, weighted toward recent behavior

Best: If recipe data + sales exist, predict consumption from sales (more accurate than counting)

Consider: Weekly patterns (weekends busier?), seasonality, trends

2. Supplier Lead Time (User Input)

For each supplier, capture:

- Lead time in business days (order to delivery)
- Order cutoff time (e.g., 'order by 2pm for next-day')
- Delivery days (some only deliver Tue/Thu)
- Minimum order value/quantity

3. Safety Stock

Buffer for consumption variability. Two approaches:

Fixed: Always keep X days extra (e.g., 2-3 days)

Dynamic: Based on consumption variance (if flour usage swings wildly, keep more buffer)

4. Additional Factors to Consider

Factor	Why It Matters
Perishability	Don't over-order items with short shelf life
Case breaks	Round up to orderable units (can't order 0.3 cases)
Volume discounts	Maybe order more if hitting a price break
Supplier grouping	Batch items from same supplier into one order
Minimum order values	Add lower-priority items to hit supplier minimums
Price trends	Optional: flag if current price is unusually high/low vs history

5. The Notification Flow

1. System detects: Ingredient will hit reorder point in X days
2. Groups with other items from same supplier needing reorder
3. Calculates suggested quantities (consumption rate x target days of stock)
4. Sends notification to admin:

    "Sysco Order Suggested"
    "3 items projected to run low"
    "Est. total: $847"
    [Review & Approve] [Dismiss] [Snooze 1 Day]

5. On approve: Sends pre-formatted order email to supplier (or logs for manual ordering)

6. Data Requirements to Add

Per Supplier:

• Lead time (business days)

• Order cutoff time

• Delivery days

• Contact email/phone for orders

• Minimum order value

Per Ingredient (optional overrides):

• Par level (manual override of algorithm)

• Reorder enabled (yes/no toggle)

• Preferred order quantity

System Settings:

• Default safety stock days

• How far ahead to look (notify X days before reorder point)

• Notification preferences (email, SMS, both)

7. Implementation Phases

Phase 1 - Practical MVP:

• Simple moving average consumption

• Fixed lead times per supplier

• Fixed safety stock (e.g., 3 days)

• Email notification with order summary

• Manual approval triggers email to supplier

Phase 2 - Smarter:

• Weighted consumption (recent weeks matter more)

• Dynamic safety stock based on variance

• Supplier order batching with minimum value logic

• Recipe-based consumption prediction

Phase 3 - Advanced:

• Day-of-week consumption patterns

• Seasonal adjustments

• Price optimization suggestions

• Machine learning predictions

Modeled vs Actual Inventory Tracking

Modeled (Theoretical) Inventory:

• Sales x Recipe ingredients = what *should* have been used

• Updates in real-time with every sale

• Great for continuous consumption rate tracking

Actual Inventory:

• Physical counts = what's *really* there

• Ground truth, but only at count moments

The Delta Tells a Story:

```
Modeled says: 50 lbs flour remaining
Actual count: 42 lbs flour remaining
Variance: -8 lbs (16% loss)
```

This variance captures: waste/spillage, portion control drift, recipe inaccuracies, theft, recording errors

For Auto-Reorder:

1. **Consumption Rate:** Use modeled (sales-driven) for predictions - it's real-time and granular
2. **Calibration Factor:** Apply correction based on historical modeled-vs-actual variance
3. **Dynamic Safety Stock:** High variance items need bigger buffers
4. **Reorder Trigger:** Projected Depletion = Current Actual + (Daily Modeled Consumption x Variance Factor)

System Architecture: The Seven Layers

Layer 1: Data Foundation

What exists (already in DB):

• Invoices - receiving dates, quantities, costs, supplier

• Inventory counts - actual quantities at points in time

• Sales transactions - what was sold, when

• Recipes - ingredient requirements per product

• Suppliers - contact info, terms

Database additions needed:

suppliers table additions:

```
lead_time_days (integer)
order_cutoff_time (e.g., '14:00')
delivery_days (e.g., 'tue,thu,sat')
order_email
minimum_order_value
```

ingredients table additions:

```
reorder_enabled (boolean, default true)
par_level_override (nullable - manual override)
safety_stock_days (nullable - per-item override)
```

new table: reorder_settings

```
default_safety_stock_days
lookahead_days (how far ahead to warn)
notification_email
notification_phone
```

Layer 2: Consumption Engine

Runs periodically (or on-demand) to calculate consumption metrics per ingredient:

1. MODELED CONSUMPTION (from sales)

• Query recent sales (e.g., last 30 days)

• Join to recipes -> sum ingredient usage

• Calculate: daily_modeled_consumption = total_used / days

2. ACTUAL CONSUMPTION (from counts)

• Query count history + invoice receipts

• Calculate: actual_depleted = (starting + received - ending)

• Calculate: daily_actual_consumption = actual_depleted / days

3. VARIANCE FACTOR

• variance = actual / modeled (e.g., 1.15 means 15% more actual)

• Store rolling average variance

4. ADJUSTED CONSUMPTION RATE

• adjusted_daily = daily_modeled x variance_factor

• This is your best predictor

Output: ingredient_consumption_metrics table

• ingredient_id

• daily_modeled_rate

• daily_actual_rate

• variance_factor

• confidence_score (based on data quality/quantity)

• last_calculated

Layer 3: Reorder Point Calculator

For each ingredient, determine when to reorder:

```
current_stock = latest count (or modeled current)
adjusted_daily_consumption = from Layer 2
lead_time = supplier.lead_time_days
safety_stock = ingredient.safety_stock_days OR settings.default

days_of_stock = current_stock / adjusted_daily_consumption
reorder_point_days = lead_time + safety_stock
```

TRIGGER when: days_of_stock <= reorder_point_days + lookahead_days

Output: reorder_alerts table

• ingredient_id

• current_stock

• days_remaining

• suggested_order_qty (e.g., 2 weeks of stock)

• urgency (critical/soon/upcoming)

• supplier_id

• created_at

• status (pending/notified/approved/dismissed)

Layer 4: Order Aggregator

Groups alerts into actionable supplier orders:

For each supplier with pending alerts:

1. Collect all ingredients needing reorder

2. Calculate quantities (round to case sizes)

3. Check against minimum order value

   - If under minimum: flag lower-priority items to add

4. Calculate estimated total cost

5. Determine optimal order date (based on delivery days)

Output: draft_order

• supplier_id

• order_date

• delivery_date (estimated)

• line_items: [{ingredient, qty, unit, est_cost}]

• total_estimated

• status: draft

Layer 5: Notification Service

Sends alerts to admins when draft_order created or updated:

Example Notification:

```
  Sysco Order Recommended
  4 items need reordering
  Est. total: $1,247
  Order by: Tomorrow 2pm for Thu delivery

  - Flour (50lb) x 2 cases - $89
  - Tomatoes (case) x 3 - $156
  - Cheese (block) x 5 - $423
  - Olive Oil (gal) x 2 - $67

  [Approve] [Modify] [Snooze 1 Day] [Dismiss]
```

Delivery channels:

• Email (with action links)

• SMS (short version + link)

• In-app notification

Layer 6: Action Handler

Processes admin responses:

On APPROVE:

1. Generate formal order (PDF or email body)
2. Send to supplier.order_email
3. Log order in purchase_orders table
4. Update reorder_alerts status
5. Set expected_delivery_date

On MODIFY:

• Open order editor UI

• Allow qty changes, item removal

• Re-approve when ready

On SNOOZE:

• Set alert.snooze_until = now + 1 day

• Will re-notify tomorrow

On DISMISS:

• Mark alert dismissed

• Optional: 'Don't suggest for X days'

Layer 7: Learning Loop (Phase 2+)

Improves predictions over time:

After each count:

1. Compare predicted vs actual
2. Adjust variance_factor
3. Flag recipes that may be wrong

After each delivery:

1. Record actual lead time
2. Update supplier.lead_time_days (rolling avg)

Weekly:

1. Recalculate consumption patterns
2. Detect trend changes
3. Flag anomalies for review

Visual Architecture Overview

ADMIN DASHBOARD

Notifications | Pending Orders | Settings

NOTIFICATION SERVICE

Email / SMS / In-App Alerts

ORDER AGGREGATOR

Group by supplier, check minimums, schedule

REORDER POINT CALCULATOR

When to order? How much? How urgent?

CONSUMPTION ENGINE

Modeled (sales x recipes) + Actual (counts) + Variance

DATA FOUNDATION

Invoices | Counts | Sales | Recipes | Suppliers

Key Questions to Resolve

1. What communication method preferred for supplier orders (email, portal, phone)?
2. Should the system auto-send orders or always require human approval?
3. What level of safety stock is acceptable (cost of overstock vs risk of stockout)?
4. Are there seasonal patterns that need to be accounted for?
5. Which suppliers have API/portal ordering vs email-only?

Voice AI Phone Ordering System

Intelligent Phone-Based Order Taking

Overview

An AI-powered voice assistant that answers incoming phone calls, takes food orders conversationally, answers customer questions, and submits orders directly to the POS system. Uses Twilio for telephony and OpenAI Realtime API for natural speech-to-speech conversation.

System Architecture

```
CUSTOMER CALL
     ■
     ▼
TWILIO VOICE (receives call, opens Media Stream)
     ■ WebSocket (audio stream)
     ▼
YOUR SERVER (WebSocket bridge, tool calls, order state)
     ■                    ■
     ▼                    ▼
OPENAI REALTIME      YOUR POS DATABASE
API (GPT-4o)         • Menu & pricing
Speech-to-Speech     • Inventory availability
Tool calling         • Order creation
```

Core Components

1. Twilio Voice + Media Streams

Purpose: Telephony infrastructure - receives calls, streams audio

Configuration needed:

- Twilio phone number
- Webhook URL for incoming calls
- Media Stream WebSocket endpoint

How it works:

1. Customer dials your Twilio number
2. Twilio sends webhook to your server
3. Server responds with TwiML to open Media Stream

4. Bidirectional audio flows over WebSocket

2. OpenAI Realtime API

Purpose: Conversational AI with native voice

Key features:

- Direct speech-to-speech (no intermediate transcription)
- Sub-second response latency
- Function/tool calling mid-conversation
- Natural interruption handling

Model: GPT-4o Realtime

3. WebSocket Bridge Server

Purpose: Connects Twilio audio stream to OpenAI, handles business logic

Responsibilities:

- Audio format conversion (Twilio μ-law ↔ OpenAI PCM)
- Tool call execution (query menu, create orders)
- Session state management
- Error handling and escalation

Tool Definitions

The AI assistant needs tools to interact with your POS system:

Menu & Pricing Tools

Tool	Parameters	Returns
get_menu	category (optional)	List of items with descriptions
get_item_details	item_name	Price, sizes, modifiers, description
get_specials	none	Today's specials
check_availability	item_name	In stock (yes/no), alternatives

Order Management Tools

Tool	Parameters	Returns
start_order	order_type (pickup/delivery)	order_session_id
add_item	item, quantity, size, modifiers	Updated order summary
remove_item	item	Updated order summary
get_order_summary	none	Items, subtotal, tax, total
submit_order	customer_name, phone, (address)	Order number, wait time

Business Info Tools

Tool	Parameters	Returns
get_hours	day (optional)	Open/close times
get_wait_time	order_type	Estimated minutes
get_location	none	Address, directions hint
transfer_to_human	reason	Transfers call to staff

Conversation Flow

Standard Order Flow:

```
1. GREETING
   AI: "Thanks for calling [Restaurant]. This is our AI assistant.
        Are you calling to place an order?"
```

```
2. ORDER TYPE
   Customer: "Yeah, I want to do a pickup"
   AI: [calls start_order("pickup")]
   AI: "Great, pickup order! What can I get for you?"

3. ITEM COLLECTION
   Customer: "Can I get a large pepperoni pizza"
   AI: [calls get_item_details("pepperoni pizza")]
   AI: [calls check_availability("pepperoni pizza")]
   AI: [calls add_item("pepperoni pizza", 1, "large")]
   AI: "I've got a large pepperoni for $18.99. Anything else?"

4. ORDER REVIEW
   Customer: "That's it"
   AI: [calls get_order_summary()]
   AI: "Your total is $20.68 including tax. Name for the order?"

5. SUBMIT
   Customer: "Mike, 555-123-4567"
   AI: [calls submit_order("Mike", "555-123-4567")]
   AI: "Thanks Mike! Order #47, ready in about 20 minutes!"
```

Escalation Triggers

The AI should transfer to human staff when:

Trigger	Action
Customer explicitly requests human	Immediate transfer
Complaint or angry tone detected	Transfer with context
Complex catering/large order	Transfer with order so far
Question AI cannot answer (2 attempts)	Transfer
Payment issue or refund request	Transfer
Allergy concern requiring confirmation	Transfer
Technical failure (API error, etc.)	Transfer with apology

Database Integration

New Tables Required:

voice_calls

```
CREATE TABLE voice_calls (
    id INTEGER PRIMARY KEY,
    call_sid TEXT UNIQUE,         -- Twilio call identifier
    phone_from TEXT,
    phone_to TEXT,
    started_at TIMESTAMP,
    ended_at TIMESTAMP,
    duration_seconds INTEGER,
    disposition TEXT,             -- completed/transferred/abandoned
    order_id INTEGER,             -- FK to orders if order placed
    transcript TEXT,              -- Full conversation log
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

voice_call_events

```
CREATE TABLE voice_call_events (
    id INTEGER PRIMARY KEY,
    call_id INTEGER REFERENCES voice_calls(id),
    event_type TEXT,              -- tool_call/transfer/error
    event_data JSON,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Integration with Existing Tables:

The voice system uses existing POS tables:

- **products** - menu items, prices
- **ingredients -** for availability checks
- **orders** - created via submit_order tool
- **order_items** - line items
- **customers** - lookup/create by phone

Concurrency & Multi-Line Support

The system supports multiple simultaneous phone calls. Each incoming call gets its own isolated session with independent state management.

```
Call 1 ■■■ Twilio ■■■ WebSocket 1 ■■■ OpenAI Session 1
Call 2 ■■■ Twilio ■■■ WebSocket 2 ■■■ OpenAI Session 2
Call 3 ■■■ Twilio ■■■ WebSocket 3 ■■■ OpenAI Session 3
     ...
```

Concurrency by Component:

Component	Handles Concurrency?	Notes
Twilio	Yes, automatically	One number handles unlimited concurrent calls
OpenAI Realtime	Yes, per-session	Each call = separate API session
Your Server	Requires async design	Must handle multiple WebSocket connections

Server Scaling Requirements:

Concurrent Calls	Server Type	Specs	Monthly Cost
1-10	Basic VPS	2 CPU, 4GB RAM	$20-40
10-50	Mid-tier server	4 CPU, 8GB RAM	$80-150
50+	Multi-instance + LB	Horizontal scaling	$300-600

Quantitative Cost Analysis

Fixed API Costs (Per Minute of Call):

These costs are constant regardless of server infrastructure:

Component	Cost/Minute	% of Total
Twilio Voice (inbound)	$0.0085	2.7%
Twilio Media Streams	Free	0%
OpenAI Realtime (audio in)	$0.06	19.4%
OpenAI Realtime (audio out)	$0.24	77.4%
Subtotal (API costs)	$0.3085	99.5%

Infrastructure Costs (Per Minute):

Server costs amortized over ~10,000 call-minutes/month:

Scale	Server Monthly	Cost/Minute	% of Total
Small (1-10 concurrent)	$30	$0.003	~1%
Medium (10-50 concurrent)	$120	$0.012	~4%
Large (50+ concurrent)	$450	$0.045	~13%

Total Cost Per Minute by Scale:

Scale	API Costs	Infrastructure	Total/Minute
Small (1-10 calls)	$0.3085	$0.003	$0.311
Medium (10-50 calls)	$0.3085	$0.012	$0.320
Large (50+ calls)	$0.3085	$0.045	$0.354

Key Insight: Infrastructure is <15% of total cost even at large scale. OpenAI Realtime audio output ($0.24/min) dominates at 77% of costs.

Monthly Cost Projections

By Call Volume (assuming 3-min avg call):

Monthly Calls	Total Minutes	API Costs	Infra (Med)	Total Cost
100	300	$93	$30	$123
500	1,500	$463	$80	$543
1,000	3,000	$925	$120	$1,045
2,500	7,500	$2,314	$200	$2,514
5,000	15,000	$4,628	$400	$5,028

Cost Per Order Analysis:

Monthly Calls	Est. Orders (70% conv.)	Cost/Order	Avg Order Value	Cost as % of Order
100	70	$1.76	$30	5.9%
500	350	$1.55	$30	5.2%
1,000	700	$1.49	$30	5.0%
2,500	1,750	$1.44	$30	4.8%
5,000	3,500	$1.44	$30	4.8%

Break-Even Analysis:

Assuming the voice AI replaces one part-time employee during peak hours:

Metric	Value
Part-time wage (phone duty)	$15/hour
Hours replaced per day	4 hours (lunch + dinner rush)
Monthly labor savings	$1,800 (30 days × 4 hrs × $15)
Break-even call volume	~1,700 calls/month
Additional benefit	No missed calls, 24/7 capability

ROI Considerations:

- **Labor offset:** Each call handled = 3-5 min staff time saved
- **Missed call recovery:** AI answers during rush when staff can't

- **Upselling consistency:** AI always offers drinks/sides (humans forget)
- **Extended hours:** Take orders before open / after close
- **Order accuracy:** No mishearing, automatic logging

Implementation Phases

Phase 1 - Basic MVP

- Twilio number + webhook setup
- WebSocket bridge server
- OpenAI Realtime integration
- Basic tools: get_menu, add_item, submit_order
- Simple order flow (pickup only)
- Transfer to human fallback

Phase 2 - Full Features

- Delivery orders with address capture
- Inventory availability checks
- Customer lookup by phone
- Rewards points integration
- Order modifications mid-call
- Specials and upselling

Phase 3 - Advanced

- Call analytics dashboard
- Sentiment detection
- Multi-language support
- Outbound calls (order ready notifications)
- Voice authentication for repeat customers

Implementation Steps

Step 1: Twilio Setup

1. Create Twilio account and purchase phone number
2. Install Twilio SDK: pip install twilio
3. Configure webhook URLs in Twilio console:

- Voice webhook: https://yourserver.com/voice/incoming
- Status callback: https://yourserver.com/voice/status

Step 2: OpenAI Realtime Setup

1. Get OpenAI API key with Realtime API access
2. Install SDK: pip install openai
3. Configure model and tools in session setup

Step 3: WebSocket Server

Create /routes/voice_routes.py:

```
@voice_bp.route('/voice/incoming', methods=['POST'])
def incoming_call():
    response = VoiceResponse()
    response.say("Please wait while I connect you.")

    connect = Connect()
    connect.stream(url='wss://yourserver.com/voice/stream')
    response.append(connect)

    return str(response)
```

Step 4: Register Routes

In app.py:

```
from routes.voice_routes import voice_bp
app.register_blueprint(voice_bp)
```

Monitoring & Analytics

Key Metrics to Track:

Metric	Target
Call completion rate	>85%
Order conversion rate	>70%
Average call duration	<4 min
Transfer rate	<15%

Customer satisfaction	>4.0/5

Dashboard Elements:

- Calls today / this week / this month
- Orders placed via voice
- Revenue from voice orders
- Common transfer reasons
- Peak call times
- Average order value (voice vs other channels)

Security Considerations

- **No payment over phone:** Collect payment in-store or redirect to secure link
- **Phone validation:** Verify caller ID when possible
- **Rate limiting:** Prevent abuse of AI minutes
- **PII handling:** Don't log sensitive data in transcripts
- **Call recording consent:** Announce if calls are recorded (state laws vary)