Nutrition Agent - Master Documentation

Complete Analysis, Decisions & Implementation Guide

Document Version: 1.0
Last Updated: October 4, 2025
Status: Post Day 6 Completion, Pre Day 7 Implementation

📋 EXECUTIVE SUMMARY

Quick Reference

- **Status:** Day 6 (Tracking System) completed. Nutrition Agent planned for Day 7.
- **Key Finding:** ~70% of originally planned Nutrition Agent functionality already exists in other services
- **Recommendation:** Build focused 6-tool agent instead of original 10-tool plan
- **Time Savings:** 4 hours (from 10 hours to 6 hours)
- **Primary Users:** WhatsApp Bot (Day 9), Frontend Chat (Day 13-14), Master Orchestrator (Day 10)

Critical Insight

The Nutrition Agent is NOT a standalone nutrition calculation system.
It's an **intelligence & orchestration layer** that combines existing services with AI reasoning to provide context-aware coaching.

✅ EXISTING FUNCTIONALITY AUDIT

What Already Works (Don't Rebuild)

1. BMR/TDEE Calculations ✅ COMPLETE

Location: `backend/app/services/onboarding.py`


python

```python
class OnboardingService:
    @staticmethod
    def calculate_bmr(weight_kg, height_cm, age, sex) -> float:
        """Mifflin-St Jeor equation - FULLY IMPLEMENTED"""
        # Men: (10 × weight) + (6.25 × height) - (5 × age) + 5
        # Women: (10 × weight) + (6.25 × height) - (5 × age) - 161

    @staticmethod
    def calculate_tdee(bmr, activity_level) -> float:
        """BMR × activity multiplier (1.2 - 1.9)"""

    @staticmethod
    def calculate_goal_calories(tdee, goal_type) -> float:
        """TDEE ± goal adjustment (±500 cal)"""
```

Database Storage:

- `UserProfile.bmr` (calculated during onboarding)
- `UserProfile.tdee` (calculated during onboarding)
- `UserProfile.goal_calories` (calculated during onboarding)

Evidence: All calculations happen in onboarding flow and are stored permanently.

❌ **DON'T BUILD:** Nutrition Agent tools for BMR/TDEE calculation

---

2. Daily Targets & Progress Tracking ✅ COMPLETE

Location: `backend/app/services/consumption_services.py`

python

```
def get_today_summary(user_id: int) -> Dict:
    """
    FULLY IMPLEMENTED - Returns:
    - meals_planned, meals_consumed, meals_skipped
    - total_calories, total_macros (protein, carbs, fat, fiber)
    - compliance_rate
    - remaining_targets (calories, macros)
    - progress percentages
    """

def _calculate_remaining_targets(user_id, daily_totals) -> Dict:
    """
    FULLY IMPLEMENTED
    - target_calories from UserProfile.goal_calories
    - remaining = target - consumed
    - percentage_complete
    """
```

API Endpoint: `GET /tracking/today` ✅ OPERATIONAL

❌ **DON'T BUILD:** Nutrition Agent tool for checking daily targets

---

3. Meal Macro Analysis ✅ COMPLETE (Basic)

Location: `backend/app/services/consumption_services.py`


python

```
def _calculate_meal_macros(meal_log) -> Dict:
    """
    FULLY IMPLEMENTED - Basic macro calculations
    - Handles portion multipliers
    - Returns calories, protein, carbs, fat, fiber
    """
```

⚠️ **PARTIAL:** Basic calculations exist. Missing: insights, quality scores, recommendations.

✅ **BUILD:** Enhanced version with intelligence layer

---

4. Weekly Progress Tracking ✅ COMPLETE

Location: `backend/app/services/consumption_services.py`

python

```python
def get_consumption_history(user_id, days=7) -> Dict:
    """
    FULLY IMPLEMENTED
    - Groups meals by date
    - Daily statistics (planned/consumed/skipped)
    - Calories and macros per day
    - Compliance trends
    """
```

API Endpoint: `GET /tracking/history?days=7` ✅ OPERATIONAL

❌ **DON'T BUILD:** Basic weekly tracking (already exists)

✅ **BUILD:** AI-enhanced progress reports with insights (enhancement only)

---

5. Recipe Goal-Based Selection ✅ COMPLETE

Location: `backend/app/agents/planning_agent.py`

python

```python
def select_recipes_for_goal(goal: str, count: int) -> List:
    """
    FULLY IMPLEMENTED
    - Queries recipes by goal type
    - Scores recipes for goal alignment
    - Returns top N recipes
    """
```

❌ **DON'T BUILD:** Basic goal-based filtering

✅ **BUILD:** Context-aware enhancement (time, mood, remaining macros)

---

6. Inventory-Aware Recipe Filtering ✅ COMPLETE

Location: `backend/app/services/inventory_service.py`

python

```
def check_recipe_availability(user_id, recipe_id) -> Dict:
    """FULLY IMPLEMENTED"""

def get_makeable_recipes(user_id, limit=10) -> List:
    """FULLY IMPLEMENTED - Filters by inventory"""
```

✅ **USE AS-IS:** Already provides inventory filtering for suggestions

---

7. Meal Timing Windows ✅ COMPLETE (Static)

Location: `backend/app/services/onboarding.py` + `UserPath` model

python

```
MEAL_WINDOWS = {
    PathType.IF_16_8: [
        {"meal": "lunch", "start_time": "12:00", "end_time": "13:00"},
        {"meal": "dinner", "start_time": "18:00", "end_time": "20:00"}
    ],
    # ... 5 path types defined
}
```

Database Storage: `UserPath.meal_windows` (JSON field)

⚠️ **PARTIAL:** Static windows exist. Missing: dynamic optimization based on context.

✅ **BUILD:** Optional enhancement for context-aware timing

---

8. Automated Notifications ✅ COMPLETE

Location: `backend/app/workers/notification_worker.py`

Meal Reminders:

python

```
async def _trigger_meal_reminders(notification_service, db):
    """
    FULLY IMPLEMENTED
    - Finds meals in 30-minute window for ALL active users
    - Sends notification_service.send_meal_reminder()
    - Runs every 5 minutes
    """
```

Daily Summaries:


python

```
async def _trigger_daily_summaries(notification_service, consumption_service):
    """
    FULLY IMPLEMENTED
    - At 9 PM (21:00) for all active users
    - Gets today's summary from consumption_service
    - Sends notification_service.send_daily_summary()
    """
```

Weekly Reports:


python

```
async def _trigger_weekly_reports(notification_service, consumption_service):
    """
    FULLY IMPLEMENTED
    - Sunday 8 PM for all active users
    - Gets 7-day analytics
    - Sends notification_service.send_weekly_report()
    """
```

❌ **DON'T BUILD:** Any notification triggering logic - fully automated

---

9. Achievement & Progress Notifications ✅ COMPLETE

Location: `backend/app/agents/tracking_agent.py`


python

```
# In log_meal_consumption():
achievements = self._check_meal_achievements(result)
for achievement in achievements:
    # ✅ Already sends achievement notification
    await self.notification_service.send_achievement(...)

    # ✅ Already broadcasts via WebSocket
    await websocket_manager.broadcast_to_user(...)

# ✅ Already sends progress update
await self.notification_service.send_progress_update(
    compliance_rate, calories_consumed, calories_remaining
)

# ✅ Already broadcasts macro update
await websocket_manager.broadcast_to_user(
    message={"event_type": "macro_update", ...}
)
```

❌ **DON'T BUILD:** Achievement detection or progress notifications - fully automated

---

❌ REDUNDANCY ANALYSIS

Original Nutrition Agent (10 Tools) - What NOT to Build

Tool #	Original Tool Name	Status	Action
1	calculate_bmr_tdee	❌ 100% REDUNDANT	Use OnboardingService
2	adjust_calories_for_goal	❌ 100% REDUNDANT	Use OnboardingService
3	analyze_meal_macros	⚠️ 50% REDUNDANT	Enhance existing
4	check_daily_targets	❌ 100% REDUNDANT	Use ConsumptionService.get_today_summary()
5	suggest_next_meal	⚠️ 30% REDUNDANT	Build with orchestration
6	calculate_meal_timing	⚠️ 60% REDUNDANT	Optional enhancement
7	provide_nutrition_education	✅ NEW	Build from scratch
8	track_weekly_progress	❌ 90% REDUNDANT	Use ConsumptionService.get_consumption_history()
9	adjust_portions	✅ NEW	Build from scratch
10	generate_progress_report	⚠️ 50% REDUNDANT	Format existing data

Conclusion: Only 3 tools are truly new. Others are either redundant or enhancements.

✅ WHAT ACTUALLY NEEDS BUILDING

Revised Nutrition Agent (6 Focused Tools)

Tool 1: suggest_next_meal() ⭐ CRITICAL

Priority: HIGH - Build Day 7
Time Estimate: 2.5 hours
Primary Users: WhatsApp Bot, Frontend Chat, Master Orchestrator

What It Does: Combines multiple data sources with AI reasoning to provide context-aware meal suggestions.

How It Works:


python

```python
def suggest_next_meal(user_id, meal_type, context):
    # 1. Gather data from existing services
    daily_summary = consumption_service.get_today_summary(user_id)
    remaining = daily_summary['remaining_targets']
    makeable = inventory_service.get_makeable_recipes(user_id)
    goal_recipes = planning_agent.select_recipes_for_goal(user_goal)

    # 2. NEW: Add intelligence layer
    scored_recipes = []
    for recipe in makeable:
        score = calculate_context_score(
            recipe,
            remaining_macros=remaining,
            time_of_day=context['time'],
            user_mood=context.get('mood'),
            weather=context.get('weather')
        )
        explanation = generate_explanation(recipe, score)
        scored_recipes.append((recipe, score, explanation))

    # 3. Return top 3 with reasoning
    return top_3_with_explanations(scored_recipes)
```

Scoring Factors:

- Macro fit (30%): How well does it fit remaining macros?
- Context match (25%): Appropriate for time/mood/weather?
- Inventory optimization (20%): Uses expiring items?
- Goal alignment (15%): Fits user's fitness goal?
- Preference (10%): Historical acceptance rate?

Example Output:


json

```
{
 "suggestions": [
  {
   "recipe": "One-Pan Chicken",
   "calories": 480,
   "protein": 40,
   "score": 0.89,
   "explanation": "Perfect for tired evenings - quick (25 min), hits your protein target, uses veggies expiring tomorrow
  }
 ]
}
```


Where Used:

- WhatsApp Bot: "What should I eat?" command
- Frontend: "Get Meal Suggestion" button
- Master Orchestrator: suggest_next_meal_workflow

Tool 2: provide_nutrition_education() ⭐ **IMPORTANT**

Priority: MEDIUM - Build Day 9 (with WhatsApp Bot)
Time Estimate: 1.5 hours
Primary Users: WhatsApp Bot, Frontend Help System

What It Does: Provides personalized educational content about nutrition topics.

Topics Database:

python

```
EDUCATION_CONTENT = {
  "protein": {
    "beginner": "Protein builds muscle. Aim for {target}g per day.",
    "intermediate": "Protein provides amino acids for muscle repair. Your target is {target}g/day ({per_kg}g per kg bo
    "advanced": "Protein synthesis is maximized at 30-40g per meal. Your current intake of {current}g is {percentage
  },
  "macros": {...},
  "meal_timing": {...},
  "hydration": {...}
}
```

Personalization:

python

```python
def provide_nutrition_education(topic, user_context):
    content = EDUCATION_CONTENT[topic]

    # Determine user's knowledge level
    level = determine_knowledge_level(user_context)

    # Get base content
    template = content[level]

    # Personalize with user data
    personalized = template.format(
        target=user_context['protein_target'],
        current=user_context['current_protein'],
        percentage=calculate_percentage(...)
    )

    # Add goal-specific advice
    if user_context['goal'] == 'muscle_gain':
        personalized += "\n\nFor muscle gain: Focus on 2g per kg body weight."

    return personalized
```

Where Used:

- WhatsApp Bot: "Why protein?", "What are macros?" questions
- Frontend: Tooltips, help sections, educational modals

Tool 3: adjust_portions() ⚡ **NICE-TO-HAVE**

Priority: LOW - Build Day 13+ (Frontend Polish)
Time Estimate: 1 hour
Primary Users: Frontend Customization, Advanced Meal Planning

What It Does: Calculates optimal portion sizes to hit specific macro targets.

Algorithm:

python

```python
def adjust_portions(recipe_id, target_macros):
    recipe_macros = get_recipe_macros(recipe_id)

    # Calculate scaling factor for each macro
    protein_factor = target_macros['protein'] / recipe_macros['protein']
    carbs_factor = target_macros['carbs'] / recipe_macros['carbs']
    fat_factor = target_macros['fat'] / recipe_macros['fat']

    # Use weighted average (prioritize protein)
    optimal_factor = (
        protein_factor * 0.5 +
        carbs_factor * 0.3 +
        fat_factor * 0.2
    )

    # Constrain to reasonable bounds
    final_factor = max(0.5, min(2.5, optimal_factor))

    return {
        "portion_multiplier": final_factor,
        "adjusted_macros": calculate_adjusted_macros(recipe_macros, final_factor),
        "explanation": f"Adjusted to {final_factor}x to optimize protein intake"
    }
```

Where Used:

- Frontend: User customization feature
- Advanced meal planning

Tool 4: Enhanced analyze_meal_macros() ⚡ **NICE-TO-HAVE**

Priority: LOW - Build Day 13+ (Frontend Polish)
Time Estimate: 1 hour
Primary Users: Frontend Meal Preview, Enhanced Logging Feedback

What It Does: Wraps existing `_calculate_meal_macros()` with intelligence layer.

Enhancement:


python

```python
def analyze_meal_macros(meal_id):
    # Get basic macros from existing service
    basic_macros = consumption_service._calculate_meal_macros(meal_id)

    # ADD: Intelligence layer
    user_targets = get_user_targets()

    enhanced = {
        **basic_macros,

        # NEW: Percentage of daily targets
        "percentage_of_daily": {
            "protein": (basic_macros['protein_g'] / user_targets['protein']) * 100,
            "carbs": (basic_macros['carbs_g'] / user_targets['carbs']) * 100,
            ...
        },

        # NEW: Quality score
        "quality_score": calculate_nutritional_quality(basic_macros),

        # NEW: Insights
        "insights": generate_meal_insights(basic_macros, user_targets),

        # NEW: Recommendations
        "recommendations": generate_recommendations(basic_macros, context)
    }

    return enhanced
```

Example Output:

json

```
{
 "calories": 450,
 "protein_g": 30,
 "percentage_of_daily": {"protein": 25, "carbs": 18, "fat": 20},
 "quality_score": 8.5,
 "insights": [
  "High protein - great for post-workout recovery",
  "Balanced macros fit your fat loss goal"
 ],
 "recommendations": ["Perfect for lunch!", "Consider smaller portion for dinner"]
}
```

Where Used:

- Enhanced WebSocket broadcasts
- Frontend meal preview cards

Tool 5: generate_progress_report() ⚡ NICE-TO-HAVE

Priority: LOW - Build Day 13+ (Frontend Polish)
Time Estimate: 1.5 hours
Primary Users: Frontend Dashboard, Enhanced Weekly Reports

What It Does: Formats existing data with AI-generated insights.

Implementation:

python

```python
def generate_progress_report(user_id, period='weekly'):
    # Get data from existing service
    history = consumption_service.get_consumption_history(user_id, days=7)

    # ADD: AI insights
    insights = []

    # Analyze protein consistency
    protein_days = [day['protein'] for day in history.values()]
    if average(protein_days) > user_target * 0.9:
        insights.append("🎯 Hitting protein targets 6/7 days - excellent!")

    # Analyze trends
    if detect_trend(history, 'compliance', direction='up'):
        insights.append("📈 Compliance up 12% from last week")

    # Analyze patterns
    if detect_pattern(history, 'dinner_portions_increasing'):
        insights.append("⚠️ Dinner portions trending higher - consider lighter evening meals")

    # Generate action items
    action_items = generate_action_items(history, insights)

    return {
        "history": history,
        "ai_insights": insights,
        "trends": analyze_trends(history),
        "action_items": action_items
    }
```

Where Used:

- Frontend dashboard
- Enhanced notification reports

Tool 6: optimize_meal_timing() ⚡ **OPTIONAL**

Priority: VERY LOW - Build if time permits
Time Estimate: 1 hour
Primary Users: Advanced Planning Features

What It Does: Dynamic meal timing based on context (workout, meetings, etc.)

Implementation:

python

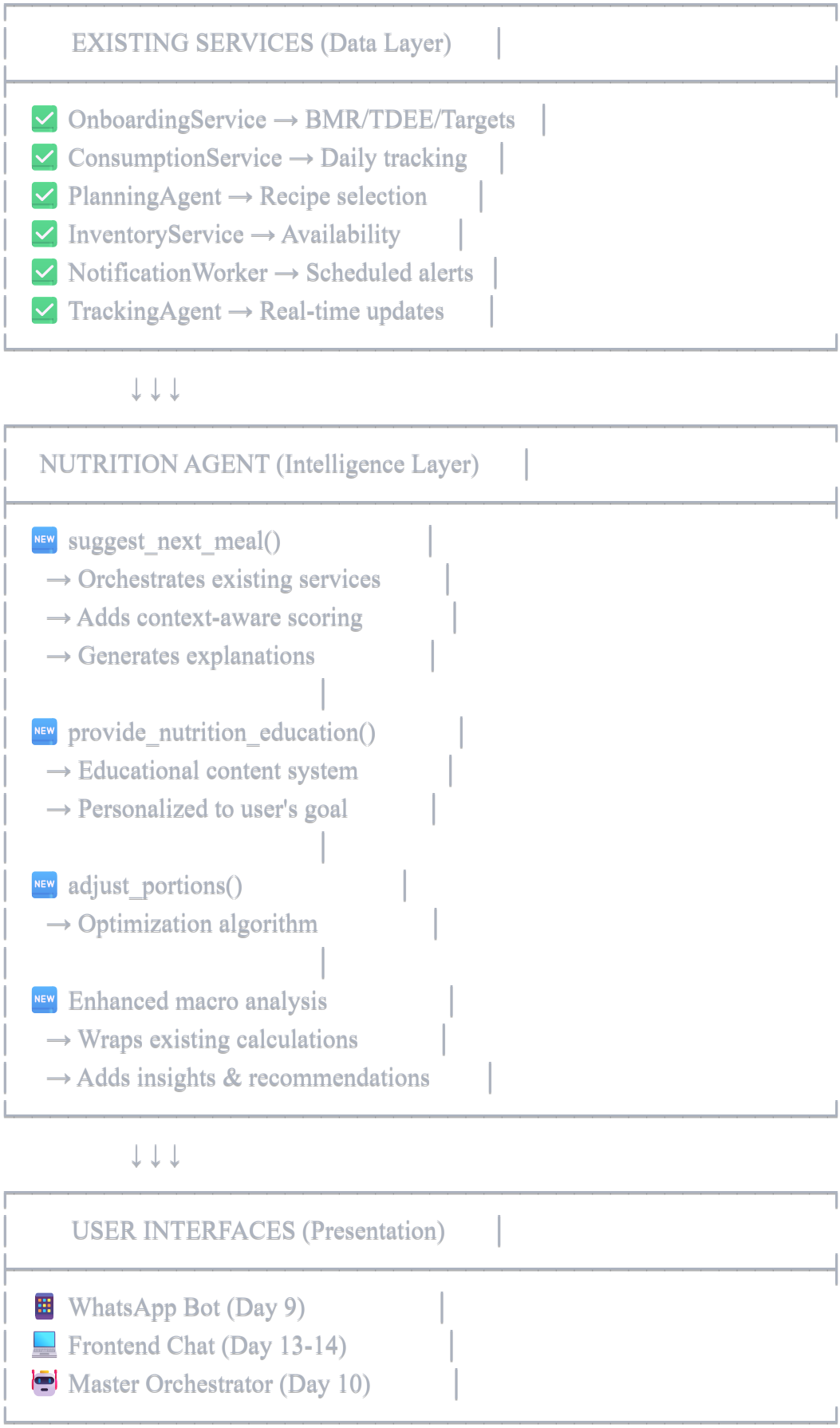
```python
def optimize_meal_timing(user_id, date, context):
    # Get static windows from UserPath
    base_windows = user_path.meal_windows

    # Adjust for context
    if context.get('gym_time'):
        # Move breakfast to post-workout
        base_windows['breakfast']['time'] = context['gym_time'] + 30_min
        base_windows['breakfast']['reasoning'] = "Post-workout for protein synthesis"

    if context.get('important_meeting'):
        # Move lunch earlier
        base_windows['lunch']['time'] = meeting_time - 45_min
        base_windows['lunch']['reasoning'] = "Before your meeting"

    return optimized_windows
```

Where Used:

- Advanced meal planning
- Workout-based optimization

🧬 ARCHITECTURAL INTEGRATION

How Nutrition Agent Fits in Your System



Key Principle: Nutrition Agent NEVER duplicates calculations. It orchestrates and enhances.

🗺️ USER JOURNEY INTEGRATION

Where Nutrition Agent Tools Are Used

Journey 1: WhatsApp Bot (Primary User)

Command: "What should I eat for dinner?"

```
User (WhatsApp): "What should I eat for dinner? I'm tired"
    ↓
WhatsApp Agent (Day 9) parses command
    ↓
Calls: nutrition_agent.suggest_next_meal(
    user_id=123,
    meal_type="dinner",
    context={"time": "18:30", "mood": "tired"}
)
    ↓
Nutrition Agent:
  1. Gets remaining macros (ConsumptionService)
  2. Gets available recipes (InventoryService)
  3. Applies context-aware scoring
  4. Generates explanation
    ↓
Returns: {
    "recipe": "One-Pan Chicken",
    "explanation": "Quick (25 min) for tired evenings,
                    hits protein target, uses expiring veggies"
}
    ↓
WhatsApp sends: "🍽️ I suggest One-Pan Chicken! ..."
```

Command: "Why do I need so much protein?"

```
User (WhatsApp): "Why do I need so much protein?"
    ↓
WhatsApp Agent parses: intent="education", topic="protein"
    ↓
Calls: nutrition_agent.provide_nutrition_education(
    topic="protein",
    user_context={"goal": "muscle_gain", "current": 120}
)
    ↓
Returns personalized education
    ↓
WhatsApp sends explanation
```

Journey 2: Frontend Chat Interface

User clicks "Get Meal Suggestion"

```
Frontend (React/Streamlit)
    ↓
API Call: POST /nutrition/suggest-meal
    ↓
nutrition.py endpoint → nutrition_agent.suggest_next_meal()
    ↓
Returns suggestions with explanations
    ↓
Frontend displays suggestion cards with "Accept" button
```

Journey 3: Master Orchestrator Workflow

From sprint doc: suggest_next_meal_workflow

Master Orchestrator triggers workflow
↓
Step 1: Context Gathering
Step 2: Parallel Analysis (Tracking + Existing Services)
Step 3: SUGGESTION GENERATION → nutrition_agent.suggest_next_meal() ⭐
Step 4: Availability Check
Step 5: Personalization
Step 6: Response Generation

Without Nutrition Agent, Step 3 fails.

Journey 4: Enhanced Meal Logging (Optional)

Current behavior (works fine):

User logs meal → WebSocket broadcasts basic macros

Enhanced behavior (if Tool 4 built):

User logs meal
↓
Tracking Agent calls nutrition_agent.analyze_meal_macros()
↓
WebSocket broadcasts enhanced data with insights
↓
Frontend shows: "Great! 25% of daily protein ✅"

IMPLEMENTATION TIMELINE

Phased Approach (Recommended)

Phase 1: Day 7 - Core Intelligence (4-5 hours)

Build:

1. Nutrition Agent core class (1.5 hours)
    - Agent initialization
    - State management
    - Tool registration system

2. suggest_next_meal() (2.5 hours) ⭐ CRITICAL
    - Context gathering from existing services
    - Scoring algorithm implementation
    - Explanation generation
    - API endpoint
3. Basic tests (1 hour)

Why Now:

- Most critical tool for WhatsApp Bot (Day 9)
- Needed by Master Orchestrator (Day 10)
- Time to refine before integration

Deliverables:

- Nutrition Agent class
- suggest_next_meal() fully functional
- API endpoint: POST /nutrition/suggest-meal
- Unit tests

Phase 2: Day 9 - Educational Content (1.5 hours)

Build:

1. provide_nutrition_education() (1.5 hours)
    - Educational content database
    - Topic routing
    - Personalization logic
    - WhatsApp integration

Why Now:

- Building WhatsApp Bot
- Will know exactly what content is needed
- Can test immediately with real interactions

Deliverables:

- Educational content system
- API endpoint: GET /nutrition/education/{topic}
- WhatsApp command integration

Phase 3: Day 13+ - Polish Features (2-3 hours)

Build (as time permits):

1. analyze_meal_macros() enhancement (1 hour)
    - Insights generation
    - Quality scoring
    - WebSocket integration
2. adjust_portions() (1 hour)
    - Optimization algorithm
    - API endpoint
3. generate_progress_report() (1.5 hours)
    - AI insights
    - Trend analysis
    - Frontend dashboard integration

Why Later:

- UI enhancements
- Not blocking core functionality
- Can be added iteratively

Deliverables:

- Enhanced features for frontend
- Better user experience

Alternative: Aggressive Timeline (If Time Constrained)

Day 7: Build ONLY suggest_next_meal() (3 hours)

- Skip everything else
- Focus on the one critical tool
- Add others only if users request them

Pros:

- ✅ Minimum viable nutrition intelligence
- ✅ Unblocks WhatsApp Bot and Orchestrator
- ✅ Can iterate based on user feedback

Cons:

- ⚠️ No educational content (WhatsApp limited)
- ⚠️ No advanced features

🎯 KEY DECISIONS & RATIONALE

Decision 1: Build 6 Tools Instead of 10

Rationale:

- 70% of functionality already exists in other services
- Avoid redundant code and maintenance burden
- Focus on true intelligence additions
- Save 4 hours of development time

Evidence:

- OnboardingService handles all BMR/TDEE calculations
- ConsumptionService handles daily tracking
- NotificationWorker handles all scheduled notifications
- TrackingAgent handles real-time meal updates

Decision 2: Prioritize suggest_next_meal()

Rationale:

- Most impactful tool for user experience
- Directly used by WhatsApp Bot (primary interface)
- Core "intelligence" of the application

- Demonstrates AI capabilities

Evidence:

- Every "What should I eat?" interaction uses this
- Master Orchestrator workflow depends on it
- Frontend chat feature requires it

Decision 3: Delay Educational Content to Day 9

Rationale:

- Can't predict what content is needed until WhatsApp Bot exists
- Better to build based on actual user questions
- Prevents wasted effort on unused content

Alternative Considered:

- Build all content on Day 7
- Rejected: Too speculative, likely to change

Decision 4: Make Enhancement Tools Optional

Rationale:

- analyze_meal_macros(), adjust_portions(), progress_report() are nice-to-have
- Don't block core functionality
- Can be added based on user feedback

Alternative Considered:

- Build everything upfront
- Rejected: Premature optimization, time-constrained

Decision 5: Nutrition Agent as Orchestration Layer

Rationale:

- Don't duplicate existing calculations
- Reuse battle-tested business logic
- Focus on adding intelligence, not reimplementing basics

Architecture Decision:

BAD: Nutrition Agent does its own calculations
→ Duplicate code, maintenance nightmare

GOOD: Nutrition Agent orchestrates existing services
→ Reuse code, add intelligence layer

🚨 CRITICAL REMINDERS

What NOT to Build (Avoid These Mistakes)

❌ **DON'T** create new BMR/TDEE calculation functions

- Use OnboardingService.calculate_bmr/tdee/goal_calories

❌ **DON'T** create new daily tracking functions

- Use ConsumptionService.get_today_summary()

❌ **DON'T** create notification triggering logic

- NotificationWorker already handles all scheduled notifications

❌ **DON'T** create achievement detection

- TrackingAgent already detects and sends achievement notifications

❌ **DON'T** create basic macro calculations

- ConsumptionService._calculate_meal_macros() exists

❌ **DON'T** create weekly tracking

- ConsumptionService.get_consumption_history() exists

What TO Build (Focus Here)

✅ **DO** build suggest_next_meal() with context-aware intelligence

- This is the CORE value-add

✅ **DO** orchestrate existing services

- Combine ConsumptionService + InventoryService + PlanningAgent

✅ **DO** add "WHY" explanations

- Generate human-readable reasoning

✅ **DO** provide educational content

- Answer user questions about nutrition

✅ **DO** add insights to existing data

- Enhance, don't replace

📝 API ENDPOINTS REFERENCE

Endpoints to Create

python

```
# Day 7
POST /nutrition/suggest-meal
  Request: {
    "meal_type": "lunch|dinner|breakfast|snack",
    "context": {
      "time": "18:30",
      "mood": "tired",
      "weather": "cold"
    }
  }
  Response: {
    "suggestions": [
      {
        "recipe_id": 123,
        "recipe_name": "One-Pan Chicken",
        "calories": 480,
        "protein_g": 40,
        "score": 0.89,
        "explanation": "Perfect for tired evenings..."
      }
    ]
  }

# Day 9
GET /nutrition/education/{topic}
  Params: topic = "protein|macros|meal_timing|hydration"
  Response: {
    "topic": "protein",
    "content": "For muscle gain, protein helps...",
    "personalized": true,
    "user_specific_data": {...}
  }

# Day 13+ (Optional)
POST /nutrition/adjust-portions
GET /nutrition/analyze-meal/{meal_id}
GET /nutrition/progress-report
```

Endpoints to REUSE (Don't Create)

python

```
# These already exist - use them
GET /tracking/today → Daily targets & progress
GET /tracking/history → Weekly tracking
GET /onboarding/targets → BMR/TDEE/goals
```

🧪 TESTING STRATEGY

Day 7 Tests (suggest_next_meal)

python

```python
def test_suggest_meal_with_context():
    """Test context-aware meal suggestion"""
    result = nutrition_agent.suggest_next_meal(
        user_id=1,
        meal_type="dinner",
        context={"time": "18:30", "mood": "tired"}
    )
    assert len(result['suggestions']) == 3
    assert all('explanation' in s for s in result['suggestions'])
    assert result['suggestions'][0]['score'] > 0.7

def test_suggest_meal_respects_remaining_macros():
    """Test that suggestions fit remaining macros"""
    # Setup: User has 400 cal, 30g protein remaining
    result = nutrition_agent.suggest_next_meal(user_id=1)

    # All suggestions should be <= 500 cal (with buffer)
    assert all(s['calories'] <= 500 for s in result['suggestions'])

def test_suggest_meal_considers_inventory():
    """Test that suggestions use available ingredients"""
    result = nutrition_agent.suggest_next_meal(user_id=1)

    # Should only suggest makeable recipes
    for suggestion in result['suggestions']:
        availability = inventory_service.check_recipe_availability(
            user_id=1,
            recipe_id=suggestion['recipe_id']
        )
        assert availability['can_make'] or availability['coverage_percentage'] >= 80
```

📚 CODE EXAMPLES

Example: Nutrition Agent Core

python

```python
# backend/app/agents/nutrition_agent.py

from typing import Dict, List, Any
from dataclasses import dataclass
from datetime import datetime

@dataclass
class NutritionContext:
    user_id: int
    current_goals: Dict
    daily_targets: Dict
    current_progress: Dict

class NutritionAgent:
    """Intelligence layer for nutrition coaching"""

    def __init__(self, db: Session):
        self.db = db
        self.consumption_service = ConsumptionService(db)
        self.inventory_service = InventoryService(db)
        self.planning_agent = PlanningAgent(db)

    async def suggest_next_meal(
        self,
        user_id: int,
        meal_type: str,
        context: Dict = None
    ) -> Dict[str, Any]:
        """
        Context-aware meal suggestion
        Orchestrates existing services + adds intelligence
        """
        # 1. Gather data from existing services
        daily_summary = self.consumption_service.get_today_summary(user_id)
        remaining = daily_summary['remaining_targets']

        makeable = self.inventory_service.get_makeable_recipes(user_id)

        user_goal = self._get_user_goal(user_id)
        goal_recipes = self.planning_agent.select_recipes_for_goal(
            goal=user_goal,
            count=20
        )
```

```
    # 2. Combine and score
    candidates = self._intersect_recipes(makeable, goal_recipes)

    scored = []
    for recipe in candidates:
        score = self._calculate_context_score(
            recipe=recipe,
            remaining_macros=remaining,
            context=context or {},
            meal_type=meal_type
        )

        explanation = self._generate_explanation(
            recipe=recipe,
            score=score,
            remaining=remaining,
            context=context
        )

        scored.append({
            "recipe_id": recipe['id'],
            "recipe_name": recipe['title'],
            "calories": recipe['macros_per_serving']['calories'],
            "protein_g": recipe['macros_per_serving']['protein_g'],
            "score": score['total'],
            "explanation": explanation
        })

    # 3. Return top 3
    scored.sort(key=lambda x: x['score'], reverse=True)
    return {"suggestions": scored[:3]}

def _calculate_context_score(
    self,
    recipe: Dict,
    remaining_macros: Dict,
    context: Dict,
    meal_type: str
) -> Dict[str, float]:
    """Calculate multi-factor score"""

    # Macro fit (30%)
    macro_score = self._score_macro_fit(recipe, remaining_macros)
```

```
    # Context match (25%)
    context_score = self._score_context(recipe, context, meal_type)

    # Inventory optimization (20%)
    inventory_score = self._score_inventory(recipe)

    # Goal alignment (15%)
    goal_score = self._score_goal_alignment(recipe)

    # Preference (10%)
    preference_score = self._score_preference(recipe, context)

    total = (
        macro_score * 0.30 +
        context_score * 0.25 +
        inventory_score * 0.20 +
        goal_score * 0.15 +
        preference_score * 0.10
    )

    return {
        "total": total,
        "breakdown": {
            "macro_fit": macro_score,
            "context": context_score,
            "inventory": inventory_score,
            "goal": goal_score,
            "preference": preference_score
        }
    }

def _generate_explanation(
    self,
    recipe: Dict,
    score: Dict,
    remaining: Dict,
    context: Dict
) -> str:
    """Generate human-readable explanation"""

    reasons = []

    # Top scoring factors
    breakdown = score['breakdown']
```

```
    top_factors = sorted(
        breakdown.items(),
        key=lambda x: x[1],
        reverse=True
    )[:2]

    # Macro fit
    if 'macro_fit' in [f[0] for f in top_factors]:
        protein = recipe['macros_per_serving']['protein_g']
        if protein >= 30:
            reasons.append(f"High protein ({protein}g) fits your remaining target")

    # Context
    if 'context' in [f[0] for f in top_factors]:
        if context.get('mood') == 'tired':
            reasons.append(f"Quick to make ({recipe['prep_time_min']} min) for tired evenings")

    # Inventory
    if 'inventory' in [f[0] for f in top_factors]:
        reasons.append("Uses ingredients expiring soon")

    return "Perfect because: " + ", ".join(reasons)
```

🔍 DEBUGGING GUIDE

Common Issues & Solutions

Issue 1: Nutrition Agent suggests recipes user can't make

python

```
    # Problem: Not checking inventory properly
    # Solution: Always filter through InventoryService first

    makeable = self.inventory_service.get_makeable_recipes(user_id)
    # Then score only these recipes
```

Issue 2: Suggestions don't fit remaining macros

python

```
# Problem: Not considering daily progress
# Solution: Always get today's summary first

daily_summary = self.consumption_service.get_today_summary(user_id)
remaining = daily_summary['remaining_targets']
# Filter recipes that fit within remaining + buffer
```

Issue 3: Explanations are generic

python

```
# Problem: Not using context data
# Solution: Parse context and use in explanation

if context.get('mood') == 'tired':
    explanation += "Quick and easy for tired evenings"
if context.get('weather') == 'cold':
    explanation += "Warm comfort food for cold weather"
```

📖 GLOSSARY

Orchestration: Coordinating multiple existing services to achieve a goal without duplicating their logic.

Intelligence Layer: Adding context-aware reasoning and explanations on top of existing calculations.

Context-Aware: Taking into account user's current situation (time, mood, weather) not just data.

Redundancy: Building functionality that already exists elsewhere in the codebase.

Enhancement: Improving existing functionality without replacing it.

🎓 LESSONS LEARNED

1. **Always audit existing code before building new features**
   - Saved 4 hours by identifying 70% redundancy
2. **Orchestration > Duplication**
   - Reusing existing services is faster and more maintainable
3. **Build for actual use cases, not hypothetical ones**
   - Educational content waits until WhatsApp Bot reveals what's needed
4. **Intelligence layer ≠ Calculation layer**
   - Nutrition Agent's value is in reasoning, not math
5. **Phased implementation reduces risk**
   - Build critical tool first (suggest_next_meal), iterate later

---

📞 CONTACTS & REFERENCES

Key Files to Reference:

- `backend/app/services/onboarding.py` - BMR/TDEE calculations
- `backend/app/services/consumption_services.py` - Daily tracking
- `backend/app/agents/tracking_agent.py` - Real-time updates
- `backend/app/workers/notification_worker.py` - Scheduled notifications
- `backend/app/agents/planning_agent.py` - Recipe selection

Sprint Documents:

- Detailed sprint document.docx - Original 14-day plan
- NutriLens AI Extended 25Day Development Sprint Plan.pdf - Extended plan
- day 6 Complete Implementation Summary.pdf - What's done

Testing Documents:

- COMPLETE_NOTIFICATION_TESTING_PLAN.md.pdf
- day 6 Complete EndtoEnd Testing Plan.pdf

📋 FINAL CHECKLIST

Before implementing Nutrition Agent, verify:

- ☐ Reviewed all existing services (Onboarding, Consumption, Planning, Inventory)
- ☐ Confirmed what functionality already exists
- ☐ Decided which 6 tools to build (vs original 10)
- ☐ Understood where tools will be used (WhatsApp, Frontend, Orchestrator)
- ☐ Planned phased implementation (Day 7, Day 9, Day 13+)
- ☐ Ready to reuse existing services instead of duplicating
- ☐ Have test strategy for suggest_next_meal()
- ☐ Know what NOT to build (BMR/TDEE, notifications, basic tracking)

Remember: Nutrition Agent is an intelligence layer, not a calculation engine.

END OF DOCUMENT

This document represents the complete analysis, decisions, and implementation guide for the Nutrition Agent based on Day 6 completion. Update this document if requirements change or new discoveries are made.