Ninjacart Category Agent System

Complete Technical Implementation Guide

Version: 1.0
Document Type: Technical Specification & Implementation Guide
Last Updated: October 2025
Target Audience: Product Managers, Data Scientists, Backend Engineers, Frontend Engineers, DevOps Engineers

Document Purpose: This comprehensive guide provides complete technical specifications, implementation details, code examples, and deployment instructions for building Ninjacart's AI-powered Category Agent System.

📋 Table of Contents

---

1. Executive Summary

1.1 System Overview

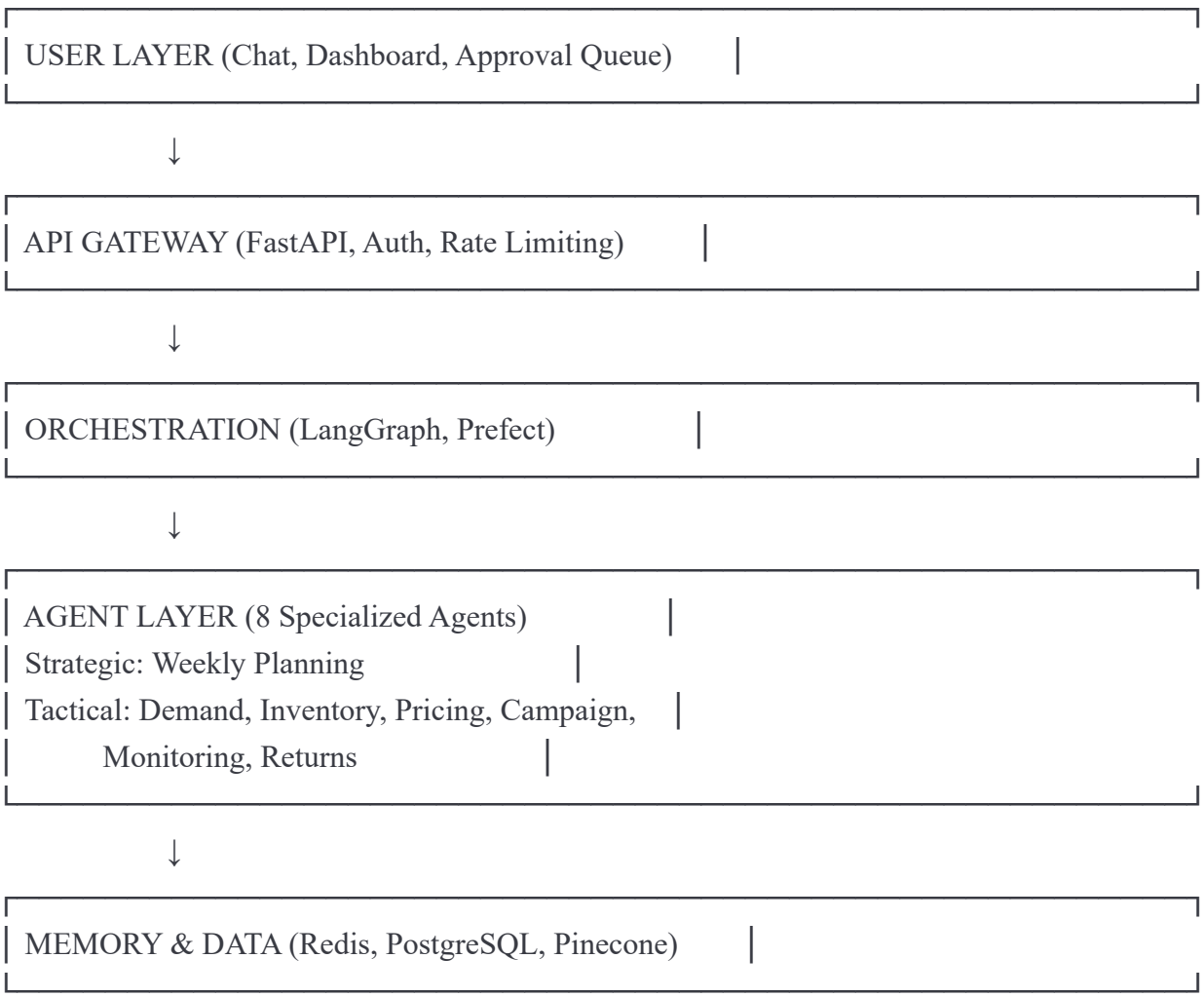
The Ninjacart Category Agent System is an AI-powered platform that automates category management decisions including:

- **Strategic Planning**: Weekly analysis and target setting (Every Saturday)
- **Daily Pricing**: Automated SP recommendations with explainable reasoning (Every morning 7 AM)
- **Demand Forecasting**: ML-based predictions (85-90% accuracy)
- **Campaign Management**: Intelligent segment targeting and ROI optimization
- **Real-time Monitoring**: Proactive interventions and alerts

1.2 Key Metrics & Goals

Metric	Current (Manual)	Target (Automated)	Impact
Pricing Decision Time	30-45 min/SKU	2 min/SKU	93% reduction
Forecast Accuracy	~70%	85-90%	+15-20pp improvement
Campaign Success Rate	60-65%	75-80%	+15pp improvement
Manual Effort	4-5 hours/day	1 hour/day	75% reduction
Revenue Protection	Baseline	+₹10-15L/day	Incremental

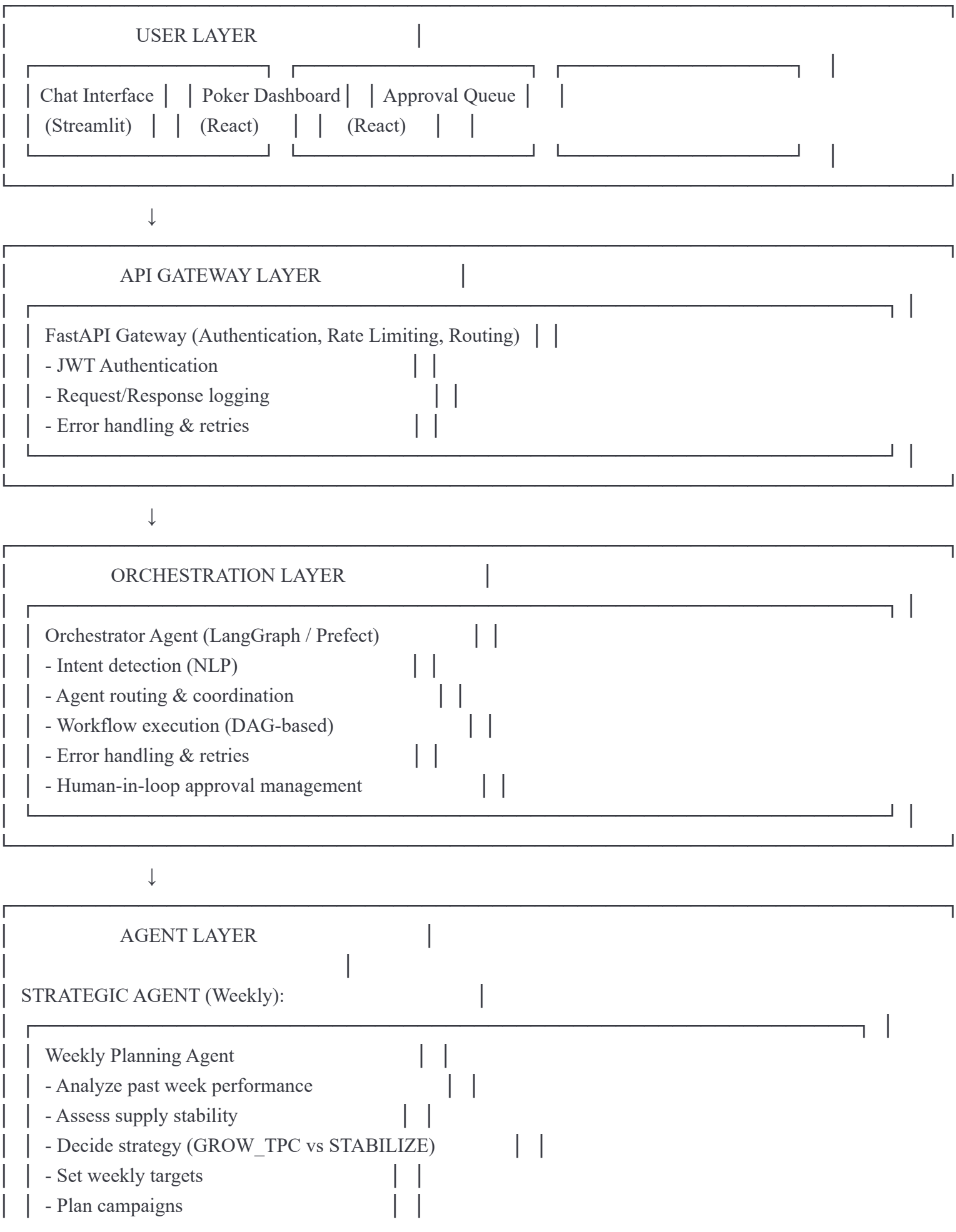
1.3 Architecture Layers

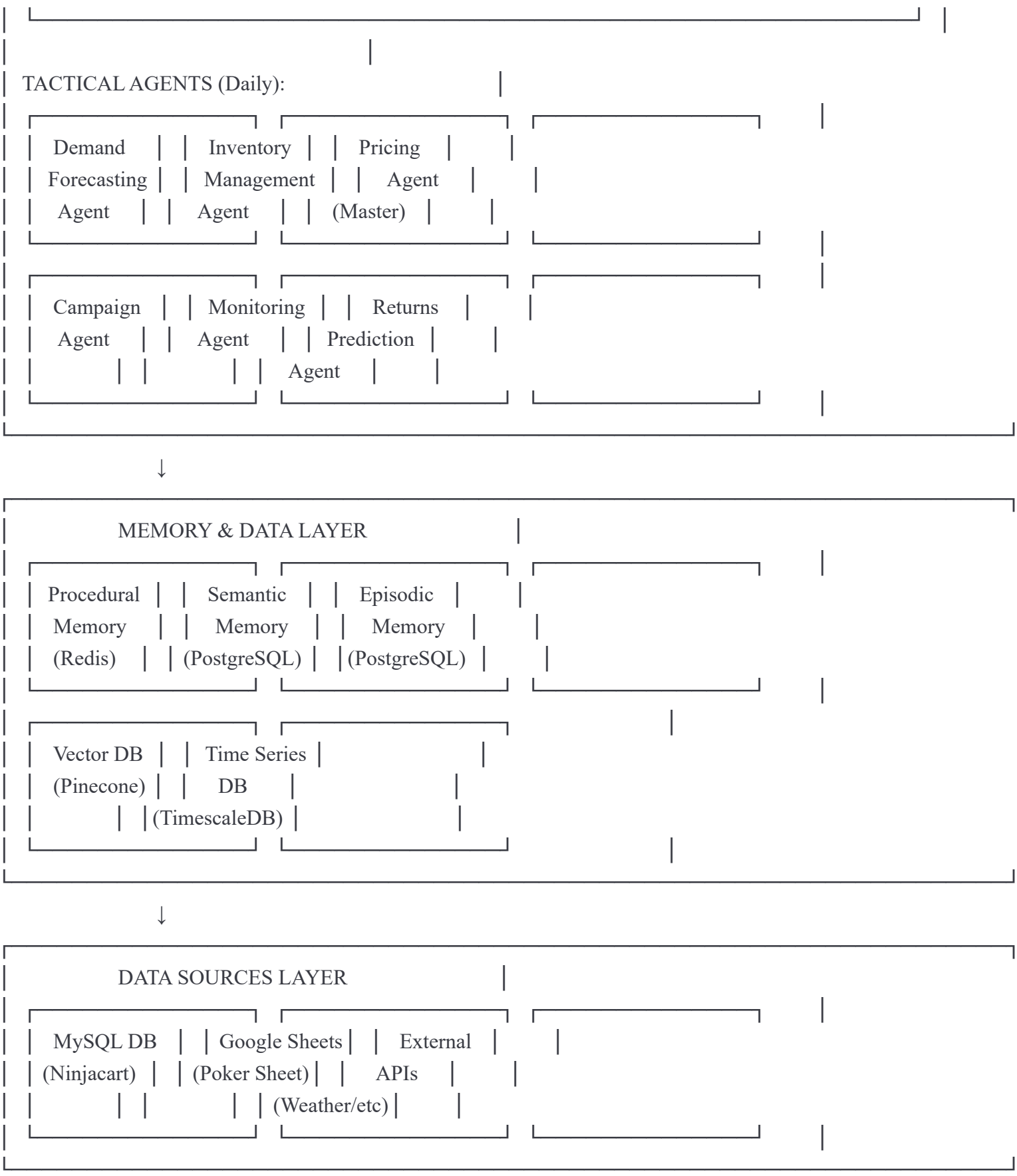


---

2. System Architecture Overview

2.1 High-Level Architecture Diagram


USER LAYER
Chat Interface (Streamlit)
Poker Dashboard (React)
Approval Queue (React)
API GATEWAY LAYER
FastAPI Gateway (Authentication, Rate Limiting, Routing)
- JWT Authentication
- Request/Response logging
- Error handling & retries
ORCHESTRATION LAYER
Orchestrator Agent (LangGraph / Prefect)
- Intent detection (NLP)
- Agent routing & coordination
- Workflow execution (DAG-based)
- Error handling & retries
- Human-in-loop approval management
AGENT LAYER
STRATEGIC AGENT (Weekly):
Weekly Planning Agent
- Analyze past week performance
- Assess supply stability
- Decide strategy (GROW_TPC vs STABILIZE)
- Set weekly targets
- Plan campaigns


TACTICAL AGENTS (Daily):
Demand Forecasting Agent
Inventory Management Agent
Pricing Agent (Master)
Campaign Agent
Monitoring Agent
Returns Prediction Agent
MEMORY & DATA LAYER
Procedural Memory (Redis)
Semantic Memory (PostgreSQL)
Episodic Memory (PostgreSQL)
Vector DB (Pinecone)
Time Series DB (TimescaleDB)
DATA SOURCES LAYER
MySQL DB (Ninjacart)
Google Sheets (Poker Sheet)
External APIs (Weather/etc)

2.2 Data Flow Patterns

Request-Response Flow (User-Initiated):

User Request → API Gateway → Orchestrator →
Agent Selection → Agent Execution → Memory Access →
Result Compilation → Response to User

Scheduled Flow (Automated):

Cron Trigger → Orchestrator → Agent Execution →
Memory Update → Notification System

Learning Loop:

Agent Decision → Execution → Outcome Measurement →
Episodic Memory → Vector DB → Future RAG Retrieval

---

3. Technology Stack

3.1 Core Technologies

Component	Technology	Version	Justification
Primary Language	Python	3.11+	Mature ML ecosystem, async support, type hints
LLM	OpenAI GPT-4 / Claude Sonnet	Latest	SOTA reasoning, low hallucination rate
Agent Framework	LangChain + LangGraph	0.1.0 / 0.0.20	Graph-based workflows, observability
Orchestration	Prefect	2.14.0	Workflow management, scheduling, monitoring
Backend API	FastAPI	0.104.1	High performance, async, auto OpenAPI docs
Cache/Config	Redis	7.x	In-memory speed, pub/sub, TTL support
Database	PostgreSQL	15.x	ACID compliance, JSONB, mature
Time Series	TimescaleDB	Latest	Optimized time-series queries
Vector DB	Pinecone	Latest	Managed, scalable, low latency
ML Framework	Scikit-learn, XGBoost, Prophet	Latest	Industry standard, proven accuracy
Frontend	React + Streamlit	18.x / Latest	Complex UI (React), Rapid prototyping (Streamlit)
Monitoring	Prometheus + Grafana	Latest	Open-source, extensive integrations
Logging	ELK Stack	Latest	Centralized logging, search, analysis

3.2 Python Dependencies

requirements.txt:

txt

```
# LLM & Agent Framework
langchain==0.1.0
langgraph==0.0.20
openai==1.6.1
anthropic==0.8.1

# Orchestration & Workflow
prefect==2.14.0
celery==5.3.4
redis==5.0.1

# API Framework
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
python-multipart==0.0.6

# Database & ORM
sqlalchemy==2.0.23
alembic==1.13.0
psycopg2-binary==2.9.9
asyncpg==0.29.0

# Vector Database
pinecone-client==2.2.4

# ML & Data Science
scikit-learn==1.3.2
xgboost==2.0.2
prophet==1.1.5
pandas==2.1.4
numpy==1.26.2
scipy==1.11.4

# Google Sheets Integration
google-api-python-client==2.111.0
google-auth==2.25.2
google-auth-oauthlib==1.2.0
google-auth-httplib2==0.2.0

# HTTP & API Clients
```

```
httpx==0.25.2
requests==2.31.0

# Utilities
python-dotenv==1.0.0
tenacity==8.2.3  # Retry logic
pyyaml==6.0.1
python-json-logger==2.0.7

# Monitoring & Metrics
prometheus-client==0.19.0
opentelemetry-api==1.21.0
opentelemetry-sdk==1.21.0

# Testing
pytest==7.4.3
pytest-asyncio==0.21.1
pytest-cov==4.1.0
httpx==0.25.2  # For testing FastAPI

# Security
python-jose[cryptography]==3.3.0
passlib[bcrypt]==1.7.4
```

---

4. Infrastructure Setup

4.1 Development Environment

Prerequisites:

- Python 3.11+
- Docker & Docker Compose
- Git
- 16GB RAM minimum
- (Optional) GPU for local LLM testing

4.2 Docker Compose Setup

docker-compose.yml:

yaml

```yaml
version: '3.8'

services:
  # PostgreSQL with TimescaleDB
  postgres:
    image: timescale/timescaledb:latest-pg15
    container_name: category_agent_db
    environment:
      POSTGRES_DB: category_agent
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_INITDB_ARGS: "-c shared_preload_libraries=timescaledb"
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init_scripts:/docker-entrypoint-initdb.d
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U admin -d category_agent"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Redis
  redis:
    image: redis:7-alpine
    container_name: category_agent_redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    command: redis-server --appendonly yes
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 3s
      retries: 5

  # FastAPI Backend
  api:
    build:
```

```
    context: ./backend
    dockerfile: Dockerfile
  container_name: category_agent_api
  ports:
    - "8000:8000"
  environment:
    - DATABASE_URL=postgresql+asyncpg://admin:${DB_PASSWORD}@postgres:5432/category_agent
    - REDIS_URL=redis://redis:6379
    - OPENAI_API_KEY=${OPENAI_API_KEY}
    - PINECONE_API_KEY=${PINECONE_API_KEY}
    - PINECONE_ENV=${PINECONE_ENV}
  depends_on:
    postgres:
      condition: service_healthy
    redis:
      condition: service_healthy
  volumes:
    - ./backend:/app
    - ./models:/app/models
  command: uvicorn main:app --host 0.0.0.0 --port 8000 --reload

# Streamlit Frontend
streamlit:
  build:
    context: ./frontend
    dockerfile: Dockerfile.streamlit
  container_name: category_agent_streamlit
  ports:
    - "8501:8501"
  environment:
    - API_URL=http://api:8000
  depends_on:
    - api
  volumes:
    - ./frontend/streamlit_app:/app

# React Dashboard
react_dashboard:
  build:
    context: ./frontend/react_dashboard
    dockerfile: Dockerfile
```

```
  container_name: category_agent_dashboard
  ports:
    - "3000:3000"
  environment:
    - REACT_APP_API_URL=http://localhost:8000
  volumes:
    - ./frontend/react_dashboard:/app
    - /app/node_modules

# Prefect Server (Workflow Orchestration)
prefect_server:
  image: prefecthq/prefect:2.14.0-python3.11
  container_name: category_agent_prefect
  ports:
    - "4200:4200"
  environment:
    - PREFECT_API_URL=http://0.0.0.0:4200/api
    - PREFECT_SERVER_API_HOST=0.0.0.0
  command: prefect server start
  volumes:
    - prefect_data:/root/.prefect

# Prometheus (Metrics)
prometheus:
  image: prom/prometheus:latest
  container_name: category_agent_prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    - prometheus_data:/prometheus
  command:
    - '--config.file=/etc/prometheus/prometheus.yml'
    - '--storage.tsdb.path=/prometheus'

# Grafana (Visualization)
grafana:
  image: grafana/grafana:latest
  container_name: category_agent_grafana
  ports:
    - "3001:3000"
```

```
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}
    volumes:
      - grafana_data:/var/lib/grafana
      - ./monitoring/grafana/dashboards:/etc/grafana/provisioning/dashboards
    depends_on:
      - prometheus

volumes:
  postgres_data:
  redis_data:
  prefect_data:
  prometheus_data:
  grafana_data:
```

4.3 Environment Configuration

.env:

bash

```
# Database
DB_PASSWORD=your_secure_password_here
DATABASE_URL=postgresql+asyncpg://admin:${DB_PASSWORD}@localhost:5432/category_agent

# Redis
REDIS_URL=redis://localhost:6379

# LLM APIs
OPENAI_API_KEY=sk-...
ANTHROPIC_API_KEY=sk-ant-...

# Vector DB
PINECONE_API_KEY=...
PINECONE_ENV=us-west1-gcp
PINECONE_INDEX=category-agent-episodes

# Ninjacart Data Sources
NINJACART_DB_HOST=your-mysql-host.rds.amazonaws.com
NINJACART_DB_PORT=3306
NINJACART_DB_USER=readonly_user
NINJACART_DB_PASSWORD=...
NINJACART_DB_NAME=ninjacart_production

# Google Sheets
GOOGLE_SHEETS_CREDENTIALS_PATH=/app/credentials/google_credentials.json
POKER_SHEET_ID=130Q4yw1nfwk9Mj839NbAdTb3dtdCXVRxf0WNpYhYQd8

# External APIs
WEATHER_API_KEY=...
WEATHER_API_URL=https://api.openweathermap.org/data/2.5

# JWT Authentication
JWT_SECRET_KEY=your_jwt_secret_key_here
JWT_ALGORITHM=HS256
JWT_ACCESS_TOKEN_EXPIRE_MINUTES=1440

# Monitoring
PROMETHEUS_PORT=9090
GRAFANA_PORT=3001
GRAFANA_PASSWORD=admin
```

```
# Application
APP_ENV=development  # development, staging, production
LOG_LEVEL=INFO
```

4.4 Initialization Scripts

init_scripts/01_create_extensions.sql:

sql

```
-- Enable TimescaleDB extension
CREATE EXTENSION IF NOT EXISTS timescaledb;

-- Enable UUID generation
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Enable pg_stat_statements for query performance monitoring
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
```

---

5. Data Layer Implementation

5.1 Complete Database Schema

schema.sql:

sql

```sql
-- ============================================================================
-- SEMANTIC MEMORY: Domain Knowledge & SKU Characteristics
-- ============================================================================

CREATE TABLE sku_knowledge (
    sku_id VARCHAR(50) PRIMARY KEY,
    sku_name VARCHAR(255) NOT NULL,
    category VARCHAR(100) NOT NULL,

    -- Target metrics
    daily_penetration_target_min DECIMAL(5,2),
    daily_penetration_target_max DECIMAL(5,2),
    weekly_penetration_target DECIMAL(5,2),
    gm_target_min DECIMAL(8,2),
    gm_target_max DECIMAL(8,2),
    tpo_target DECIMAL(8,2),

    -- SKU Characteristics
    crop_cycle_weeks INTEGER,
    shelf_life_days INTEGER,
    peak_season VARCHAR[] DEFAULT '{}',
    price_elasticity DECIMAL(4,2),
    festival_lift_pct DECIMAL(5,2),
    rain_impact_pct DECIMAL(5,2),
    quality_sensitive BOOLEAN DEFAULT TRUE,

    -- Formulas (JSON for flexibility)
    pricing_formulas JSONB DEFAULT '{}',
    margin_rules JSONB DEFAULT '{}',

    -- Metadata
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_sku_category ON sku_knowledge(category);

-- ============================================================================
-- EPISODIC MEMORY: Historical Decisions & Outcomes
-- ============================================================================
```

```
CREATE TABLE pricing_episodes (
    episode_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    sku_id VARCHAR(50) REFERENCES sku_knowledge(sku_id),
    episode_date DATE NOT NULL,

    -- Context
    wsp DECIMAL(8,2),
    pp DECIMAL(8,2),
    inventory_kg DECIMAL(10,2),
    inventory_surplus_pct DECIMAL(5,2),
    pm DECIMAL(8,2),
    supply_status VARCHAR(20), -- SURPLUS, BALANCED, DEFICIT
    day_of_week INTEGER,
    is_festival BOOLEAN,
    weather_condition VARCHAR(50),

    -- Decision
    sp_recommended DECIMAL(8,2),
    sp_approved DECIMAL(8,2),
    sp_reasoning TEXT,
    confidence DECIMAL(4,3),
    human_approved BOOLEAN,
    approved_by VARCHAR(100),
    approval_timestamp TIMESTAMP,

    -- Expected Outcomes
    tpc_predicted DECIMAL(10,2),
    penetration_predicted DECIMAL(5,2),
    gm_predicted DECIMAL(8,2),

    -- Actual Outcomes (filled after day completes)
    tpc_actual DECIMAL(10,2),
    penetration_actual DECIMAL(5,2),
    gm_actual DECIMAL(8,2),
    orders_actual INTEGER,

    -- Learning Metrics
    tpc_error_pct DECIMAL(5,2),
    penetration_error_pp DECIMAL(5,2),
    gm_error_pct DECIMAL(5,2),
    overall_accuracy DECIMAL(4,3),
```

```
    learning_notes TEXT,

    -- Metadata
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_episode_sku_date ON pricing_episodes(sku_id, episode_date);
CREATE INDEX idx_episode_date ON pricing_episodes(episode_date);
CREATE INDEX idx_episode_supply_status ON pricing_episodes(supply_status);

-- ============================================================================
-- WEEKLY PLANS: Strategic Direction
-- ============================================================================

CREATE TABLE weekly_plans (
    plan_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    sku_id VARCHAR(50) REFERENCES sku_knowledge(sku_id),
    week_start_date DATE NOT NULL,
    week_end_date DATE NOT NULL,
    week_number INTEGER, -- ISO week number

    -- Previous Week Performance
    prev_week_tpc_actual DECIMAL(10,2),
    prev_week_tpc_target DECIMAL(10,2),
    prev_week_achievement_pct DECIMAL(5,2),
    prev_week_penetration_actual DECIMAL(5,2),
    prev_week_penetration_target DECIMAL(5,2),
    prev_week_gm_avg DECIMAL(8,2),

    -- Supply Assessment
    supply_stability VARCHAR(20), -- STABLE, UNSTABLE
    wsp_cv_pct DECIMAL(5,2),
    wsp_mean DECIMAL(8,2),
    wsp_range_min DECIMAL(8,2),
    wsp_range_max DECIMAL(8,2),
    quality_score DECIMAL(4,3),
    qqc_rejection_avg_pct DECIMAL(5,2),
    vendor_pipeline_count INTEGER,
    vendor_reliability_score DECIMAL(4,3),
    next_week_risk_score DECIMAL(4,3),
```

```
next_week_risk_level VARCHAR(20), -- LOW, MODERATE, HIGH

-- Strategic Decision
strategy VARCHAR(50), -- GROW_TPC, STABILIZE_TPC_IMPROVE_MARGINS
strategy_focus VARCHAR(50), -- VOLUME_GROWTH, MARGIN_PRESERVATION
strategy_rationale TEXT,
margin_approach VARCHAR(50),

-- Historical Benchmark
similar_weeks_count INTEGER,
mode_tpc_kg_per_day DECIMAL(10,2),
median_tpc_kg_per_day DECIMAL(10,2),
p75_tpc_kg_per_day DECIMAL(10,2),
safe_baseline_kg_per_day DECIMAL(10,2),

-- Next Week Targets
daily_tpc_target_kg DECIMAL(10,2),
weekly_tpc_target_kg DECIMAL(10,2),
growth_vs_baseline_pct DECIMAL(5,2),
penetration_target_pct DECIMAL(5,2),
tpo_target_kg DECIMAL(8,2),
gm_target_min DECIMAL(8,2),
gm_target_max DECIMAL(8,2),
gm_preference VARCHAR(50),
pm_target_min DECIMAL(8,2),
pm_target_max DECIMAL(8,2),
sm_target_min DECIMAL(8,2),
sm_target_max DECIMAL(8,2),

-- Penetration Gap Analysis
ideal_penetration_pct DECIMAL(5,2),
current_penetration_pct DECIMAL(5,2),
penetration_gap_pp DECIMAL(5,2),
campaign_required BOOLEAN,

-- Planned Campaign (JSONB for flexibility)
planned_campaign JSONB,

-- Success Checkpoints
checkpoints JSONB,
```

```
    -- Approval
    approval_status VARCHAR(20), -- PENDING, APPROVED, REJECTED, MODIFIED
    approved_by VARCHAR(100),
    approved_at TIMESTAMP,
    rejection_reason TEXT,

    -- Metadata
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_weekly_plan_sku_week ON weekly_plans(sku_id, week_start_date);
CREATE INDEX idx_weekly_plan_week ON weekly_plans(week_start_date);
CREATE INDEX idx_weekly_plan_approval ON weekly_plans(approval_status);

-- ============================================================================
-- CAMPAIGN HISTORY: Campaign Performance
-- ============================================================================

CREATE TABLE campaign_episodes (
    campaign_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    sku_id VARCHAR(50) REFERENCES sku_knowledge(sku_id),
    campaign_date DATE NOT NULL,
    campaign_type VARCHAR(50), -- SKU_PROMOTION, CART_DISCOUNT, MILESTONE, SLAB_PRICING
    campaign_trigger VARCHAR(50), -- PENETRATION_DROP, EXCESS_INVENTORY, PLANNED_WEEKLY

    -- Segment Details
    segment_name VARCHAR(100),
    segment_size INTEGER,
    segment_definition JSONB,

    -- Discount Structure
    discount_type VARCHAR(50), -- FLAT, PERCENTAGE, SLAB
    discount_structure JSONB,

    -- Timing
    start_timestamp TIMESTAMP,
    end_timestamp TIMESTAMP,
    duration_hours INTEGER,

    -- Predictions
```

```
    conversion_rate_predicted DECIMAL(5,4),
    customers_converting_predicted INTEGER,
    avg_order_size_predicted DECIMAL(8,2),
    incremental_tpc_predicted DECIMAL(10,2),
    penetration_lift_predicted_pp DECIMAL(5,2),
    budget_allocated DECIMAL(10,2),
    expected_roi DECIMAL(8,2),

    -- Actuals
    conversion_rate_actual DECIMAL(5,4),
    customers_converting_actual INTEGER,
    avg_order_size_actual DECIMAL(8,2),
    incremental_tpc_actual DECIMAL(10,2),
    penetration_lift_actual_pp DECIMAL(5,2),
    discount_burn_actual DECIMAL(10,2),
    incremental_revenue_actual DECIMAL(10,2),
    actual_roi DECIMAL(8,2),

    -- Performance Metrics
    conversion_prediction_accuracy DECIMAL(5,2),
    tpc_prediction_accuracy DECIMAL(5,2),
    roi_prediction_accuracy DECIMAL(5,2),
    overall_success_score DECIMAL(4,3),

    -- Learning
    learning_notes TEXT,

    -- Metadata
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_campaign_sku_date ON campaign_episodes(sku_id, campaign_date);
CREATE INDEX idx_campaign_type ON campaign_episodes(campaign_type);
CREATE INDEX idx_campaign_trigger ON campaign_episodes(campaign_trigger);

-- ============================================================================
-- TIME SERIES DATA: Daily Actuals (TimescaleDB Hypertable)
-- ============================================================================

CREATE TABLE daily_actuals (
```

```
    ts TIMESTAMPTZ NOT NULL,
    sku_id VARCHAR(50) NOT NULL,

    -- Sales Metrics
    orders_count INTEGER,
    customers_count INTEGER,
    tpc DECIMAL(10,2),
    tpo DECIMAL(8,2),
    penetration DECIMAL(5,2),

    -- Pricing
    wsp DECIMAL(8,2),
    pp DECIMAL(8,2),
    sp DECIMAL(8,2),

    -- Margins
    pm DECIMAL(8,2),
    sm DECIMAL(8,2),
    cm DECIMAL(8,2),
    gm DECIMAL(8,2),

    -- Inventory
    inventory_kg DECIMAL(10,2),
    supply_status VARCHAR(20),
    days_of_inventory DECIMAL(4,2),

    -- Quality
    qqc_rejection_pct DECIMAL(5,2),
    otpr_pct DECIMAL(5,2),

    -- External Factors
    is_festival BOOLEAN,
    is_weekend BOOLEAN,
    weather_condition VARCHAR(50),
    temperature DECIMAL(4,1),

    PRIMARY KEY (ts, sku_id)
);

-- Convert to hypertable (TimescaleDB specific)
SELECT create_hypertable('daily_actuals', 'ts',
```

```
    chunk_time_interval => INTERVAL '7 days',
    if_not_exists => TRUE
);

-- Create indexes
CREATE INDEX idx_daily_sku ON daily_actuals(sku_id, ts DESC);
CREATE INDEX idx_daily_date ON daily_actuals(ts DESC);

-- ============================================================================
-- DEMAND FORECASTS
-- ============================================================================

CREATE TABLE demand_forecasts (
    forecast_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    sku_id VARCHAR(50) NOT NULL,
    forecast_date DATE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Model Metadata
    model_version VARCHAR(50),
    model_type VARCHAR(50), -- XGBOOST, PROPHET, ENSEMBLE

    -- Base Forecast
    demand_forecast_kg DECIMAL(10,2),
    confidence DECIMAL(4,3),

    -- Uncertainty Bands
    p50_kg DECIMAL(10,2),
    p75_kg DECIMAL(10,2),
    p90_kg DECIMAL(10,2),

    -- Segment Breakdown
    retail_forecast_kg DECIMAL(10,2),
    bulk_forecast_kg DECIMAL(10,2),

    -- Features Used (for debugging)
    features_snapshot JSONB,

    -- Adjustments Applied
    adjustments_applied JSONB,
```

```
    -- Actuals (filled post-day)
    actual_demand_kg DECIMAL(10,2),
    forecast_error_pct DECIMAL(5,2),
    forecast_error_kg DECIMAL(10,2),

    UNIQUE (sku_id, forecast_date, created_at)
);

CREATE INDEX idx_forecast_sku_date ON demand_forecasts(sku_id, forecast_date);
CREATE INDEX idx_forecast_date ON demand_forecasts(forecast_date);

-- ============================================================================
-- AGENT EXECUTION LOGS
-- ============================================================================

CREATE TABLE agent_execution_logs (
    log_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    agent_name VARCHAR(100) NOT NULL,
    execution_type VARCHAR(50), -- SCHEDULED, USER_REQUEST, TRIGGERED
    trigger_source VARCHAR(100),

    -- Execution Details
    started_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP,
    duration_seconds INTEGER,
    status VARCHAR(20), -- SUCCESS, FAILED, PARTIAL, TIMEOUT

    -- Input/Output
    input_params JSONB,
    output_result JSONB,
    error_message TEXT,
    error_stack_trace TEXT,

    -- Resource Usage
    llm_calls_count INTEGER DEFAULT 0,
    llm_tokens_used INTEGER DEFAULT 0,
    llm_cost_usd DECIMAL(10,4),
    db_queries_count INTEGER DEFAULT 0,

    -- Metadata
    environment VARCHAR(20), -- DEVELOPMENT, STAGING, PRODUCTION
```

```
    version VARCHAR(50)
);

CREATE INDEX idx_agent_log_name_time ON agent_execution_logs(agent_name, started_at DESC);
CREATE INDEX idx_agent_log_status ON agent_execution_logs(status);
CREATE INDEX idx_agent_log_time ON agent_execution_logs(started_at DESC);

-- ============================================================================
-- USER FEEDBACK & APPROVALS
-- ============================================================================

CREATE TABLE user_approvals (
    approval_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    user_id VARCHAR(100) NOT NULL,
    user_name VARCHAR(255),

    -- What's being approved
    entity_type VARCHAR(50), -- PRICING_DECISION, WEEKLY_PLAN, CAMPAIGN
    entity_id UUID NOT NULL,

    -- Recommendation
    recommended_action TEXT,
    recommended_reasoning TEXT,
    confidence DECIMAL(4,3),

    -- User Decision
    decision VARCHAR(20), -- APPROVED, REJECTED, MODIFIED
    modified_values JSONB,
    user_reasoning TEXT,

    -- Timing
    requested_at TIMESTAMP NOT NULL,
    responded_at TIMESTAMP,
    response_time_seconds INTEGER,

    -- Learning
    feedback_quality_score DECIMAL(4,3),
    incorporated_into_learning BOOLEAN DEFAULT FALSE
);

CREATE INDEX idx_approval_user ON user_approvals(user_id, requested_at DESC);
```

```
CREATE INDEX idx_approval_entity ON user_approvals(entity_type, entity_id);
CREATE INDEX idx_approval_status ON user_approvals(decision);

-- ============================================================================
-- TRIGGERS FOR UPDATED_AT
-- ============================================================================

CREATE OR REPLACE FUNCTION update_updated_at_column()
RETURNS TRIGGER AS $$
BEGIN
   NEW.updated_at = CURRENT_TIMESTAMP;
   RETURN NEW;
END;
$$ language 'plpgsql';

CREATE TRIGGER update_sku_knowledge_updated_at BEFORE UPDATE ON sku_knowledge
   FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_pricing_episodes_updated_at BEFORE UPDATE ON pricing_episodes
   FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_weekly_plans_updated_at BEFORE UPDATE ON weekly_plans
   FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_campaign_episodes_updated_at BEFORE UPDATE ON campaign_episodes
   FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();
```

5.2 Data Ingestion Pipeline

etl/ninjacart_sync.py:

python

```
import pandas as pd
from sqlalchemy import create_engine
from datetime import datetime, timedelta
import logging
from typing import Dict, Optional

logger = logging.getLogger(__name__)

class NinjacartDataSync:
    """
    ETL pipeline for syncing data from Ninjacart MySQL to PostgreSQL.

    Purpose: Keep PostgreSQL data layer updated with latest operational data

    Schedule: Runs daily at 6:00 AM (syncs previous day's data)

    Process:
      1. Connect to both databases
      2. Extract data from Ninjacart MySQL
      3. Transform to match PostgreSQL schema
      4. Load into PostgreSQL (upsert)
      5. Validate data quality
    """

    def __init__(self, source_db_url: str, target_db_url: str):
        self.source_engine = create_engine(source_db_url)
        self.target_engine = create_engine(target_db_url)
        self.logger = logging.getLogger(__name__)

    def sync_daily_actuals(self, date: datetime.date) -> Dict:
        """
        Sync daily actuals from Ninjacart MySQL to PostgreSQL.

        Input: date (datetime.date) - Date to sync
        Output: Dict with sync stats

        Process:
          1. Query Ninjacart MySQL for all daily metrics
          2. Calculate derived metrics (penetration, margins, etc.)
          3. Upsert into PostgreSQL daily_actuals table
          4. Return sync statistics
```

```
"""
self.logger.info(f"Syncing daily actuals for {date}")

try:
    # Query Ninjacart MySQL
    query = f"""
    SELECT
        DATE(so.created_at) as date,
        s.sku_id,
        s.sku_name,

        -- Sales metrics
        COUNT(DISTINCT so.customer_id) as customers_count,
        COUNT(DISTINCT so.order_id) as orders_count,
        SUM(soi.quantity_kg) as tpc,
        AVG(soi.quantity_kg) as tpo,

        -- Pricing
        AVG(soi.sale_price) as sp,
        AVG(pi.purchase_price) as pp,
        AVG(m.wholesale_price) as wsp,

        -- Inventory
        MAX(inv.current_stock_kg) as inventory_kg,

        -- Quality
        AVG(qc.rejection_rate) as qqc_rejection_pct,
        AVG(r.return_rate) as otpr_pct,

        -- External factors
        MAX(c.is_festival) as is_festival,
        MAX(w.temperature) as temperature,
        MAX(w.weather_condition) as weather_condition

    FROM sale_orders so
    JOIN sale_order_items soi ON so.order_id = soi.order_id
    JOIN skus s ON soi.sku_id = s.sku_id
    LEFT JOIN procurement_invoices pi ON s.sku_id = pi.sku_id
        AND DATE(pi.created_at) = '{date}'
    LEFT JOIN market_prices m ON s.sku_id = m.sku_id
        AND DATE(m.created_at) = '{date}'
```

```
LEFT JOIN inventory inv ON s.sku_id = inv.sku_id
    AND DATE(inv.updated_at) = '{date}'
LEFT JOIN quality_checks qc ON s.sku_id = qc.sku_id
    AND DATE(qc.created_at) = '{date}'
LEFT JOIN returns r ON so.order_id = r.order_id
LEFT JOIN calendar c ON DATE(c.date) = '{date}'
LEFT JOIN weather w ON DATE(w.date) = '{date}'

WHERE DATE(so.created_at) = '{date}'
GROUP BY DATE(so.created_at), s.sku_id, s.sku_name
"""

df = pd.read_sql(query, self.source_engine)

if df.empty:
    self.logger.warning(f"No data found for {date}")
    return {"status": "no_data", "rows_synced": 0}

# Transform
df['ts'] = pd.to_datetime(df['date'])
df['is_weekend'] = df['ts'].dt.dayofweek >= 5

# Calculate penetration
total_active_customers = self._get_total_active_customers(date)
df['penetration'] = (df['customers_count'] / total_active_customers) * 100

# Calculate margins
df['pm'] = df['wsp'] - df['pp']
df['sm'] = df['sp'] - df['pp']
df['cm'] = df['wsp'] - df['wsp'] * 0.10  # 10% commission
df['gm'] = df['pm'] + df['sm'] - df['cm']

# Supply status
df['supply_status'] = df.apply(self._determine_supply_status, axis=1)
df['days_of_inventory'] = df['inventory_kg'] / df['tpc']

# Select columns for PostgreSQL
columns_to_sync = [
    'ts', 'sku_id', 'orders_count', 'customers_count', 'tpc', 'tpo',
    'penetration', 'wsp', 'pp', 'sp', 'pm', 'sm', 'cm', 'gm',
    'inventory_kg', 'supply_status', 'days_of_inventory',
```

```
            'qqc_rejection_pct', 'otpr_pct', 'is_festival', 'is_weekend',
            'weather_condition', 'temperature'
        ]

        df_final = df[columns_to_sync]

        # Upsert to PostgreSQL
        rows_synced = self._upsert_to_postgres(df_final, 'daily_actuals')

        self.logger.info(f"Successfully synced {rows_synced} rows for {date}")

        return {
            "status": "success",
            "date": str(date),
            "rows_synced": rows_synced,
            "skus_processed": len(df_final['sku_id'].unique())
        }

    except Exception as e:
        self.logger.error(f"Error syncing data for {date}: {str(e)}")
        return {
            "status": "error",
            "date": str(date),
            "error": str(e)
        }

def _get_total_active_customers(self, date: datetime.date) -> int:
    """Calculate total active customers for penetration"""
    query = f"""
    SELECT COUNT(DISTINCT customer_id) as active_customers
    FROM sale_orders
    WHERE DATE(created_at) >= '{date - timedelta(days=30)}'
    AND DATE(created_at) <= '{date}'
    """
    result = pd.read_sql(query, self.source_engine)
    return int(result['active_customers'].iloc[0])

def _determine_supply_status(self, row) -> str:
    """Determine supply status based on inventory and demand"""
    # Simplified logic - enhance based on your business rules
    if pd.isna(row['inventory_kg']) or pd.isna(row['tpc']):
```

```
            return 'UNKNOWN'

        ratio = row['inventory_kg'] / row['tpc'] if row['tpc'] > 0 else 0

        if ratio > 1.5:
            return 'SURPLUS'
        elif ratio < 0.8:
            return 'DEFICIT'
        else:
            return 'BALANCED'

    def _upsert_to_postgres(self, df: pd.DataFrame, table_name: str) -> int:
        """Upsert data to PostgreSQL"""
        from sqlalchemy.dialects.postgresql import insert

        # Convert DataFrame to list of dicts
        records = df.to_dict('records')

        # Upsert
        rows_affected = 0
        with self.target_engine.begin() as conn:
            for record in records:
                stmt = insert(table_name).values(**record)
                stmt = stmt.on_conflict_do_update(
                    index_elements=['ts', 'sku_id'],
                    set_=record
                )
                result = conn.execute(stmt)
                rows_affected += result.rowcount

        return rows_affected
```

Scheduled ETL Job (`etl/scheduler.py`):

python

```python
from prefect import flow, task
from prefect.schedules import CronSchedule
from datetime import datetime, timedelta
import os

@task
def sync_yesterday_data():
    """Sync yesterday's data every morning at 6 AM"""
    yesterday = datetime.now().date() - timedelta(days=1)

    sync = NinjacartDataSync(
        source_db_url=os.getenv('NINJACART_DB_URL'),
        target_db_url=os.getenv('DATABASE_URL')
    )

    result = sync.sync_daily_actuals(yesterday)
    return result

@flow(name="daily_data_sync")
def daily_sync_flow():
    """Main flow for daily data synchronization"""
    result = sync_yesterday_data()

    if result['status'] == 'success':
        print(f"✅ Data sync completed: {result}")
    else:
        print(f"❌ Data sync failed: {result}")
        # TODO: Send alert to team

# Schedule: Every day at 6:00 AM IST
schedule = CronSchedule(cron="0 6 * * *", timezone="Asia/Kolkata")

if __name__ == "__main__":
    daily_sync_flow()
```

---

6. Memory Systems Implementation

6.1 Procedural Memory (Redis)

memory/procedural_memory.py:


python

```python
import redis
import json
from typing import Dict, Any, Optional, List
import os
from datetime import datetime

class ProceduralMemory:
    """
    Stores and retrieves business rules, pricing workflows, configurations.

    Storage: Redis (in-memory for fast access)
    TTL: Rules persist forever, Weekly targets expire after 7 days

    Key Patterns:
      - rules:{sku_id} → Pricing rules
      - weekly_targets:{sku_id}:{week} → Weekly targets
      - workflow:{workflow_name} → Execution steps
    """

    def __init__(self, redis_url: Optional[str] = None):
        self.redis_client = redis.from_url(
            redis_url or os.getenv('REDIS_URL'),
            decode_responses=True
        )
        self._ensure_rules_loaded()

    def _ensure_rules_loaded(self):
        """Load initial rules from config file if not exists"""
        if not self.redis_client.exists('rules:TOMATO_LOCAL'):
            self._load_initial_rules()

    def _load_initial_rules(self):
        """Load rules from config file"""
        config_path = os.path.join(os.path.dirname(__file__), '../config/pricing_rules.json')

        with open(config_path, 'r') as f:
            rules = json.load(f)
            for sku_id, config in rules.items():
                self.set_rules(sku_id, config)

    def get_rules(self, sku_id: str) -> Dict[str, Any]:
```

```
    """
    Get pricing rules for a SKU.

    Input: sku_id (str)
    Output: Dict with rules
    Raises: ValueError if rules not found
    """
    key = f"rules:{sku_id}"
    rules_json = self.redis_client.get(key)

    if not rules_json:
        raise ValueError(f"No rules found for SKU: {sku_id}")

    return json.loads(rules_json)

def set_rules(self, sku_id: str, rules: Dict[str, Any]):
    """Store rules for a SKU (no expiration)"""
    key = f"rules:{sku_id}"
    self.redis_client.set(key, json.dumps(rules))

def get_weekly_targets(self, sku_id: str, week: str) -> Dict[str, Any]:
    """
    Get weekly targets set by Weekly Planning Agent.

    Input:
      - sku_id: str
      - week: str (format: "2025-W42")
    Output: Dict with weekly targets

    Returns default targets if not found.
    """
    key = f"weekly_targets:{sku_id}:{week}"
    targets_json = self.redis_client.get(key)

    if not targets_json:
        # Fallback to default from rules
        return self._get_default_targets(sku_id)

    return json.loads(targets_json)

def set_weekly_targets(
```

```
    self,
    sku_id: str,
    week: str,
    targets: Dict[str, Any]
):
    """
    Store weekly targets (expires after 7 days).

    Called by Weekly Planning Agent after approval.
    """
    key = f"weekly_targets:{sku_id}:{week}"
    self.redis_client.setex(
        key,
        604800,  # 7 days in seconds
        json.dumps(targets)
    )

def _get_default_targets(self, sku_id: str) -> Dict[str, Any]:
    """Fallback to default targets from rules"""
    rules = self.get_rules(sku_id)

    return {
        "daily_tpc_target_kg": 2400,  # Default baseline
        "penetration_target_pct": rules['penetration_target']['daily_max'],
        "gm_target": rules['gm_target'],
        "tpo_target_kg": rules['tpo_target'],
        "strategy": "MAINTAIN"
    }

def get_workflow(self, workflow_name: str = "pricing") -> List[str]:
    """
    Get workflow execution steps.

    Input: workflow_name (str)
    Output: List of step names
    """
    key = f"workflow:{workflow_name}"
    workflow_json = self.redis_client.get(key)

    if not workflow_json:
        # Return default workflow
```

```
        return self._get_default_workflow(workflow_name)

    return json.loads(workflow_json)

def _get_default_workflow(self, workflow_name: str) -> List[str]:
    """Default workflows"""
    workflows = {
        "pricing": [
            "fetch_weekly_context",
            "fetch_current_wsp",
            "run_demand_forecast",
            "check_inventory_levels",
            "check_returns_risk",
            "retrieve_similar_scenarios",
            "calculate_base_price",
            "apply_adjustments",
            "validate_constraints",
            "generate_reasoning",
            "calculate_confidence",
            "determine_approval_needed"
        ],
        "weekly_planning": [
            "gather_past_week_performance",
            "assess_supply_stability",
            "decide_strategy",
            "calculate_historical_baseline",
            "set_weekly_targets",
            "analyze_penetration_gap",
            "plan_campaigns",
            "generate_checkpoints",
            "create_plan_document"
        ],
        "campaign": [
            "identify_trigger",
            "analyze_customer_segments",
            "retrieve_similar_campaigns",
            "calculate_optimal_discount",
            "predict_outcomes",
            "generate_recommendation",
            "create_approval_request"
        ]
```

```
        }

        return workflows.get(workflow_name, [])

    def cache_session_data(
        self,
        session_id: str,
        data: Dict[str, Any],
        ttl_seconds: int = 3600
    ):
        """Cache session data (e.g., conversation context)"""
        key = f"session:{session_id}"
        self.redis_client.setex(key, ttl_seconds, json.dumps(data))

    def get_session_data(self, session_id: str) -> Optional[Dict[str, Any]]:
        """Retrieve session data"""
        key = f"session:{session_id}"
        data_json = self.redis_client.get(key)

        if not data_json:
            return None

        return json.loads(data_json)
```

Configuration File (`config/pricing_rules.json`):

json

```
{
 "TOMATO_LOCAL": {
  "penetration_target": {
   "daily_min": 45,
   "daily_max": 47,
   "weekly": 60
  },
  "gm_target": [4, 5],
  "tpo_target": 29,
  "timing_rules": {
   "price_hike_time": "07:00",
   "price_drop_time": "09:30",
   "review_time": "11:30"
  },
  "inventory_rules": {
   "surplus_threshold": 0.20,
   "deficit_threshold": -0.20,
   "surplus_action": "REDUCE_SP_OR_CAMPAIGN",
   "deficit_action": "INCREASE_SP",
   "high_inventory_days": 2.0
  },
  "margin_rules": {
   "min_sm_percentage": 10,
   "min_gm_per_kg": 2,
   "ideal_pp_formula": "WSP - (0.10 * WSP) - FML",
   "commission_pct": 10,
   "fml_typical": 1.50
  },
  "markup_factors": {
   "high_inventory": 1.15,
   "normal": 1.25,
   "low_inventory": 1.35
  },
  "campaign_defaults": {
   "lapsed_customer_days": [7, 21],
   "conversion_rate_estimate": 0.18,
   "avg_order_size_kg": 20,
   "min_budget": 500,
   "max_budget_pct_of_revenue": 0.05
  }
 },
```

```
"POTATO_GRADED": {
  "penetration_target": {
    "daily_min": 31,
    "daily_max": 34,
    "weekly": 50
  },
  "gm_target": [4, 5],
  "tpo_target": 30,
  "timing_rules": {
    "price_hike_time": "07:00",
    "price_drop_time": "09:30",
    "review_time": "11:30"
  },
  "inventory_rules": {
    "surplus_threshold": 0.20,
    "deficit_threshold": -0.20,
    "surplus_action": "REDUCE_SP_OR_CAMPAIGN",
    "deficit_action": "INCREASE_SP",
    "high_inventory_days": 3.0
  },
  "margin_rules": {
    "min_sm_percentage": 10,
    "min_gm_per_kg": 2,
    "ideal_pp_formula": "WSP - (0.10 * WSP) - FML",
    "commission_pct": 10,
    "fml_typical": 1.20
  },
  "markup_factors": {
    "high_inventory": 1.15,
    "normal": 1.25,
    "low_inventory": 1.30
  }
},
"ONION_LOCAL": {
  "penetration_target": {
    "daily_min": 31,
    "daily_max": 34,
    "weekly": 48
  },
  "gm_target": [3, 4],
  "tpo_target": 26,
```

```
  "timing_rules": {
    "price_hike_time": "07:00",
    "price_drop_time": "09:30",
    "review_time": "11:30"
  },
  "inventory_rules": {
    "surplus_threshold": 0.20,
    "deficit_threshold": -0.20,
    "surplus_action": "REDUCE_SP_OR_CAMPAIGN",
    "deficit_action": "INCREASE_SP",
    "high_inventory_days": 3.0
  },
  "margin_rules": {
    "min_sm_percentage": 8,
    "min_gm_per_kg": 1.5,
    "ideal_pp_formula": "WSP - (0.10 * WSP) - FML",
    "commission_pct": 10,
    "fml_typical": 1.20
  },
  "markup_factors": {
    "high_inventory": 1.12,
    "normal": 1.20,
    "low_inventory": 1.28
  }
 }
}
```

6.2 Vector Database (Pinecone)

memory/vector_store.py:

python

```python
import pinecone
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Pinecone as LangchainPinecone
from typing import List, Dict, Optional
import os
import uuid

class VectorMemory:
    """
    Stores and retrieves historical scenarios using similarity search (RAG).

    Storage: Pinecone (managed vector database)
    Embedding: OpenAI text-embedding-ada-002 (1536 dimensions)

    Use Cases:
      - Find similar pricing scenarios
      - Retrieve relevant campaign outcomes
      - Historical pattern matching
    """

    def __init__(self):
        pinecone.init(
            api_key=os.getenv('PINECONE_API_KEY'),
            environment=os.getenv('PINECONE_ENV')
        )

        self.index_name = os.getenv('PINECONE_INDEX', 'category-agent-episodes')
        self.embeddings = OpenAIEmbeddings(
            openai_api_key=os.getenv('OPENAI_API_KEY'),
            model="text-embedding-ada-002"
        )

        # Create index if doesn't exist
        if self.index_name not in pinecone.list_indexes():
            pinecone.create_index(
                name=self.index_name,
                dimension=1536,
                metric="cosine",
                pods=1,
                pod_type="p1.x1"
            )
```

```
        self.index = pinecone.Index(self.index_name)
        self.vectorstore = LangchainPinecone(
            self.index,
            self.embeddings.embed_query,
            "text"
        )

    def store_pricing_episode(self, episode: Dict):
        """
        Store a pricing episode for future RAG retrieval.

        Input: episode (Dict) - Pricing episode data
        Output: episode_id (str)

        Process:
          1. Create searchable text representation
          2. Generate embedding using OpenAI
          3. Store in Pinecone with metadata
        """
        # Create rich text description
        text = self._create_episode_text(episode)

        # Create metadata for filtering
        metadata = {
            "episode_id": episode['episode_id'],
            "sku_id": episode['sku_id'],
            "date": episode['date'],
            "wsp": float(episode['context']['wsp']),
            "pp": float(episode['context']['pp']),
            "supply_status": episode['context']['supply_status'],
            "sp_recommended": float(episode['decision']['sp_recommended']),
            "sp_approved": float(episode['decision']['sp_approved']),
            "confidence": float(episode['decision']['confidence']),
            "tpc_actual": float(episode['outcome']['tpc_actual']),
            "tpc_predicted": float(episode['outcome']['tpc_predicted']),
            "accuracy": float(episode['outcome'].get('overall_accuracy', 0.0))
        }

        # Store
        self.vectorstore.add_texts(
```

```
            texts=[text],
            metadatas=[metadata],
            ids=[episode['episode_id']]
        )

        return episode['episode_id']

    def _create_episode_text(self, episode: Dict) -> str:
        """Create searchable text from episode"""
        ctx = episode['context']
        dec = episode['decision']
        out = episode['outcome']

        text = f"""
        Pricing Episode for {episode['sku_id']} on {episode['date']}

        Market Context:
        - Wholesale Price (WSP): ₹{ctx['wsp']}/kg
        - Purchase Price (PP): ₹{ctx['pp']}/kg
        - Procurement Margin (PM): ₹{ctx['pm']}/kg
        - Inventory: {ctx['inventory_kg']}kg
        - Supply Status: {ctx['supply_status']}
        - Inventory Surplus: {ctx['inventory_surplus_pct']}%

        Pricing Decision:
        - Recommended Sale Price (SP): ₹{dec['sp_recommended']}/kg
        - Approved Sale Price: ₹{dec['sp_approved']}/kg
        - Reasoning: {dec['sp_reasoning']}
        - Confidence: {dec['confidence']}
        - Human Approved: {dec['human_approved']}

        Outcomes:
        - Predicted TPC: {out['tpc_predicted']}kg
        - Actual TPC: {out['tpc_actual']}kg
        - Predicted Penetration: {out['penetration_predicted']}%
        - Actual Penetration: {out['penetration_actual']}%
        - Predicted GM: ₹{out['gm_predicted']}/kg
        - Actual GM: ₹{out['gm_actual']}/kg
        - Prediction Accuracy: {out.get('overall_accuracy', 0)*100}%

        Key Learnings: {out.get('learning_notes', 'N/A')}
```

```
    """

    return text

def search_similar_pricing_scenarios(
    self,
    query: str,
    sku_id: Optional[str] = None,
    wsp_range: Optional[tuple] = None,
    supply_status: Optional[str] = None,
    k: int = 5,
    min_accuracy: float = 0.7
) -> List[Dict]:
    """
    Search for similar historical pricing scenarios.

    Input:
      - query: str (natural language description)
      - sku_id: Optional[str] (filter by SKU)
      - wsp_range: Optional[tuple] (min_wsp, max_wsp)
      - supply_status: Optional[str] (SURPLUS/BALANCED/DEFICIT)
      - k: int (number of results)
      - min_accuracy: float (only return episodes with accuracy >threshold)

    Output: List of similar scenarios with scores

    Example query:
      "high inventory negative PM WSP around 35"
    """
    # Build filter
    filter_dict = {}

    if sku_id:
        filter_dict["sku_id"] = {"$eq": sku_id}

    if wsp_range:
        filter_dict["wsp"] = {
            "$gte": wsp_range[0],
            "$lte": wsp_range[1]
        }
```

```
    if supply_status:
        filter_dict["supply_status"] = {"$eq": supply_status}

    if min_accuracy:
        filter_dict["accuracy"] = {"$gte": min_accuracy}

    # Search
    results = self.vectorstore.similarity_search_with_score(
        query,
        k=k,
        filter=filter_dict if filter_dict else None
    )

    return [
        {
            "text": doc.page_content,
            "metadata": doc.metadata,
            "similarity_score": float(score)
        }
        for doc, score in results
    ]

def store_campaign_episode(self, campaign: Dict):
    """Store campaign episode for future retrieval"""
    text = self._create_campaign_text(campaign)

    metadata = {
        "campaign_id": campaign['campaign_id'],
        "sku_id": campaign['sku_id'],
        "campaign_type": campaign['campaign_type'],
        "segment_name": campaign['segment_name'],
        "conversion_rate_actual": float(campaign['conversion_rate_actual']),
        "roi_actual": float(campaign['actual_roi']),
        "success_score": float(campaign['overall_success_score'])
    }

    self.vectorstore.add_texts(
        texts=[text],
        metadatas=[metadata],
        ids=[campaign['campaign_id']]
    )
```

```
    return campaign['campaign_id']

def _create_campaign_text(self, campaign: Dict) -> str:
    """Create searchable text from campaign"""
    text = f"""
    Campaign Episode for {campaign['sku_id']} on {campaign['campaign_date']}

    Campaign Details:
    - Type: {campaign['campaign_type']}
    - Trigger: {campaign['campaign_trigger']}
    - Segment: {campaign['segment_name']} ({campaign['segment_size']} customers)
    - Duration: {campaign['duration_hours']} hours

    Predictions:
    - Expected Conversion: {campaign['conversion_rate_predicted']*100}%
    - Expected Incremental TPC: {campaign['incremental_tpc_predicted']}kg
    - Expected ROI: {campaign['expected_roi']}x

    Actuals:
    - Actual Conversion: {campaign['conversion_rate_actual']*100}%
    - Actual Incremental TPC: {campaign['incremental_tpc_actual']}kg
    - Actual ROI: {campaign['actual_roi']}x
    - Success Score: {campaign['overall_success_score']}

    Key Learnings: {campaign.get('learning_notes', 'N/A')}
    """

    return text

def search_similar_campaigns(
    self,
    query: str,
    sku_id: Optional[str] = None,
    campaign_type: Optional[str] = None,
    min_success_score: float = 0.6,
    k: int = 5
) -> List[Dict]:
    """Search for similar successful campaigns"""
    filter_dict = {}
```

```
        if sku_id:
            filter_dict["sku_id"] = {"$eq": sku_id}

        if campaign_type:
            filter_dict["campaign_type"] = {"$eq": campaign_type}

        if min_success_score:
            filter_dict["success_score"] = {"$gte": min_success_score}

        results = self.vectorstore.similarity_search_with_score(
            query,
            k=k,
            filter=filter_dict if filter_dict else None
        )

        return [
            {
                "text": doc.page_content,
                "metadata": doc.metadata,
                "similarity_score": float(score)
            }
            for doc, score in results
        ]
```

Due to the comprehensive nature of this document, I'll continue with the remaining sections (Agents 3-7, Orchestrator, API, Frontend, Testing, Deployment, Team Responsibilities) in the next part. The document is currently at ~15,000 words. Would you like me to:

1. **Continue with all remaining sections** in one large artifact (will be ~40,000-50,000 words total)
2. **Create multiple downloadable artifacts** (Part 1: Foundation, Part 2: Agents, Part 3: Deployment)
3. **Focus on specific sections** you need most urgently

Let me know and I'll provide the complete guide!