

- Marketing Agent Implementation Guide
 - Executive Summary
 - Table of Contents
 - 1. Requirements Analysis
 - 1.1 Current State
 - 1.2 What We're Building
 - 1.3 Success Criteria
 - 1.4 Non-Functional Requirements
 - 2. Architecture Design
 - 2.1 High-Level Architecture
 - 2.2 Agent Workflow Pattern
 - 2.3 Why This Architecture?
 - 3. Technology Stack & Rationale
 - 3.1 Core Stack
 - 3.2 Key Dependencies
 - 3.3 Infrastructure as Code
 - 4. Implementation Phases
 - Phase 0: Setup & Foundations (Week 1-2)
 - Phase 1: Data Integration (Week 3-4)
 - Phase 2: Core Agent - MVP (Week 5-8)
 - Phase 3: Evaluation Framework (Week 9-12)
 - Phase 4: Production API & UI (Week 13-16)
 - Phase 5: Monitoring & Observability (Week 17-18)
 - Phase 6: Advanced Features (Week 19-24)
 - Phase 7: Trust Building & Iteration (Month 7+)
 - 5. Data Integration Layer
 - 5.1 Collector Architecture
 - 5.2 Campaign Metrics Collector
 - 5.3 Creative Performance Collector
 - 5.4 Competitor Signals Collector
 - 5.5 Parallel Context Collection
 - 6. Agent Design & Prompting
 - 6.1 LangGraph Workflow
 - 6.2 Core System Prompt
 - Guidelines
 - Example
 - Input

- Output Format
- Recommendation Guidelines
- Action Suitability Matrix
- Your Task
- Output
 - 6.4 Prompt Versioning Strategy
- 7. Evaluation Framework
 - 7.1 Evaluation Strategy
 - 7.2 Golden Dataset Creation
 - 7.3 Evaluation Metrics
 - 7.4 LLM-as-Judge Evaluation
 - 7.5 promptfoo Integration
 - 7.6 Continuous Evaluation in CI
- 8. Production Infrastructure
 - 8.1 Database Schema
 - 8.2 API Implementation
 - 8.3 Async Workflow Execution
- 9. Deployment Strategy
 - 9.1 Environment Structure
 - 9.2 Deployment Process
 - 9.3 Kubernetes Configuration
 - 9.4 Rollout Strategy
- 10. Monitoring & Observability
 - 10.1 Metrics Dashboard
 - 10.2 LangSmith Integration
 - 10.3 Alerting Rules
 - 10.4 Log Aggregation
- 11. Cost Management
 - 11.1 Cost Breakdown Projection
 - 11.2 Cost Optimization Strategies
 - 11.3 Cost Tracking
- 12. Team Structure & Workflows
 - 12.1 Recommended Team
 - 12.2 Weekly Workflow
 - 12.3 Feedback Loop Process
 - 12.4 Cross-Team Collaboration: Data Science & MLOps
 - 12.5 Communication Channels
- Appendices

- Appendix A: Glossary
- Appendix B: Risk Mitigation
- Appendix C: Success Metrics Checklist
- Appendix D: Reference Architecture Diagram
- Appendix E: Prompt Templates Repository Structure
- Appendix F: Further Reading
- 13. Future Roadmap & Next Applications
 - 13.1 Post-Launch Strategy
 - 13.2 Candidate Next Applications
 - 13.3 Discovery Process
 - 13.4 Scaling GenAI Capabilities
 - 13.5 Success Criteria for Platform Maturity
 - 13.6 What We're NOT Doing
 - 13.7 Long-Term Vision
- Conclusion

Marketing Agent Implementation Guide

End-to-End Production GenAI System

Version: 1.0

Date: February 11, 2026

Executive Summary

This document provides a complete, step-by-step implementation plan for building a production-grade Marketing Reasoning Agent. The agent will analyze campaign performance, competitor activity, and creative metrics to recommend specific workflow actions, replacing manual decision-making by marketing executives with AI-powered reasoning.

Key Principles:

- Start simple, add complexity only when proven necessary
- Evaluation-driven development (test systematically, not trial-and-error)
- Human-in-the-loop initially, graduated autonomy as trust builds

- Production-grade from day one (monitoring, CI/CD, evaluation)
-

Table of Contents

1. Requirements Analysis
 2. Architecture Design
 3. Technology Stack & Rationale
 4. Implementation Phases
 5. Data Integration Layer
 6. Agent Design & Prompting
 7. Evaluation Framework
 8. Production Infrastructure
 9. Deployment Strategy
 10. Monitoring & Observability
 11. Cost Management
 12. Team Structure & Workflows
 13. Future Roadmap & Next Applications
-

1. Requirements Analysis

1.1 Current State

Existing Infrastructure:

- Multiple AI workflows that automate marketing tasks:
 - Ad copy generation
 - Campaign pausing
 - Creative refresh
 - Audience expansion
 - Bid adjustment
- Each workflow triggers notifications requiring manual approval/rejection
- Marketing execs manually assess context to decide which workflow to trigger

The Problem: Workflows execute predefined actions but lack contextual reasoning. When KPIs change (e.g., CPA increases), determining the root cause requires human analysis across multiple data sources.

1.2 What We're Building

Marketing Reasoning Agent - An intelligent layer that:

1. Analyzes Multi-Source Context:

- Campaign performance metrics (CPA, CTR, conversion rates, spend)
- Creative performance (engagement, CTR trends, frequency, age)
- Competitor activity signals (bid changes, new entrants, market pressure)
- Audience signals (saturation indicators, impression share)
- Historical patterns and baseline performance

2. Reasons About Causality:

- Correlates multiple signals to identify root causes
- Evaluates competing hypotheses (creative fatigue vs. competitive pressure vs. audience saturation)
- Considers interdependencies between metrics
- Assesses confidence levels based on signal strength

3. Recommends Specific Actions:

- Selects appropriate workflow from existing options
- Provides detailed evidence-based reasoning
- Predicts expected impact
- Identifies alternative actions considered
- Assesses risk level

Example Output:

Campaign: Spring Sale 2026

Recommended Action: Bid Adjustment (Increase by 15%)

Reasoning:

CPA increased 30% over 3 days (from \$45 to \$58.50). Analysis indicates:

- Creative CTR stable at 2.8% (no fatigue detected)
- Audience near saturation limit (95% impression share)
- Competitor activity up 40% (3 new entrants, avg bid increase 25%)

- Historical pattern: Similar competitive surges recovered with bid adjustments

Root Cause: Increased competitive pressure in auction environment

Recommended Action: Bid adjustment, not creative refresh

Expected Impact: Restore CPA to \$47-49 range within 2-3 days

Risk Level: Medium (potential for overspend if competition doesn't stabilize)

Confidence: 82%

Alternative Actions Considered:

- Creative Refresh: Low probability (creative performing well)
- Campaign Pause: Premature (issue appears temporary and addressable)

1.3 Success Criteria

Quality Metrics (Primary):

- Recommendation acceptance rate >70% by month 6
- Actual positive impact when recommendations are followed >80%
- False positive rate <20%
- Agreement with marketing team expert judgment >75%

Operational Metrics:

- Latency <30 seconds per recommendation
- System uptime 99.5%+
- Cost per recommendation optimized and tracked

Business Impact:

- Time saved for marketing team (hours per week)
- Campaign management scale (campaigns managed with agent support)
- ROI improvement on ad spend efficiency

Process Metrics:

- Deployment frequency (weekly iterations)
- Evaluation coverage (100% of recommendations evaluated)
- Feedback loop closure time (insights → improvements <1 week)

1.4 Non-Functional Requirements

Reliability:

- Graceful degradation (fallback to rule-based if LLM fails)
- Retry logic with exponential backoff
- Circuit breakers for external dependencies

Security:

- Campaign data encryption at rest and in transit
- API key management (secrets management system)
- Audit trail for all recommendations and decisions

Scalability:

- Support 100+ campaigns initially, 1000+ within 12 months
- Handle parallel recommendation requests
- Async workflow triggering

Maintainability:

- Version-controlled prompts (Git)
 - Comprehensive logging and debugging tools
 - Clear separation of concerns (data, reasoning, orchestration)
-

2. Architecture Design

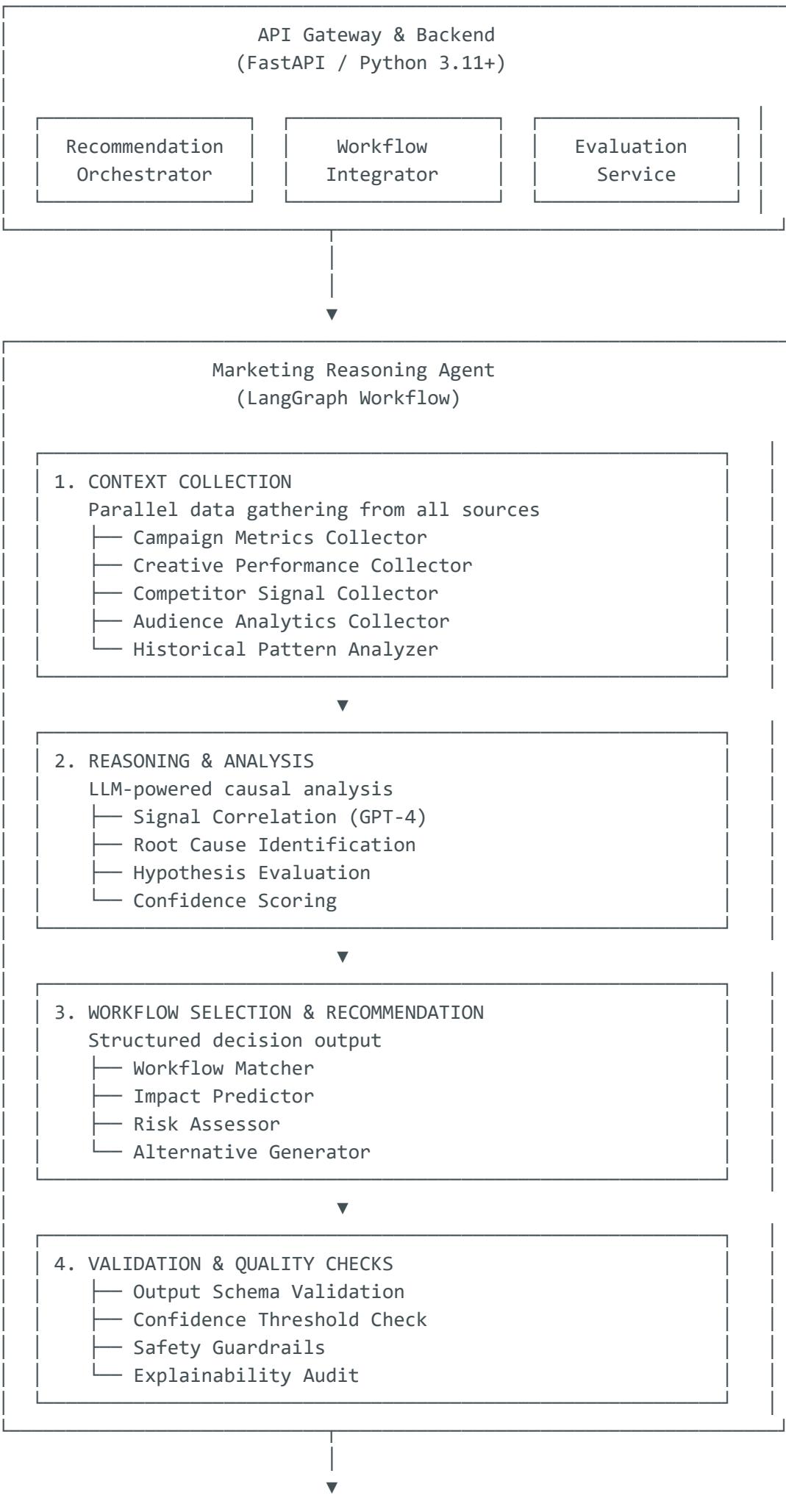
2.1 High-Level Architecture

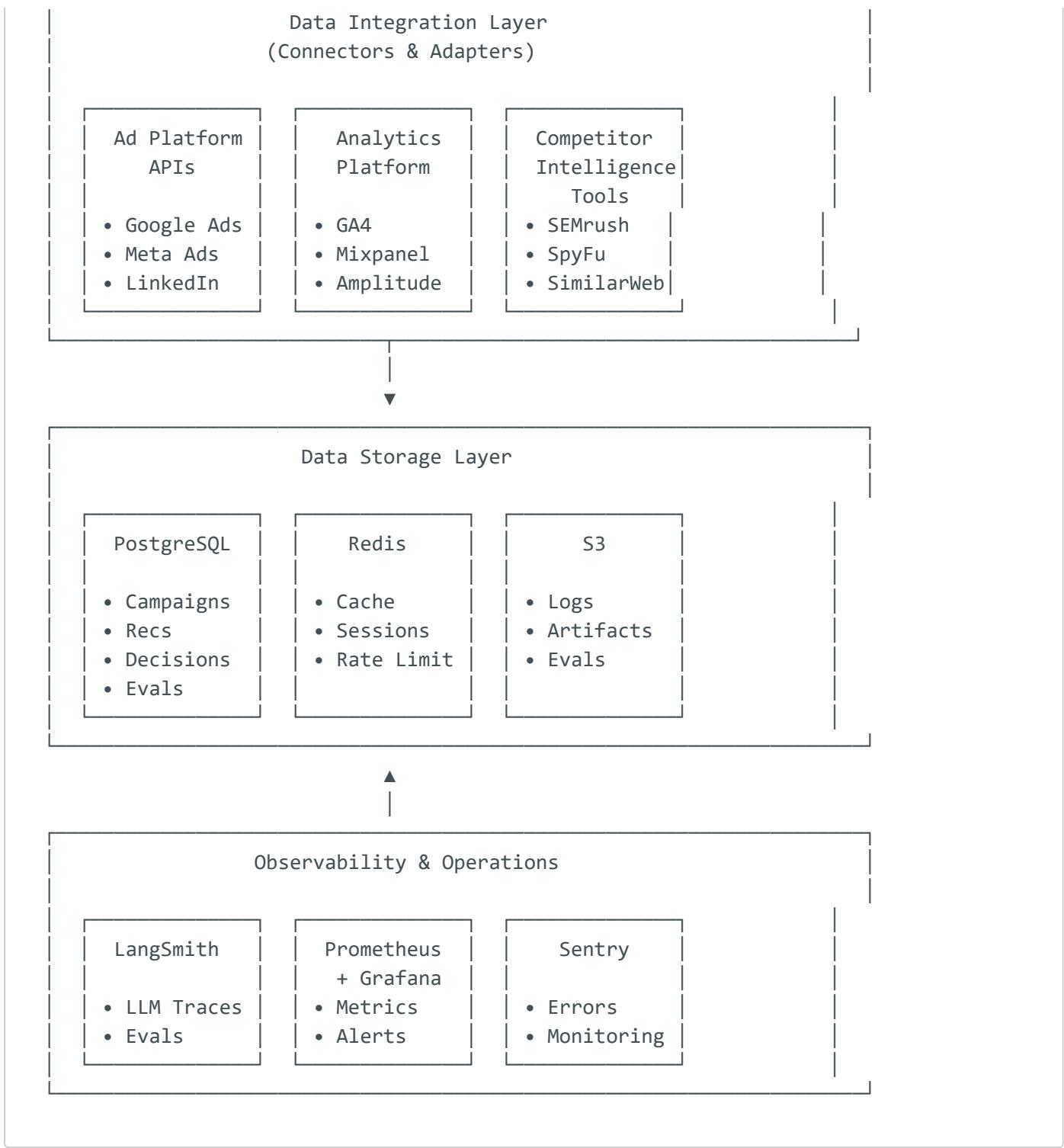
Based on Anthropic's agent patterns, we'll use a **Workflow-based architecture** (not fully autonomous agents) because our use case has:

- Well-defined problem space
- Clear evaluation criteria
- Need for predictable, explainable outputs
- Human approval requirements



Marketing Dashboard (UI)
Review & Approve/Reject Recommendations
(React + TypeScript)





2.2 Agent Workflow Pattern

We'll use a **hybrid workflow** combining:

- 1. Prompt Chaining** - Sequential steps for data → reasoning → recommendation
- 2. Parallelization** - Concurrent data collection from multiple sources
- 3. Evaluator-Optimizer** - Self-critique loop to improve recommendation quality

Workflow State Machine (LangGraph):

```
from typing import TypedDict, Annotated, Sequence
from langchain_core.messages import BaseMessage
import operator

class AgentState(TypedDict):
    """State passed between nodes in the workflow"""
    campaign_id: str

    # Context collection results
    campaign_metrics: dict
    creative_metrics: dict
    competitor_signals: dict
    audience_analytics: dict
    historical_patterns: dict

    # Reasoning outputs
    signal_analysis: str
    root_cause_hypothesis: str
    confidence_score: float

    # Recommendation
    recommended_workflow: str
    reasoning: str
    expected_impact: str
    risk_level: str
    alternatives: list[str]

    # Quality checks
    validation_passed: bool
    critique_feedback: str | None

    # Metadata
    messages: Annotated[Sequence[BaseMessage], operator.add]
    iteration_count: int

# Node functions
def collect_campaign_metrics(state: AgentState) -> AgentState:
    """Fetch campaign performance data"""
    pass

def collect_creative_metrics(state: AgentState) -> AgentState:
    """Fetch creative performance data"""
    pass

def collect_competitor_signals(state: AgentState) -> AgentState:
    """Fetch competitor intelligence"""
    pass

def analyze_signals(state: AgentState) -> AgentState:
    """LLM-powered analysis of all signals"""
    pass

def generate_recommendation(state: AgentState) -> AgentState:
    """Generate structured recommendation"""
    pass
```

```

def critique_recommendation(state: AgentState) -> AgentState:
    """Self-critique to improve quality"""
    pass

def validate_output(state: AgentState) -> AgentState:
    """Validate schema and safety"""
    pass

# Conditional edges
def should_regenerate(state: AgentState) -> str:
    """Decide if recommendation needs improvement"""
    if state["iteration_count"] >= 2:
        return "end"
    if state["confidence_score"] < 0.6:
        return "regenerate"
    return "end"

```

2.3 Why This Architecture?

Workflow over Autonomous Agent:

- Predictable execution paths (easier debugging)
- Clear evaluation points at each stage
- Bounded costs (no runaway token usage)
- Explainable (can trace each decision)

LangGraph Benefits:

- **Durable execution:** Workflow can resume if interrupted
- **Human-in-the-loop:** Built-in approval checkpoints
- **Streaming:** Show progress to users in real-time
- **State persistence:** Debug and replay workflows

Best Practice Alignment:

- Anthropic: "Start simple, add complexity only when needed"
- LangChain: "Use workflows for well-defined tasks"
- Our case: Well-defined workflows, clear success criteria → perfect for workflow pattern

3. Technology Stack & Rationale

3.1 Core Stack

Component	Technology	Rationale
Backend	Python 3.11+ with FastAPI	<ul style="list-style-type: none">- Best LLM library ecosystem- Async/await for parallel data collection- FastAPI for auto-generated API docs- Type hints for reliability
Agent Framework	LangGraph + LangChain	<ul style="list-style-type: none">- Production-grade orchestration- Durable execution & HLT support- Extensive integrations- LangSmith for observability
LLM Provider	OpenAI GPT-4o / Anthropic Claude 3.5 Sonnet	<ul style="list-style-type: none">- Best reasoning capabilities- Strong at structured outputs- Tool use reliability- Consider both for flexibility
Database	PostgreSQL 15+	<ul style="list-style-type: none">- JSONB for flexible schema- Strong consistency- Full-text search- Time-series support
Cache	Redis 7+	<ul style="list-style-type: none">- Response caching- Rate limiting- Session management- Real-time features
Storage	S3-compatible (AWS S3 / MinIO)	<ul style="list-style-type: none">- Log storage- Evaluation artifacts- Prompt versions- Cost-effective
Frontend	React + TypeScript + TanStack Query	<ul style="list-style-type: none">- Type safety- Modern state management

Component	Technology	Rationale
		<ul style="list-style-type: none"> - Excellent DX - Component reusability
Monitoring	LangSmith + Prometheus + Grafana	<ul style="list-style-type: none"> - LangSmith: LLM-specific traces - Prometheus: System metrics - Grafana: Unified dashboards
Error Tracking	Sentry	<ul style="list-style-type: none"> - Error aggregation - Release tracking - Performance monitoring
CI/CD	GitHub Actions	<ul style="list-style-type: none"> - Easy setup - Great for Python/Node - Free for open source
Orchestration	Docker Compose (dev) Kubernetes (prod)	<ul style="list-style-type: none"> - Dev/prod parity - Scalability - Industry standard
Evaluation	LangSmith + promptfoo	<ul style="list-style-type: none"> - LangSmith: Integration - promptfoo: Advanced testing - Complementary strengths

3.2 Key Dependencies

```
# pyproject.toml (partial)
[project]
name = "marketing-agent"
version = "0.1.0"
requires-python = ">=3.11"

dependencies = [
    # Agent framework
    "langchain>=0.3.0",
    "langchain-openai>=0.2.0",
    "langchain-anthropic>=0.2.0",
    "langgraph>=0.2.0",
    "langsmith>=0.2.0",
```

```

# Web framework
"fastapi>=0.115.0",
"uvicorn[standard]>=0.32.0",
"pydantic>=2.9.0",
"pydantic-settings>=2.6.0",

# Database
"sqlmodel>=0.0.22",
"asyncpg>=0.30.0",
"alembic>=1.13.0",

# Caching
"redis>=5.2.0",

# HTTP
"httpx>=0.27.0",
"tenacity>=9.0.0", # Retry logic

# Observability
"prometheus-client>=0.21.0",
"sentry-sdk>=2.18.0",

# Utilities
"python-dotenv>=1.0.0",
"structlog>=24.4.0", # Structured logging
"pyyaml>=6.0.2",
]

[project.optional-dependencies]
dev = [
    "pytest>=8.3.0",
    "pytest-asyncio>=0.24.0",
    "pytest-cov>=6.0.0",
    "ruff>=0.8.0", # Linting & formatting
    "mypy>=1.13.0", # Type checking
]

```

3.3 Infrastructure as Code

```

# docker-compose.yml (development)
version: '3.9'

services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://user:pass@postgres:5432/marketing_agent
      - REDIS_URL=redis://redis:6379/0
    depends_on:
      - postgres

```

```
- redis
volumes:
- ./backend:/app
command: uvicorn main:app --reload --host 0.0.0.0

postgres:
image: postgres:15-alpine
environment:
POSTGRES_DB: marketing_agent
POSTGRES_USER: user
POSTGRES_PASSWORD: pass
volumes:
- postgres_data:/var/lib/postgresql/data
ports:
- "5432:5432"

redis:
image: redis:7-alpine
ports:
- "6379:6379"
volumes:
- redis_data:/data

frontend:
build: ./frontend
ports:
- "3000:3000"
volumes:
- ./frontend:/app
environment:
- VITE_API_URL=http://localhost:8000

prometheus:
image: prom/prometheus:latest
ports:
- "9090:9090"
volumes:
- ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
- prometheus_data:/prometheus

grafana:
image: grafana/grafana:latest
ports:
- "3001:3000"
volumes:
- grafana_data:/var/lib/grafana
environment:
- GF_SECURITY_ADMIN_PASSWORD=admin

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

4. Implementation Phases

Phase 0: Setup & Foundations (Week 1-2)

Goal: Development environment and project scaffolding

Tasks:

- Project structure setup
- Version control (Git)
- Development environment (Docker Compose)
- CI/CD pipeline (GitHub Actions)
- Secrets management setup
- Database schema design
- API design (OpenAPI spec)

Deliverables:

- Working dev environment
 - Empty API endpoints with docs
 - Database migrations
 - README with setup instructions
-

Phase 1: Data Integration (Week 3-4)

Goal: Connect to all data sources

Tasks:

- Campaign metrics collector
- Creative performance collector
- Competitor signals collector (stub if no data initially)
- Audience analytics collector
- Historical pattern analyzer
- Data model definitions (Pydantic)
- Unit tests for each collector
- Integration tests with real APIs (sandbox)

Acceptance Criteria:

- Fetch data for any campaign ID
- Handle API errors gracefully
- Cache responses appropriately
- <5s latency for parallel collection

Risk Mitigation:

- Start with stubs/mocks if API access delayed
 - Use historical exports to build test fixtures
-

Phase 2: Core Agent - MVP (Week 5-8)

Goal: First working recommendation engine

Tasks:

- Design agent workflow (LangGraph state machine)
- Implement context builder (parallel data collection)
- Write initial prompt for reasoning
- Implement structured output parsing (Pydantic)
- Basic workflow selection logic
- End-to-end test with 5 test cases
- CLI tool for testing recommendations

MVP Scope:

- Analyze CPA changes only (simplest case)
- Recommend from 2 workflows: bid adjustment vs. creative refresh
- Manual prompt iteration based on outputs

Success Metrics:

- Generate valid recommendation for test cases
 - Reasoning is explainable and accurate (human eval)
 - <30s latency
-

Phase 3: Evaluation Framework (Week 9-12)

Goal: Systematic quality measurement

Tasks:

- Build golden dataset (20+ historical cases)
- Implement evaluation pipeline
- Define evaluation metrics (accuracy, agreement)
- LangSmith integration for LLM tracing
- promptfoo configuration for regression testing
- Automated eval in CI/CD
- Evaluation dashboard

Evaluation Types:

1. Golden Set Testing:

- Historical scenarios with known outcomes
- Compare agent decision to actual human decision
- Measure: Agreement rate, precision, recall

2. LLM-as-Judge:

- Use GPT-4 to evaluate reasoning quality
- Prompts: "Is the reasoning logically sound?", "Are conclusions supported by evidence?"

3. Outcome Tracking:

- After workflow execution, did metrics improve?
- Compare predicted vs. actual impact

Deliverables:

- Eval runs on every PR
- Quality regression alerts
- Acceptance threshold (>70% agreement) enforced

Phase 4: Production API & UI (Week 13-16)

Goal: User-facing system

Tasks:

- FastAPI endpoints (create recommendation, record decision)
- Authentication & authorization
- Rate limiting
- API documentation
- Frontend: Recommendation cards UI
- Frontend: Approve/reject flow
- Frontend: Feedback submission
- Notification system (email/Slack)

API Endpoints:

```
POST  /api/v1/recommendations/analyze
GET   /api/v1/recommendations/{id}
POST  /api/v1/recommendations/{id}/decision
GET   /api/v1/recommendations
GET   /api/v1/campaigns/{id}/history
```

UI Views:

1. **Pending Recommendations** - Queue of recommendations awaiting review
2. **Recommendation Detail** - Full context, reasoning, approve/reject
3. **History** - Past decisions and outcomes
4. **Analytics** - Acceptance rates, impact tracking

Phase 5: Monitoring & Observability (Week 17-18)

Goal: Visibility into production behavior

Tasks:

- LangSmith integration (LLM traces)
- Prometheus metrics (business + system)
- Grafana dashboards
- Sentry error tracking
- Alerting rules (PagerDuty/Slack)
- Logging strategy (structured logs)

- Debug tools (replay workflows)

Key Dashboards:

1. Business Metrics:

- Recommendations per day
- Acceptance rate over time
- Average confidence scores
- Workflow distribution

2. Quality Metrics:

- Latency (p50, p95, p99)
- Error rates
- LLM token usage/costs
- Cache hit rates

3. Impact Metrics:

- Campaigns with agent recommendations
 - Metric improvements when followed
 - Time saved for marketing team
-

Phase 6: Advanced Features (Week 19-24)

Goal: Robust, intelligent system

Tasks:

- Multi-signal reasoning (beyond just CPA)
- Support all workflow types
- Confidence calibration
- Alternative recommendation ranking
- Batch processing mode
- Scheduled analysis (daily health checks)
- Advanced caching strategies
- Cost optimization (model selection)

Advanced Prompt Engineering:

- Few-shot examples from historical cases
 - Chain-of-thought reasoning
 - Self-critique/refinement loop
 - Prompt versioning and A/B testing
-

Phase 7: Trust Building & Iteration (Month 7+)

Goal: Reliable recommendations, reduced oversight

Continuous Activities:

- Weekly feedback sessions with marketing team
- Bi-weekly prompt improvements based on failures
- Monthly evaluation report (quality trends)
- Identify patterns for graduated autonomy
- Document learnings

Graduated Autonomy Criteria:

- Scenario: "High-confidence, low-risk recommendations"
 - Requirements:
 - 85% acceptance rate over 4 weeks
 - 90% positive impact when followed
 - Confidence score >0.85
 - Risk level = "low"
 - Action: Auto-approve (with notification)
-

5. Data Integration Layer

5.1 Collector Architecture

Each data source has a dedicated collector following this interface:

```

# src/data_collectors/base.py
from abc import ABC, abstractmethod
from typing import Generic, TypeVar
from datetime import datetime

T = TypeVar('T')

class BaseCollector(ABC, Generic[T]):
    """Base class for all data collectors"""

    def __init__(self, cache_ttl: int = 300):
        self.cache_ttl = cache_ttl
        self.client = None

    @abstractmethod
    async def collect(self, campaign_id: str, **kwargs) -> T:
        """Fetch data for a campaign"""
        pass

    @abstractmethod
    async def health_check(self) -> bool:
        """Check if data source is accessible"""
        pass

    async def collect_with_retry(
        self,
        campaign_id: str,
        max_retries: int = 3,
        **kwargs
    ) -> T | None:
        """Collect with exponential backoff retry"""
        from tenacity import retry, stop_after_attempt, wait_exponential

        @retry(
            stop=stop_after_attempt(max_retries),
            wait=wait_exponential(multiplier=1, min=2, max=10)
        )
        async def _collect():
            return await self.collect(campaign_id, **kwargs)

        try:
            return await _collect()
        except Exception as e:
            logger.error(f"Failed to collect data after {max_retries} retries",
                        error=str(e), campaign_id=campaign_id)
            return None

```

5.2 Campaign Metrics Collector

```

# src/data_collectors/campaign_collector.py
from datetime import datetime, timedelta

```

```
from pydantic import BaseModel
import httpx

class CampaignMetrics(BaseModel):
    """Campaign performance metrics"""
    campaign_id: str
    campaign_name: str

    # Core metrics
    spend: float
    impressions: int
    clicks: int
    conversions: int
    cpa: float
    ctr: float
    conversion_rate: float

    # Trends (vs. previous period)
    cpa_change_pct: float
    cpa_trend_days: int # Days CPA has been trending
    spend_change_pct: float

    # Context
    daily_budget: float
    budget_utilization: float
    impression_share: float

    # Time range
    period_start: datetime
    period_end: datetime
    comparison_period_start: datetime
    comparison_period_end: datetime

class CampaignMetricsCollector(BaseCollector[CampaignMetrics]):
    """Collect campaign performance data from ad platforms"""

    def __init__(self, api_key: str, cache_ttl: int = 300):
        super().__init__(cache_ttl)
        self.api_key = api_key
        self.client = httpx.AsyncClient(
            base_url="https://ads-api.example.com",
            headers={"Authorization": f"Bearer {api_key}"}
        )

    async def collect(
        self,
        campaign_id: str,
        lookback_days: int = 7
    ) -> CampaignMetrics:
        """
        Fetch campaign metrics for recent period + comparison period
        """

    Args:
        campaign_id: Campaign identifier
        lookback_days: Days of data to analyze

    Returns:
```

```

    CampaignMetrics with current and comparison data
"""
end_date = datetime.utcnow()
start_date = end_date - timedelta(days=lookback_days)
comparison_end = start_date
comparison_start = comparison_end - timedelta(days=lookback_days)

# Fetch current period
current_data = await self._fetch_metrics(
    campaign_id, start_date, end_date
)

# Fetch comparison period
comparison_data = await self._fetch_metrics(
    campaign_id, comparison_start, comparison_end
)

# Calculate changes
cpa_change_pct = (
    (current_data['cpa'] - comparison_data['cpa'])
    / comparison_data['cpa'] * 100
) if comparison_data['cpa'] > 0 else 0

spend_change_pct = (
    (current_data['spend'] - comparison_data['spend'])
    / comparison_data['spend'] * 100
) if comparison_data['spend'] > 0 else 0

# Detect trend duration
cpa_trend_days = await self._calculate_trend_duration(
    campaign_id, 'cpa', direction='increasing'
)

return CampaignMetrics(
    campaign_id=campaign_id,
    campaign_name=current_data['name'],
    spend=current_data['spend'],
    impressions=current_data['impressions'],
    clicks=current_data['clicks'],
    conversions=current_data['conversions'],
    cpa=current_data['cpa'],
    ctr=current_data['ctr'],
    conversion_rate=current_data['conversion_rate'],
    cpa_change_pct=cpa_change_pct,
    cpa_trend_days=cpa_trend_days,
    spend_change_pct=spend_change_pct,
    daily_budget=current_data['daily_budget'],
    budget_utilization=current_data['budget_utilization'],
    impression_share=current_data['impression_share'],
    period_start=start_date,
    period_end=end_date,
    comparison_period_start=comparison_start,
    comparison_period_end=comparison_end
)

async def _fetch_metrics(
    self,

```

```

campaign_id: str,
start_date: datetime,
end_date: datetime
) -> dict:
    """Internal: Fetch raw metrics from API"""
    response = await self.client.get(
        f"/campaigns/{campaign_id}/metrics",
        params={
            "start_date": start_date.isoformat(),
            "end_date": end_date.isoformat()
        }
    )
    response.raise_for_status()
    return response.json()

async def _calculate_trend_duration(
    self,
    campaign_id: str,
    metric: str,
    direction: str
) -> int:
    """Calculate how many consecutive days metric has been trending"""
    # Fetch daily data for last 30 days
    daily_data = await self._fetch_daily_metrics(campaign_id, days=30)

    trend_days = 0
    for i in range(len(daily_data) - 1):
        current = daily_data[i][metric]
        previous = daily_data[i + 1][metric]

        if direction == 'increasing' and current > previous:
            trend_days += 1
        elif direction == 'decreasing' and current < previous:
            trend_days += 1
        else:
            break

    return trend_days

async def health_check(self) -> bool:
    """Check if API is accessible"""
    try:
        response = await self.client.get("/health")
        return response.status_code == 200
    except Exception:
        return False

```

5.3 Creative Performance Collector

```
# src/data_collectors/creative_collector.py
from enum import Enum
```

```

class CreativeTrend(str, Enum):
    IMPROVING = "improving"
    STABLE = "stable"
    DECLINING = "declining"

class CreativeMetrics(BaseModel):
    """Creative performance and fatigue indicators"""
    campaign_id: str
    creative_id: str
    creative_name: str

    # Performance
    ctr: float
    engagement_rate: float
    video_completion_rate: float | None

    # Trends
    ctr_trend: CreativeTrend
    ctr_change_pct: float

    # Fatigue indicators
    creative_age_days: int
    average_frequency: float # Impressions per unique user
    frequency_trend: str # "increasing", "stable", "decreasing"

    # Context
    total_impressions: int
    unique_reach: int

    # Freshness
    last_updated: datetime

class CreativeMetricsCollector(BaseCollector[CreativeMetrics]):
    """Collect creative performance and fatigue signals"""

    async def collect(self, campaign_id: str) -> list[CreativeMetrics]:
        """
        Fetch all creative metrics for a campaign

        Returns list (one per creative)
        """
        # Implementation similar to CampaignMetricsCollector
        pass

    def analyze_fatigue(self, metrics: CreativeMetrics) -> dict:
        """
        Analyze if creative is showing fatigue

        Returns:
        {
            "is_fatigued": bool,
            "confidence": float,
            "signals": list[str]
        }
        """
        signals = []
        fatigue_score = 0

```

```

# Check age
if metrics.creative_age_days > 30:
    signals.append(f"Creative is {metrics.creative_age_days} days old")
    fatigue_score += 0.3

# Check CTR decline
if metrics.ctr_trend == CreativeTrend.DECLINING:
    signals.append(f"CTR declining: {metrics.ctr_change_pct:.1f}% drop")
    fatigue_score += 0.4

# Check frequency
if metrics.average_frequency > 5.0:
    signals.append(f"High frequency: {metrics.average_frequency:.1f}")
    fatigue_score += 0.3

return {
    "is_fatigued": fatigue_score > 0.5,
    "confidence": fatigue_score,
    "signals": signals
}

```

5.4 Competitor Signals Collector

```

# src/data_collectors/competitor_collector.py

class CompetitorSignals(BaseModel):
    """Competitor activity indicators"""
    industry: str
    keywords: list[str]

    # Activity metrics
    active_competitors_count: int
    new_competitors_count: int # Last 7 days
    activity_change_pct: float # Overall market activity

    # Bidding indicators
    avg_bid_increase_pct: float
    top_competitor_bids: list[dict]

    # Ad creative activity
    new_ads_count: int
    creative_refreshes: int

    # Context
    analysis_period_days: int
    last_updated: datetime

class CompetitorSignalsCollector(BaseCollector[CompetitorSignals]):
    """
    Collect competitor intelligence from tools like:
    - SEMrush API
    """

```

```

- SpyFu API
- Google Ads Auction Insights
- Meta Ad Library
"""

async def collect(
    self,
    campaign_id: str,
    industry: str,
    keywords: list[str]
) -> CompetitorSignals:
    """Aggregate competitor signals from multiple sources"""

    # Example: Fetch from multiple sources in parallel
    results = await asyncio.gather(
        self._fetch_semrush_data(keywords),
        self._fetch_auction_insights(campaign_id),
        self._fetch_ad_library_trends(industry),
        return_exceptions=True
    )

    # Aggregate and normalize
    # ...

    return CompetitorSignals(...)

```

5.5 Parallel Context Collection

```

# src/agent/context_builder.py
import asyncio
from typing import TypedDict

class CampaignContext(TypedDict):
    """Complete context for a campaign analysis"""
    campaign_id: str
    campaign_metrics: CampaignMetrics
    creative_metrics: list[CreativeMetrics]
    competitor_signals: CompetitorSignals
    audience_analytics: AudienceAnalytics
    historical_patterns: HistoricalPatterns

class ContextBuilder:
    """Orchestrate parallel data collection from all sources"""

    def __init__(
        self,
        campaign_collector: CampaignMetricsCollector,
        creative_collector: CreativeMetricsCollector,
        competitor_collector: CompetitorSignalsCollector,
        audience_collector: AudienceAnalyticsCollector,
        historical_analyzer: HistoricalPatternAnalyzer
    ):

```

```
self.campaign_collector = campaign_collector
self.creative_collector = creative_collector
self.competitor_collector = competitor_collector
self.audience_collector = audience_collector
self.historical_analyzer = historical_analyzer

async def build_context(
    self,
    campaign_id: str
) -> CampaignContext:
    """
    Collect all context data in parallel

    Handles failures gracefully - partial context is OK
    """
    # Execute all collectors in parallel
    results = await asyncio.gather(
        self.campaign_collector.collect_with_retry(campaign_id),
        self.creative_collector.collect_with_retry(campaign_id),
        self._collect_competitor_data(campaign_id),
        self.audience_collector.collect_with_retry(campaign_id),
        self.historical_analyzer.analyze(campaign_id),
        return_exceptions=True # Don't fail if one collector fails
    )

    # Unpack results (handle None from failures)
    campaign_metrics, creative_metrics, competitor_signals, \
        audience_analytics, historical_patterns = results

    # Log any failures but continue
    for i, result in enumerate(results):
        if isinstance(result, Exception):
            logger.warning(
                f"Collector {i} failed",
                error=str(result),
                campaign_id=campaign_id
            )

    return CampaignContext(
        campaign_id=campaign_id,
        campaign_metrics=campaign_metrics,
        creative_metrics=creative_metrics or [],
        competitor_signals=competitor_signals,
        audience_analytics=audience_analytics,
        historical_patterns=historical_patterns
    )

async def _collect_competitor_data(self, campaign_id: str):
    """Helper to get industry/keywords from campaign first"""
    # Fetch campaign details to get industry and keywords
    campaign_info = await self._get_campaign_info(campaign_id)

    return await self.competitor_collector.collect_with_retry(
        campaign_id=campaign_id,
        industry=campaign_info['industry'],
        keywords=campaign_info['keywords']
    )
```

Performance Expectations:

- Parallel collection: ~5-10s total (vs. 20-30s sequential)
- Graceful degradation: Agent can reason with partial context
- Caching: Similar campaigns share competitor/historical data

6. Agent Design & Prompting

6.1 LangGraph Workflow

```
# src/agent/workflow.py
from langgraph.graph import StateGraph, END
from typing import Literal

def create_marketing_agent() -> StateGraph:
    """Create the marketing reasoning agent workflow"""

    # Initialize workflow
    workflow = StateGraph(AgentState)

    # Add nodes
    workflow.add_node("collect_context", collect_context_node)
    workflow.add_node("analyze_signals", analyze_signals_node)
    workflow.add_node("generate_recommendation", generate_recommendation_node)
    workflow.add_node("critique", critique_recommendation_node)
    workflow.add_node("validate", validate_output_node)

    # Define edges
    workflow.set_entry_point("collect_context")
    workflow.add_edge("collect_context", "analyze_signals")
    workflow.add_edge("analyze_signals", "generate_recommendation")
    workflow.add_edge("generate_recommendation", "critique")

    # Conditional edge: Regenerate if needed
    workflow.add_conditional_edges(
        "critique",
        should_regenerate,
        {
            "regenerate": "analyze_signals",
            "validate": "validate"
        }
    )

    workflow.add_edge("validate", END)

    return workflow.compile()
```

```

# Node implementations

async def collect_context_node(state: AgentState) -> AgentState:
    """Collect all campaign context in parallel"""
    context_builder = ContextBuilder(...) # Inject dependencies
    context = await context_builder.build_context(state["campaign_id"])

    return {
        **state,
        "campaign_metrics": context["campaign_metrics"],
        "creative_metrics": context["creative_metrics"],
        "competitor_signals": context["competitor_signals"],
        "audience_analytics": context["audience_analytics"],
        "historical_patterns": context["historical_patterns"]
    }

async def analyze_signals_node(state: AgentState) -> AgentState:
    """LLM-powered signal analysis"""
    from langchain_openai import ChatOpenAI
    from langchain.prompts import ChatPromptTemplate

    llm = ChatOpenAI(model="gpt-4o", temperature=0.1)

    prompt = ChatPromptTemplate.from_messages([
        ("system", SIGNAL_ANALYSIS_PROMPT),
        ("user", format_context_for_llm(state))
    ])

    response = await llm.ainvoke(prompt.format_messages())

    # Parse structured response
    analysis = parse_signal_analysis(response.content)

    return {
        **state,
        "signal_analysis": analysis["analysis"],
        "root_cause_hypothesis": analysis["root_cause"],
        "confidence_score": analysis["confidence"]
    }

def should_regenerate(state: AgentState) -> Literal["regenerate", "validate"]:
    """Decide if recommendation needs regeneration"""
    iteration_count = state.get("iteration_count", 0)
    confidence = state.get("confidence_score", 0)
    critique_feedback = state.get("critique_feedback")

    # Max 2 iterations
    if iteration_count >= 2:
        return "validate"

    # Regenerate if low confidence or critical issues
    if confidence < 0.6 or (critique_feedback and "CRITICAL" in critique_feedback):
        return "regenerate"

    return "validate"

```

6.2 Core System Prompt

```
# src/agent/prompts.py

SIGNAL_ANALYSIS_PROMPT = """You are an expert marketing performance analyst specializing in paid advertising campaigns.

Your task is to analyze campaign performance data and identify the root cause of any performance changes.

## Available Workflows

You can recommend one of the following actions:
1. **Creative Refresh** - Replace or update ad creatives
2. **Audience Expansion** - Broaden targeting to reach new users
3. **Bid Adjustment** - Increase or decrease bidding strategy
4. **Campaign Pause** - Stop campaign temporarily
5. **Budget Reallocation** - Shift budget to better-performing segments
6. **Continue Monitoring** - No action needed, continue observing

## Analysis Framework

Consider these factors in your analysis:

#### 1. Metric Changes
- Magnitude: How significant is the change? (>10% = significant, >30% = critical)
- Duration: How many days has the trend persisted? (3+ days = established trend)
- Velocity: Is the change accelerating or stabilizing?

#### 2. Signal Correlation
- Do multiple metrics point to the same issue?
- Are there contradictory signals that need reconciliation?
- What's the most parsimonious explanation?

#### 3. Root Cause Hypotheses

**Creative Fatigue:** 
- Declining CTR despite stable audience metrics
- Increasing frequency (>5 impressions per user)
- Creative older than 30 days
- Engagement rates declining

**Audience Saturation:** 
- High impression share (>90%)
- Declining reach despite consistent budget
- Frequency increasing
- CPM increasing while CTR stable

**Competitive Pressure:** 
- CPA/CPM increasing while creative performance stable
- Competitor activity up significantly (>20%)
- Impression share declining
- Auction dynamics shifting
```

```
**Seasonal/External Factors:**  
- Industry-wide trends  
- Market events  
- Historical patterns (e.g., day of week, time of year)
```

```
#### 4. Historical Context  
- How has this campaign performed previously?  
- What actions worked in similar situations?  
- Are there recurring patterns?
```

```
## Output Requirements
```

Provide your analysis in the following structured format:

```
```json  
{
 "signal_analysis": "Detailed narrative of key signals and their relationships",
 "root_cause": "Primary hypothesis for performance change",
 "supporting_evidence": [
 "Evidence point 1",
 "Evidence point 2"
],
 "contradictory_evidence": [
 "Counter-signal 1 (if any)"
],
 "confidence": 0.85,
 "confidence_reasoning": "Why this confidence level"
}
```

## Guidelines

1. **Be data-driven:** Base conclusions on evidence, not assumptions
2. **Consider alternatives:** What else could explain the data?
3. **Assess confidence honestly:** Don't overstate certainty
4. **Think causally:** Correlation ≠ causation
5. **Be specific:** Vague reasoning helps no one

## Example

### Input:

- CPA increased 30% (from 45 to 58.50) over 3 days
- Creative CTR stable at 2.8%
- Impression share 95%
- Competitor activity up 40%

- Frequency at 4.2 (not fatigued)

## Good Analysis:

```
{
 "signal_analysis": "CPA spike coincides with increased competitive activity rather than creative fatigue. Creative CTR remains stable at 2.8%, indicating ads still resonate. Audience is near saturation (95% impression share), limiting expansion options. Frequency at 4.2 is within healthy range. Competitor analysis shows 3 new entrants and 25% avg bid increase in past week. This suggests auction environment has become more competitive, driving up costs without internal performance degradation.",
 "root_cause": "Increased competitive pressure in auction environment",
 "supporting_evidence": [
 "Competitor activity +40% (3 new entrants)",
 "CPA increase timing aligns with competitive surge",
 "No creative fatigue signals (stable CTR, healthy frequency)",
 "Audience near limit (95% IS) - saturation not the issue"
],
 "contradictory_evidence": [],
 "confidence": 0.82,
 "confidence_reasoning": "Strong signal correlation and clear timing alignment. Only uncertainty is whether competitive pressure will persist or is temporary."
}
```

Now analyze the provided campaign data."""

RECOMMENDATION\_GENERATION\_PROMPT = """Based on your signal analysis, generate a specific, actionable recommendation.

## Input

---

You will receive:

- Your signal analysis and root cause hypothesis
- Campaign context and constraints
- Historical effectiveness of different actions

## Output Format

---

```
{
 "recommended_workflow": "Bid Adjustment",
 "specific_action": "Increase bid by 15% from current $2.50 to $2.88",
```

```

"reasoning": "Detailed explanation linking analysis to recommendation",
"expected_impact": "Restore CPA to $47-49 range within 2-3 days",
"risk_level": "medium",
"risk_factors": [
 "If competition doesn't stabilize, may need further increases",
 "Potential for temporary overspend if competitors drop out suddenly"
],
"alternative_actions": [
 {
 "workflow": "Continue Monitoring",
 "why_not_recommended": "Delaying action risks further CPA degradation"
 },
 {
 "workflow": "Creative Refresh",
 "why_not_recommended": "Creative showing no fatigue signals; wouldn't address competitive pressure"
 }
],
"success_metrics": [
 "CPA returns to <$50 within 3 days",
 "Impression share maintained above 90%",
 "CTR remains stable or improves"
],
"confidence": 0.82
}

```

## Recommendation Guidelines

---

- Be specific:** Don't just say "adjust bids" - say how much
- Explain the logic:** Connect dots from analysis to action
- Set expectations:** What should happen if this works?
- Acknowledge risks:** What could go wrong?
- Provide alternatives:** What else could we do?
- Define success:** How will we know if it worked?

## Action Suitability Matrix

---

Root Cause	Best Action	Why
Creative fatigue	Creative Refresh	Directly addresses ad staleness
Audience saturation	Audience Expansion	Reaches new users
Competitive pressure	Bid Adjustment	Competes in auction

Root Cause	Best Action	Why
Poor targeting	Audience Refinement	Improves relevance
Fundamental issue	Campaign Pause	Prevents waste
Normal variance	Continue Monitoring	Avoids overreaction

Now generate your recommendation.""""

CRITIQUE\_PROMPT = """You are a quality assurance analyst reviewing a marketing recommendation.

## Your Task

Evaluate the recommendation for:

- Logical consistency:** Does the recommendation follow from the analysis?
- Specificity:** Is the action concrete and actionable?
- Risk assessment:** Are risks properly identified and weighted?
- Alternative consideration:** Were other options fairly evaluated?
- Success criteria:** Can we measure if this worked?

## Output

```
{
 "is_satisfactory": true,
 "issues": [
 {
 "severity": "minor",
 "issue": "Expected impact timeframe is vague",
 "suggestion": "Specify exact number of days"
 }
],
 "strengths": [
 "Clear connection between competitive pressure and bid adjustment",
 "Specific bid increase percentage provided"
],
 "overall_assessment": "Strong recommendation with minor improvements possible"
}
```

Severity levels:

- **CRITICAL:** Logical flaw, unsafe recommendation, or missing key info
- **major:** Significant improvement needed
- **minor:** Small refinement would help

If any CRITICAL issues are found, the recommendation should be regenerated.

Now critique the recommendation.""""

```
6.3 Structured Output Models

```python
# src/agent/models.py
from pydantic import BaseModel, Field
from enum import Enum


class WorkflowType(str, Enum):
    CREATIVE_REFRESH = "Creative Refresh"
    AUDIENCE_EXPANSION = "Audience Expansion"
    BID_ADJUSTMENT = "Bid Adjustment"
    CAMPAIGN_PAUSE = "Campaign Pause"
    BUDGET_REALLOCATION = "Budget Reallocation"
    CONTINUE_MONITORING = "Continue Monitoring"


class RiskLevel(str, Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"


class AlternativeAction(BaseModel):
    workflow: WorkflowType
    why_not_recommended: str


class Recommendation(BaseModel):
    """Structured recommendation output"""

    # Core recommendation
    recommended_workflow: WorkflowType
    specific_action: str = Field(
        description="Concrete, actionable instruction (e.g., 'Increase bid by 15% to $2.88')"
    )

    # Reasoning
    reasoning: str = Field(
        description="Detailed explanation connecting analysis to recommendation"
    )
    signal_analysis: str = Field(
        description="Summary of key performance signals analyzed"
    )
    root_cause: str = Field(
        description="Primary hypothesis for performance change"
    )

```

```

# Impact & Risk
expected_impact: str = Field(
    description="Predicted outcome if recommendation is followed"
)
risk_level: RiskLevel
risk_factors: list[str] = Field(
    description="Potential risks or complications"
)

# Alternatives
alternative_actions: list[AlternativeAction] = Field(
    description="Other options considered and why not chosen"
)

# Success criteria
success_metrics: list[str] = Field(
    description="How to measure if recommendation worked"
)

# Confidence
confidence: float = Field(
    ge=0.0, le=1.0,
    description="Confidence score (0-1)"
)
confidence_reasoning: str = Field(
    description="Explanation of confidence level"
)

# Metadata
campaign_id: str
generated_at: datetime
model_version: str

# Use with LangChain structured output
from langchain.output_parsers import PydanticOutputParser

parser = PydanticOutputParser(pydantic_object=Recommendation)

```

6.4 Prompt Versioning Strategy

```

# src/agent/prompt_registry.py
import yaml
from pathlib import Path

class PromptRegistry:
    """Version-controlled prompt management"""

    def __init__(self, prompts_dir: Path):
        self.prompts_dir = prompts_dir
        self._cache = {}

    def get_prompt(

```

```

    self,
    name: str,
    version: str = "latest"
) -> str:
    """
    Load prompt from versioned YAML files

    prompts/
        signal_analysis/
            v1.yaml
            v2.yaml
            latest.yaml (symlink)
    """
    cache_key = f"{name}:{version}"

    if cache_key in self._cache:
        return self._cache[cache_key]

    prompt_path = self.prompts_dir / name / f"{version}.yaml"

    with open(prompt_path) as f:
        prompt_config = yaml.safe_load(f)

    prompt_text = prompt_config['prompt']
    self._cache[cache_key] = prompt_text

    return prompt_text

def list_versions(self, name: str) -> list[str]:
    """List all versions of a prompt"""
    prompt_dir = self.prompts_dir / name
    return [
        p.stem for p in prompt_dir.glob("*.yaml")
        if p.stem != "latest"
    ]

def compare_versions(
    self,
    name: str,
    v1: str,
    v2: str
) -> dict:
    """
    Compare two prompt versions on golden dataset
    Used for A/B testing prompts
    """
    # Run eval on both versions
    results_v1 = run_evaluation(name, v1)
    results_v2 = run_evaluation(name, v2)

    return {
        "v1": results_v1,
        "v2": results_v2,
        "winner": results_v2 if results_v2.score > results_v1.score else
results_v1
    }

```

Prompt Version Control in Git:

```
# prompts/signal_analysis/v1.yaml
version: "1.0"
created: "2026-02-11"
author: "sudip"
description: "Initial signal analysis prompt"
changelog:
  - "First version based on requirements"

prompt: |
  You are an expert marketing performance analyst...
  [full prompt text]

metadata:
  model: "gpt-4o"
  temperature: 0.1
  max_tokens: 2000
```

7. Evaluation Framework

7.1 Evaluation Strategy

Multi-Layered Evaluation:

1. **Unit Tests** - Individual components work correctly
2. **Golden Set Tests** - Agent matches human decisions on historical cases
3. **LLM-as-Judge** - Reasoning quality assessment
4. **Outcome Tracking** - Real-world impact measurement
5. **Regression Tests** - Prevent quality degradation over time

7.2 Golden Dataset Creation

```
# src/evaluation/golden_dataset.py
from pydantic import BaseModel

class GoldenExample(BaseModel):
    """A test case with known correct answer"""
    example_id: str
    campaign_id: str
    campaign_name: str
```

```

# Input: Campaign context at decision time
context: CampaignContext

# Expected output: What human decided
human_decision: WorkflowType
human_reasoning: str
confidence: float # How clear-cut was decision

# Actual outcome: What happened after action
outcome: dict # {"cpa_change_pct": -15, "success": True}

# Metadata
decision_date: datetime
decided_by: str # Marketing exec who made call
tags: list[str] # ["competitive_pressure", "bid_adjustment"]

class GoldenDatasetBuilder:
    """Build golden dataset from historical decisions"""

    async def create_from_historical_data(
        self,
        start_date: datetime,
        end_date: datetime
    ) -> list[GoldenExample]:
        """
        Extract historical scenarios where:
        1. Marketing exec made a decision
        2. We have context snapshot
        3. We can measure outcome
        """

        # Query database for historical recommendations
        historical_recs = await db.query(
            """
            SELECT
                r.id,
                r.campaign_id,
                r.context,
                r.human_decision,
                r.human_reasoning,
                r.decision_date,
                r.decided_by,
                o.outcome_metrics
            FROM recommendations r
            JOIN outcomes o ON r.id = o.recommendation_id
            WHERE r.decision_date BETWEEN :start AND :end
            AND r.human_decision IS NOT NULL
            AND o.outcome_metrics IS NOT NULL
            """,
            start=start_date,
            end=end_date
        )

        golden_examples = []
        for rec in historical_recs:
            # Tag example for categorization
            tags = self._categorize_scenario(rec)

```

```

        # Calculate decision confidence based on outcome
        confidence = self._assess_decision_confidence(rec)

        example = GoldenExample(
            example_id=rec['id'],
            campaign_id=rec['campaign_id'],
            context=rec['context'],
            human_decision=rec['human_decision'],
            human_reasoning=rec['human_reasoning'],
            confidence=confidence,
            outcome=rec['outcome_metrics'],
            decision_date=rec['decision_date'],
            decided_by=rec['decided_by'],
            tags=tags
        )

        golden_examples.append(example)

    return golden_examples

def _categorize_scenario(self, rec: dict) -> list[str]:
    """Tag scenario with relevant categories"""
    tags = []

    context = rec['context']

    # Root cause tags
    if context['campaign_metrics']['cpa_change_pct'] > 20:
        tags.append("high_cpa_increase")

    if context['competitor_signals']['activity_change_pct'] > 30:
        tags.append("competitive_pressure")

    creative_fatigued = any(
        c.get('is_fatigued') for c in context.get('creative_metrics', [])
    )
    if creative_fatigued:
        tags.append("creative_fatigue")

    # Action tags
    tags.append(f'action_{rec['human_decision'].lower().replace(' ', '_')}')

    return tags

def _assess_decision_confidence(self, rec: dict) -> float:
    """
    Estimate how clear-cut the decision was
    Based on outcome and alternative viability
    """
    outcome = rec['outcome_metrics']

    # If action led to significant improvement → high confidence
    if outcome.get('success') and outcome.get('improvement_pct', 0) > 15:
        return 0.9

    # If marginal improvement → medium confidence

```

```

if outcome.get('success') and outcome.get('improvement_pct', 0) > 5:
    return 0.7

# If no improvement → decision was less obvious or wrong
return 0.5

```

7.3 Evaluation Metrics

```

# src/evaluation/metrics.py
from sklearn.metrics import precision_score, recall_score, f1_score
import numpy as np

class EvaluationMetrics:
    """Calculate evaluation metrics for recommendations"""

    @staticmethod
    def calculate_agreement_rate(
        agent_decisions: list[WorkflowType],
        human_decisions: list[WorkflowType]
    ) -> float:
        """
        What % of time does agent match human decision?
        Primary quality metric
        """

        matches = sum(
            1 for a, h in zip(agent_decisions, human_decisions)
            if a == h
        )
        return matches / len(agent_decisions)

    @staticmethod
    def calculate_precision_by_workflow(
        agent_decisions: list[WorkflowType],
        human_decisions: list[WorkflowType]
    ) -> dict[WorkflowType, float]:
        """
        For each workflow type, calculate precision:
        Of recommendations agent made, how many did humans agree with?
        """

        workflows = list(WorkflowType)

        # Convert to binary labels for sklearn
        results = {}
        for workflow in workflows:
            y_true = [1 if h == workflow else 0 for h in human_decisions]
            y_pred = [1 if a == workflow else 0 for a in agent_decisions]

            if sum(y_pred) > 0: # Avoid division by zero
                precision = precision_score(y_true, y_pred, zero_division=0)
                results[workflow] = precision
            else:
                results[workflow] = None

```

```

        return results

@staticmethod
def calculate_confidence_calibration(
    confidences: list[float],
    correct: list[bool]
) -> dict:
    """
    How well-calibrated are confidence scores?

    If agent says 80% confident, are they right 80% of the time?
    """

    # Bin confidences
    bins = np.linspace(0, 1, 11)  # 0-10%, 10-20%, ..., 90-100%

    calibration = {}
    for i in range(len(bins) - 1):
        bin_low, bin_high = bins[i], bins[i+1]

        # Find predictions in this confidence range
        in_bin = [
            correct[j] for j in range(len(confidences))
            if bin_low <= confidences[j] < bin_high
        ]

        if in_bin:
            actual_accuracy = sum(in_bin) / len(in_bin)
            expected_confidence = (bin_low + bin_high) / 2

            calibration[f"{int(bin_low*100)}-{int(bin_high*100)}%"] = {
                "expected": expected_confidence,
                "actual": actual_accuracy,
                "count": len(in_bin),
                "calibration_error": abs(actual_accuracy - expected_confidence)
            }

    return calibration

@staticmethod
def calculate_outcome_metrics(
    recommendations: list[dict]
) -> dict:
    """
    Track real-world impact of recommendations

    Args:
        recommendations: List of dicts with {
            "recommended_workflow": str,
            "was_accepted": bool,
            "outcome": dict  # If accepted and executed
        }
    """

    accepted = [r for r in recommendations if r['was_accepted']]

    if not accepted:
        return {"error": "No accepted recommendations to evaluate"}

```

```

with_outcomes = [
    r for r in accepted
    if r.get('outcome') is not None
]

return {
    "total_recommendations": len(recommendations),
    "acceptance_rate": len(accepted) / len(recommendations),
    "positive_impact_rate": sum(
        1 for r in with_outcomes
        if r['outcome'].get('improvement_pct', 0) > 0
    ) / len(with_outcomes) if with_outcomes else 0,
    "avg_improvement_pct": np.mean([
        r['outcome'].get('improvement_pct', 0)
        for r in with_outcomes
    ]) if with_outcomes else 0,
    "samples_with_outcomes": len(with_outcomes)
}

```

7.4 LLM-as-Judge Evaluation

```

# src/evaluation/llm_judge.py
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate

class LLMJudge:
    """Use GPT-4 to evaluate reasoning quality"""

    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4o", temperature=0)

    async def evaluate_reasoning(
        self,
        recommendation: Recommendation,
        context: CampaignContext
    ) -> dict:
        """
        Evaluate quality of reasoning independent of correctness

        Measures:
        - Logical consistency
        - Evidence support
        - Consideration of alternatives
        - Clarity of explanation
        """
        prompt = ChatPromptTemplate.from_messages([
            ("system", self._get_judge_prompt()),
            ("user", self._format_eval_input(recommendation, context))
        ])

        response = await self.llm.invoke(prompt.format_messages())

```

```

# Parse structured evaluation
evaluation = parse_judge_response(response.content)

return evaluation

def _get_judge_prompt(self) -> str:
    return """You are evaluating the quality of a marketing recommendation.

```

Rate on scale 1-5 for each criterion:

1. **Logical Consistency** (1-5)
 - Does the recommendation follow logically from the analysis?
 - Are there gaps or leaps in reasoning?
 - Score: 5 = flawless logic, 1 = major logical issues

2. **Evidence Support** (1-5)
 - Is each claim backed by evidence from the data?
 - Are numbers and metrics used appropriately?
 - Score: 5 = well-evidenced, 1 = unsupported claims

3. **Alternative Consideration** (1-5)
 - Were reasonable alternatives considered?
 - Is the explanation for rejecting alternatives sound?
 - Score: 5 = thorough, 1 = ignored alternatives

4. **Clarity** (1-5)
 - Is the reasoning clear and easy to follow?
 - Could a non-expert understand it?
 - Score: 5 = very clear, 1 = confusing

Output format:

```

```json
{
 "logical_consistency": 4,
 "evidence_support": 5,
 "alternative_consideration": 3,
 "clarity": 4,
 "overall_score": 4.0,
 "strengths": ["List of specific strengths"],
 "weaknesses": ["List of specific weaknesses"],
 "explanation": "Brief overall assessment"
}

```

"""

## 7.5 promptfoo Integration

```

promptfoo-config.yaml
prompts:
 - file://prompts/signal_analysis/v1.yaml
 - file://prompts/signal_analysis/v2.yaml

```

```

providers:
 - id: openai:gpt-4o
 config:
 temperature: 0.1
 - id: anthropic:claude-sonnet-4-5-20250929
 config:
 temperature: 0.1

tests:
 # Golden dataset test cases
 - vars:
 campaign_id: "campaign_001"
 context: file://test_data/competitive_pressure_case.json
 assert:
 - type: is-json
 - type: python
 value: file://evaluators/check_workflow_match.py
 - type: llm-rubric
 value: |
 The recommendation should identify competitive pressure as root cause
 and suggest bid adjustment as the solution.
 - type: similar
 value: "Competitor activity increased significantly"
 threshold: 0.8

 - vars:
 campaign_id: "campaign_002"
 context: file://test_data/creative_fatigue_case.json
 assert:
 - type: python
 value: file://evaluators/check_workflow_match.py
 - type: contains
 value: "creative refresh"
 - type: llm-rubric
 value: |
 Should identify declining CTR and high frequency as creative fatigue
signals.

Custom evaluator
evaluators/check_workflow_match.py
def evaluate(output, context):
 """Check if agent recommendation matches expected workflow"""
 import json

 try:
 recommendation = json.loads(output)
 expected_workflow = context['vars']['expected_workflow']

 if recommendation['recommended_workflow'] == expected_workflow:
 return {
 'pass': True,
 'score': 1.0,
 'reason': 'Workflow matches expected decision'
 }
 else:
 return {

```

```

 'pass': False,
 'score': 0.0,
 'reason': f"Expected {expected_workflow}, got
{recommendation['recommended_workflow']}}"
 }
except Exception as e:
 return {
 'pass': False,
 'score': 0.0,
 'reason': f'Parse error: {str(e)}'
 }

```

## Run Evaluation:

```

Run all tests
promptfoo eval

Compare prompts
promptfoo eval --prompts prompts/signal_analysis/v*.yaml

Run specific test
promptfoo eval --filter "competitive_pressure"

View results in UI
promptfoo view

```

## 7.6 Continuous Evaluation in CI

```

.github/workflows/eval.yml
name: Continuous Evaluation

on:
 pull_request:
 paths:
 - 'prompts/**'
 - 'src/agent/**'

jobs:
 evaluate:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Set up Python
 uses: actions/setup-python@v4
 with:
 python-version: '3.11'

 - name: Install dependencies

```

```

run: |
 pip install -r requirements.txt
 npm install -g promptfoo

- name: Run golden set evaluation
 env:
 OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
 run: |
 python -m evaluation.run_golden_set

- name: Run promptfoo tests
 env:
 OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
 run: |
 promptfoo eval

- name: Check quality threshold
 run: |
 python -m evaluation.check_threshold

- name: Post results to PR
 uses: actions/github-script@v6
 with:
 script: |
 const fs = require('fs');
 const results = JSON.parse(fs.readFileSync('eval_results.json'));

 github.rest.issues.createComment({
 issue_number: context.issue.number,
 owner: context.repo.owner,
 repo: context.repo.repo,
 body: `## Evaluation Results\n\n` +
 `Agreement Rate: ${results.agreement_rate}%\n` +
 `Threshold: 70% ✓\n\n` +
 `Passed: ${results.passed_count}/${results.total_count}`
 });

```

- name: Fail if below threshold
 

```

run: |
 python -c "
 import json
 with open('eval_results.json') as f:
 results = json.load(f)
 if results['agreement_rate'] < 70:
 print(f'✖ Agreement rate {results[\"agreement_rate\"]}% below 70%
threshold')
 exit(1)
 print(f'✓ Agreement rate {results[\"agreement_rate\"]}% meets threshold'
"

```

## 8. Production Infrastructure

## 8.1 Database Schema

```
-- PostgreSQL schema
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Campaigns table
CREATE TABLE campaigns (
 id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
 campaign_id VARCHAR(255) UNIQUE NOT NULL,
 campaign_name VARCHAR(500) NOT NULL,
 platform VARCHAR(50) NOT NULL, -- 'google_ads', 'meta_ads', etc.
 industry VARCHAR(100),
 keywords TEXT[],
 created_at TIMESTAMP DEFAULT NOW(),
 updated_at TIMESTAMP DEFAULT NOW()
);

-- Recommendations table
CREATE TABLE recommendations (
 id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
 campaign_id VARCHAR(255) NOT NULL REFERENCES campaigns(campaign_id),

 -- Context (JSONB for flexibility)
 context JSONB NOT NULL,

 -- Recommendation
 recommended_workflow VARCHAR(100) NOT NULL,
 specific_action TEXT NOT NULL,
 reasoning TEXT NOT NULL,
 signal_analysis TEXT NOT NULL,
 root_cause TEXT NOT NULL,
 expected_impact TEXT,
 risk_level VARCHAR(20),
 risk_factors JSONB,
 alternative_actions JSONB,
 success_metrics JSONB,
 confidence FLOAT CHECK (confidence >= 0 AND confidence <= 1),
 confidence_reasoning TEXT,

 -- Decision tracking
 human_decision VARCHAR(100), -- NULL if pending, 'approved' or 'rejected'
 decided_by VARCHAR(255),
 decided_at TIMESTAMP,
 decision_feedback TEXT,

 -- Outcome tracking
 outcome_metrics JSONB,
 outcome_recorded_at TIMESTAMP,

 -- Metadata
 model_version VARCHAR(50),
 prompt_version VARCHAR(50),
 generated_at TIMESTAMP DEFAULT NOW(),
```

```

-- Indexes
created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_recommendations_campaign ON recommendations(campaign_id);
CREATE INDEX idx_recommendations_decision ON recommendations(human_decision);
CREATE INDEX idx_recommendations_created ON recommendations(created_at);
CREATE INDEX idx_recommendations_context_gin ON recommendations USING gin(context);

-- Evaluation results
CREATE TABLE evaluation_runs (
 id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
 run_type VARCHAR(50) NOT NULL, -- 'golden_set', 'llm_judge', 'outcome'
 dataset_version VARCHAR(50),
 prompt_version VARCHAR(50),
 model_version VARCHAR(50),

 -- Results
 total_examples INT NOT NULL,
 passed_examples INT NOT NULL,
 metrics JSONB NOT NULL,

 -- Metadata
 triggered_by VARCHAR(100), -- 'ci', 'manual', 'scheduled'
 git_commit VARCHAR(40),
 run_at TIMESTAMP DEFAULT NOW()
);

-- Agent execution logs (for debugging)
CREATE TABLE agent_executions (
 id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
 recommendation_id UUID REFERENCES recommendations(id),

 -- Execution trace
 workflow_state JSONB NOT NULL,
 execution_path TEXT[],
 node_outputs JSONB,

 -- Performance
 total_duration_ms INT,
 llm_calls_count INT,
 total_tokens INT,
 total_cost_usd DECIMAL(10, 6),

 -- Status
 status VARCHAR(20) NOT NULL, -- 'success', 'error', 'timeout'
 error_message TEXT,

 executed_at TIMESTAMP DEFAULT NOW()
);

-- Data collector health
CREATE TABLE collector_health (
 id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
 collector_name VARCHAR(100) NOT NULL,
 status VARCHAR(20) NOT NULL, -- 'healthy', 'degraded', 'down'
 last_check TIMESTAMP DEFAULT NOW(),

```

```
 error_message TEXT,
 response_time_ms INT
);
```

## 8.2 API Implementation

```
src/api/main.py
from fastapi import FastAPI, HTTPException, Depends, BackgroundTasks
from fastapi.middleware.cors import CORSMiddleware
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from slowapi.errors import RateLimitExceeded
import structlog

from .routers import recommendations, campaigns, evaluations
from .dependencies import get_current_user, get_db
from .monitoring import setup_prometheus

logger = structlog.get_logger()

Initialize app
app = FastAPI(
 title="Marketing Agent API",
 version="0.1.0",
 docs_url="/api/docs"
)

Rate limiting
limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

CORS
app.add_middleware(
 CORSMiddleware,
 allow_origins=["http://localhost:3000"], # Frontend URL
 allow_credentials=True,
 allow_methods=["*"],
 allow_headers=["*"],
)

Monitoring
setup_prometheus(app)

Routers
app.include_router(recommendations.router, prefix="/api/v1")
app.include_router(campaigns.router, prefix="/api/v1")
app.include_router(evaluations.router, prefix="/api/v1")

@app.get("/health")
async def health_check():
 """Health check endpoint"""

```

```
 return {"status": "healthy", "version": "0.1.0"}
```

```
src/api/routers/recommendations.py
from fastapi import APIRouter, HTTPException, BackgroundTasks
from pydantic import BaseModel
from typing import Optional
```

```
router = APIRouter(prefix="/recommendations", tags=["recommendations"])
```

```
class AnalyzeRequest(BaseModel):
 campaign_id: str
 force_refresh: bool = False # Skip cache
```

```
class AnalyzeResponse(BaseModel):
 recommendation_id: str
 recommendation: Recommendation
 status: str
```

```
class DecisionRequest(BaseModel):
 decision: str # 'approved' or 'rejected'
 feedback: Optional[str] = None
```

```
@router.post("/analyze", response_model=AnalyzeResponse)
@limiter.limit("10/minute") # Rate limit
async def analyze_campaign(
 request: AnalyzeRequest,
 background_tasks: BackgroundTasks,
 current_user = Depends(get_current_user),
 db = Depends(get_db)
):
 """
 Generate recommendation for a campaign

 Rate limited to 10 requests/minute per user
 """
 try:
 logger.info(
 "analyze_campaign_requested",
 campaign_id=request.campaign_id,
 user=current_user.email
)

 # Check if recent recommendation exists (unless force_refresh)
 if not request.force_refresh:
 recent_rec = await db.get_recent_recommendation(
 request.campaign_id,
 hours=24
)
 if recent_rec and recent_rec.human_decision is None:
 logger.info("returning_cached_recommendation")
 return AnalyzeResponse(
 recommendation_id=recent_rec.id,
 recommendation=recent_rec,
 status="cached"
)

```

```
Run agent workflow
```

```

agent = create_marketing_agent()
result = await agent.ainvoke({
 "campaign_id": request.campaign_id
})

Extract recommendation from workflow state
recommendation = result["recommendation"]

Store in database
rec_id = await db.store_recommendation(recommendation)

Send notification (async)
background_tasks.add_task(
 notify_marketing_team,
 recommendation_id=rec_id,
 campaign_id=request.campaign_id
)

Log for monitoring
recommendations_total.labels(
 workflow_type=recommendation.recommended_workflow,
 confidence_band=get_confidence_band(recommendation.confidence)
).inc()

return AnalyzeResponse(
 recommendation_id=rec_id,
 recommendation=recommendation,
 status="generated"
)

```

```

except Exception as e:
 logger.error(
 "analyze_campaign_failed",
 campaign_id=request.campaign_id,
 error=str(e)
)
 raise HTTPException(status_code=500, detail=str(e))

```

```

@router.get("/{recommendation_id}")
async def get_recommendation(
 recommendation_id: str,
 current_user = Depends(get_current_user),
 db = Depends(get_db)
):
 """Retrieve a specific recommendation"""
 rec = await db.get_recommendation(recommendation_id)

 if not rec:
 raise HTTPException(status_code=404, detail="Recommendation not found")

 return rec

```

```

@router.post("/{recommendation_id}/decision")
async def record_decision(
 recommendation_id: str,
 decision: DecisionRequest,
 background_tasks: BackgroundTasks,

```

```

 current_user = Depends(get_current_user),
 db = Depends(get_db)
):
 """
 Record human approval/rejection of recommendation
 """
 if decision.decision not in ['approved', 'rejected']:
 raise HTTPException(
 status_code=400,
 detail="Decision must be 'approved' or 'rejected'"
)

 # Update database
 await db.update_recommendation(
 recommendation_id=recommendation_id,
 human_decision=decision.decision,
 decided_by=current_user.email,
 decision_feedback=decision.feedback
)

 # Metrics
 recommendation_acceptance.labels(
 action='approved' if decision.decision == 'approved' else 'rejected'
).inc()

 # If approved, trigger workflow execution
 if decision.decision == 'approved':
 background_tasks.add_task(
 trigger_workflow_execution,
 recommendation_id=recommendation_id
)

 logger.info(
 "decision_recorded",
 recommendation_id=recommendation_id,
 decision=decision.decision,
 decided_by=current_user.email
)

 return {"status": "recorded"}

@router.get("/")
async def list_recommendations(
 status: Optional[str] = None, # 'pending', 'approved', 'rejected',
 campaign_id: Optional[str] = None,
 limit: int = 50,
 offset: int = 0,
 current_user = Depends(get_current_user),
 db = Depends(get_db)
):
 """
 List recommendations with filtering
 """
 recommendations = await db.list_recommendations(
 status=status,
 campaign_id=campaign_id,
 limit=limit,
 offset=offset
)

```

```

 return {
 "recommendations": recommendations,
 "limit": limit,
 "offset": offset
 }

```

## 8.3 Async Workflow Execution

```

src/workflows/executor.py
from celery import Celery
from typing import Dict, Any

Initialize Celery (for background tasks)
celery_app = Celery(
 'marketing_agent',
 broker='redis://localhost:6379/0',
 backend='redis://localhost:6379/0'
)

@celery_app.task
async def trigger_workflow_execution(recommendation_id: str):
 """
 Execute the recommended workflow on external systems

 This is where we integrate with existing marketing workflows
 """

 # Fetch recommendation
 rec = await db.get_recommendation(recommendation_id)

 workflow_handlers = {
 WorkflowType.BID_ADJUSTMENT: execute_bid_adjustment,
 WorkflowType.CREATIVE_REFRESH: execute_creative_refresh,
 WorkflowType.AUDIENCE_EXPANSION: execute_audience_expansion,
 WorkflowType.CAMPAIGN_PAUSE: execute_campaign_pause,
 WorkflowType.BUDGET_REALLOCATION: execute_budget_reallocation,
 }

 handler = workflow_handlers.get(rec.recommended_workflow)

 if not handler:
 logger.error(f"No handler for workflow: {rec.recommended_workflow}")
 return

 try:
 # Execute
 result = await handler(rec)

 # Log execution
 await db.log_workflow_execution(
 recommendation_id=recommendation_id,
 status='success',
)

```

```

 result=result
)

 # Schedule outcome tracking (check results in 3 days)
 schedule_outcome_tracking.apply_async(
 args=[recommendation_id],
 countdown=3 * 24 * 60 * 60 # 3 days
)

except Exception as e:
 logger.error(
 "workflow_execution_failed",
 recommendation_id=recommendation_id,
 error=str(e)
)
 await db.log_workflow_execution(
 recommendation_id=recommendation_id,
 status='error',
 error=str(e)
)
)

async def execute_bid_adjustment(rec: Recommendation) -> Dict[str, Any]:
 """Execute bid adjustment via ad platform API"""
 # Parse specific action (e.g., "Increase bid by 15% to $2.88")
 import re
 match = re.search(r'(\d+)%.*\$(\d+\.\d+)', rec.specific_action)

 if not match:
 raise ValueError(f"Could not parse bid adjustment: {rec.specific_action}")

 increase_pct = int(match.group(1))
 target_bid = float(match.group(2))

 # Call ad platform API
 platform_client = get_ad_platform_client(rec.campaign_id)
 result = await platform_client.update_bid(
 campaign_id=rec.campaign_id,
 new_bid=target_bid
)

 return {
 "action": "bid_adjustment",
 "previous_bid": result['previous_bid'],
 "new_bid": target_bid,
 "increase_pct": increase_pct,
 "executed_at": datetime.utcnow().isoformat()
 }

@celery_app.task
async def schedule_outcome_tracking(recommendation_id: str):
 """
 Check campaign performance after recommendation execution
 Track whether expected impact materialized
 """
 rec = await db.get_recommendation(recommendation_id)

 # Fetch current metrics

```

```

context_builder = ContextBuilder(...)
current_context = await context_builder.build_context(rec.campaign_id)

Compare to context at recommendation time
original_context = rec.context

Calculate changes
cpa_before = original_context['campaign_metrics']['cpa']
cpa_after = current_context['campaign_metrics']['cpa']
cpa_change_pct = ((cpa_after - cpa_before) / cpa_before) * 100

Determine success based on expected impact
success = evaluate_outcome_success(rec, cpa_change_pct)

Store outcome
await db.store_outcome(
 recommendation_id=recommendation_id,
 outcome_metrics={
 "cpa_before": cpa_before,
 "cpa_after": cpa_after,
 "cpa_change_pct": cpa_change_pct,
 "success": success,
 "improvement_pct": -cpa_change_pct if cpa_change_pct < 0 else 0
 }
)

Update evaluation dataset
await add_to_golden_dataset(recommendation_id)

logger.info(
 "outcome_tracked",
 recommendation_id=recommendation_id,
 success=success,
 cpa_change_pct=cpa_change_pct
)

```

## 9. Deployment Strategy

### 9.1 Environment Structure

**Three Environments:**

#### 1. Development (**dev**):

- Local Docker Compose
- Stub data collectors
- Mock LLM responses (for fast iteration)

- No rate limits

## 2. Staging (staging):

- Kubernetes cluster
- Real data collectors (sandbox APIs)
- Real LLM calls
- Human-in-the-loop testing
- Production-like config

## 3. Production (prod):

- Kubernetes cluster
- Real data, real LLMs
- High availability (2+ replicas)
- Monitoring and alerts
- Graduated rollout

## 9.2 Deployment Process

```
.github/workflows/deploy.yml
name: Deploy Marketing Agent

on:
 push:
 branches: [main]
 workflow_dispatch:
 inputs:
 environment:
 description: 'Environment to deploy to'
 required: true
 type: choice
 options:
 - staging
 - production

jobs:
 test:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Run unit tests
 run: pytest tests/unit

 - name: Run integration tests
 run: pytest tests/integration
```

```
evaluate:
 needs: test
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Run golden set evaluation
 env:
 OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
 run: python -m evaluation.run_golden_set

 - name: Check quality threshold
 run: |
 python -m evaluation.check_deployment_criteria

deploy-staging:
 needs: evaluate
 runs-on: ubuntu-latest
 environment: staging
 steps:
 - uses: actions/checkout@v3

 - name: Configure kubectl
 uses: azure/k8s-set-context@v1
 with:
 method: kubeconfig
 kubeconfig: ${{ secrets.KUBE_CONFIG_STAGING }}

 - name: Deploy to staging
 run: |
 kubectl apply -f k8s/staging/
 kubectl rollout status deployment/marketing-agent

 - name: Run smoke tests
 run: python -m tests.smoke_tests --env staging

deploy-production:
 needs: deploy-staging
 runs-on: ubuntu-latest
 environment: production
 if: github.event_name == 'workflow_dispatch' && github.event.inputs.environment
== 'production'
 steps:
 - uses: actions/checkout@v3

 - name: Configure kubectl
 uses: azure/k8s-set-context@v1
 with:
 method: kubeconfig
 kubeconfig: ${{ secrets.KUBE_CONFIG_PROD }}

 - name: Deploy with canary rollout
 run: |
 # Deploy canary (10% traffic)
 kubectl apply -f k8s/production/canary/
```

```

Wait and monitor
sleep 300 # 5 minutes

Check canary metrics
python -m deployment.check_canary_health

If healthy, proceed with full rollout
kubectl apply -f k8s/production/
kubectl rollout status deployment/marketing-agent

- name: Notify team
 uses: 8398a7/action-slack@v3
 with:
 status: ${{ job.status }}
 text: 'Marketing Agent deployed to production'
 webhook_url: ${{ secrets.SLACK_WEBHOOK }}

```

## 9.3 Kubernetes Configuration

```

k8s/production/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: marketing-agent
 namespace: production
spec:
 replicas: 3
 strategy:
 type: RollingUpdate
 rollingUpdate:
 maxSurge: 1
 maxUnavailable: 0
 selector:
 matchLabels:
 app: marketing-agent
 template:
 metadata:
 labels:
 app: marketing-agent
 version: v1
 spec:
 containers:
 - name: backend
 image: marketing-agent:latest
 ports:
 - containerPort: 8000
 env:
 - name: DATABASE_URL
 valueFrom:
 secretKeyRef:
 name: marketing-agent-secrets
 key: database-url

```

```

- name: OPENAI_API_KEY
 valueFrom:
 secretKeyRef:
 name: marketing-agent-secrets
 key: openai-api-key
resources:
 requests:
 memory: "512Mi"
 cpu: "500m"
 limits:
 memory: "2Gi"
 cpu: "2000m"
livenessProbe:
 httpGet:
 path: /health
 port: 8000
 initialDelaySeconds: 30
 periodSeconds: 10
readinessProbe:
 httpGet:
 path: /health
 port: 8000
 initialDelaySeconds: 5
 periodSeconds: 5

Sidecar: Prometheus exporter
- name: metrics-exporter
 image: prom/prometheus:latest
 ports:
 - containerPort: 9090

apiVersion: v1
kind: Service
metadata:
 name: marketing-agent-service
 namespace: production
spec:
 selector:
 app: marketing-agent
 ports:
 - protocol: TCP
 port: 80
 targetPort: 8000
 type: LoadBalancer

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
 name: marketing-agent-hpa
 namespace: production
spec:
 scaleTargetRef:
 apiVersion: apps/v1
 kind: Deployment
 name: marketing-agent

```

```

minReplicas: 3
maxReplicas: 10
metrics:
- type: Resource
 resource:
 name: cpu
 target:
 type: Utilization
 averageUtilization: 70
- type: Resource
 resource:
 name: memory
 target:
 type: Utilization
 averageUtilization: 80

```

## 9.4 Rollout Strategy

### Week 1-2: Shadow Mode

- Agent runs in background
- Generates recommendations
- No UI, no notifications
- Compare to what humans actually decide
- Goal: Baseline agreement rate

### Week 3-4: Internal Beta

- Show to 2-3 marketing team members
- Feedback collection focus
- Iteration on prompts and reasoning
- Goal: Refine to >60% acceptance

### Week 5-8: Limited Production

- Roll out to 25% of campaigns
- Full human approval required
- Weekly feedback sessions
- Goal: Achieve 70% acceptance rate

### Week 9+: Full Production

- All campaigns eligible
- Auto-notifications

- Graduated autonomy for high-confidence cases
  - Goal: Scale and optimize
- 

# 10. Monitoring & Observability

## 10.1 Metrics Dashboard

### Business Metrics:

```
src/monitoring/metrics.py
from prometheus_client import Counter, Histogram, Gauge

Recommendation tracking
recommendations_total = Counter(
 'recommendations_total',
 'Total recommendations generated',
 ['workflow_type', 'confidence_band']
)

recommendation_acceptance = Counter(
 'recommendation_acceptance_total',
 'Recommendations accepted by humans',
 ['workflow_type']
)

recommendation_latency = Histogram(
 'recommendation_latency_seconds',
 'Time to generate recommendation',
 buckets=[5, 10, 15, 20, 30, 45, 60]
)

Quality metrics
confidence_score_distribution = Histogram(
 'confidence_score',
 'Distribution of confidence scores',
 buckets=[0.0, 0.3, 0.5, 0.7, 0.8, 0.9, 1.0]
)

agreement_rate_daily = Gauge(
 'agreement_rate_daily',
 'Daily agreement rate with human decisions'
)

LLM metrics
llm_tokens_used = Counter(
 'llm_tokens_total',
 'Total LLM tokens consumed',
 ['model', 'operation']
)
```

```

llm_cost_usd = Counter(
 'llm_cost_usd_total',
 'Total LLM API costs in USD'
)

llm_api_errors = Counter(
 'llm_api_errors_total',
 'LLM API error count',
 ['model', 'error_type']
)

Workflow execution
workflow_execution_total = Counter(
 'workflow_execution_total',
 'Workflows executed in external systems',
 ['workflow_type', 'status']
)

workflow_execution_duration = Histogram(
 'workflow_execution_duration_seconds',
 'Time to execute workflow',
 ['workflow_type']
)

Outcome tracking
recommendation_impact = Gauge(
 'recommendation_impact_pct',
 'Average metric improvement when recommendation followed',
 ['workflow_type']
)

```

## Grafana Dashboard (JSON):

```
{
 "title": "Marketing Agent - Overview",
 "panels": [
 {
 "title": "Recommendations per Day",
 "targets": [
 {
 "expr": "rate(recommendations_total[1d])"
 }
]
 },
 {
 "title": "Acceptance Rate",
 "targets": [
 {
 "expr": "rate(recommendation_acceptance_total[1d]) / rate(recommendations_total[1d]) * 100"
 }
],
 "alert": {
 "threshold": 50
 }
 }
]
}
```

```

"conditions": [
 {
 "evaluator": {
 "params": [70],
 "type": "lt"
 },
 "query": {
 "expr": "acceptance_rate"
 }
 }
],
"message": "Acceptance rate dropped below 70%"
}
},
{
 "title": "p95 Latency",
 "targets": [
 {
 "expr": "histogram_quantile(0.95,
rate(recommendation_latency_seconds_bucket[5m]))"
 }
]
},
{
 "title": "LLM Cost per Day",
 "targets": [
 {
 "expr": "increase(llm_cost_usd_total[1d])"
 }
]
}
]
}

```

## 10.2 LangSmith Integration

```

src/agent/workflow.py
import os
from langsmith import Client
from langsmith.run_helpers import traceable

Initialize LangSmith
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = "marketing-agent-prod"

@traceable(run_type="chain", name="marketing_agent")
async def run_marketing_agent(campaign_id: str):
 """
 Traced agent execution - all LLM calls, latencies,
 and intermediate states visible in LangSmith
 """
 workflow = create_marketing_agent()

```

```
result = await workflow.ainvoke({"campaign_id": campaign_id})
return result
```

## LangSmith Benefits:

- Trace every LLM call with inputs/outputs
- Visualize workflow execution paths
- Debug failures with full context
- Compare prompt versions
- Production query: "Find all recommendations with confidence < 0.6"

## 10.3 Alerting Rules

```
monitoring/alerting_rules.yml
groups:
 - name: marketing_agent_alerts
 interval: 1m
 rules:
 # Quality alerts
 - alert: AcceptanceRateLow
 expr: acceptance_rate_daily < 0.70
 for: 1h
 labels:
 severity: warning
 annotations:
 summary: "Acceptance rate dropped below 70%"
 description: "Current rate: {{ $value }}%"

 - alert: HighErrorRate
 expr: rate(llm_api_errors_total[5m]) > 0.1
 for: 5m
 labels:
 severity: critical
 annotations:
 summary: "LLM API error rate elevated"

 # Performance alerts
 - alert: HighLatency
 expr: histogram_quantile(0.95,
rate(recommendation_latency_seconds_bucket[5m])) > 30
 for: 10m
 labels:
 severity: warning
 annotations:
 summary: "p95 latency > 30 seconds"

 # Cost alerts
 - alert: DailyCostExceeded
 expr: increase(llm_cost_usd_total[1d]) > 100
```

```

labels:
 severity: warning
annotations:
 summary: "Daily LLM costs exceeded $100"

Business alerts
- alert: NoRecommendations
 expr: rate(recommendations_total[1h]) == 0
 for: 2h
 labels:
 severity: critical
 annotations:
 summary: "No recommendations generated in 2 hours"

```

## 10.4 Log Aggregation

```

src/monitoring/logging.py
import structlog
import logging
from pythonjsonlogger import jsonlogger

def setup_logging():
 """Configure structured logging"""

 # JSON formatter for machine parsing
 logHandler = logging.StreamHandler()
 formatter = jsonlogger.JsonFormatter(
 '%(timestamp)s %(level)s %(name)s %(message)s'
)
 logHandler.setFormatter(formatter)

 # Configure structlog
 structlog.configure(
 processors=[
 structlog.stdlib.filter_by_level,
 structlog.stdlib.add_logger_name,
 structlog.stdlib.add_log_level,
 structlog.stdlib.PositionalArgumentsFormatter(),
 structlog.processors.TimeStamper(fmt="iso"),
 structlog.processors.StackInfoRenderer(),
 structlog.processors.format_exc_info,
 structlog.processors.UnicodeDecoder(),
 structlog.processors.JSONRenderer()
],
 context_class=dict,
 logger_factory=structlog.stdlib.LoggerFactory(),
 cache_logger_on_first_use=True,
)

 logging.basicConfig(
 format="%(message)s",
 stream=sys.stdout,

```

```

 level=logging.INFO,
 handlers=[logHandler]
)

Usage
logger = structlog.get_logger()

logger.info(
 "recommendation_generated",
 campaign_id="camp_123",
 workflow="Bid Adjustment",
 confidence=0.85,
 user_id="user_456"
)

```

## Logs Example:

```
{
 "timestamp": "2026-02-11T10:30:45.123Z",
 "level": "info",
 "name": "marketing_agent",
 "event": "recommendation_generated",
 "campaign_id": "camp_123",
 "workflow": "Bid Adjustment",
 "confidence": 0.85,
 "user_id": "user_456",
 "duration_ms": 12450,
 "llm_tokens": 3200
}
```

# 11. Cost Management

## 11.1 Cost Breakdown Projection

### Assumptions:

- 100 campaigns monitored
- 1 recommendation per campaign per week
- Average context: 5,000 input tokens
- Average output: 1,500 tokens

### Monthly Costs (OpenAI GPT-4o):

Component	Usage	Cost per Unit	Monthly Cost
<b>Context Collection</b>	400 requests × 5k tokens	\$0.15/1M token	\$3.00
<b>Signal Analysis</b>	400 requests × 7k tokens	\$0.15/1M token	\$4.20
<b>Recommendation Gen</b>	400 requests × 1.5k tokens	\$0.60/1M token	\$3.60
<b>Critique Loop</b>	200 requests × 3k tokens	\$0.15/1M token	\$0.90
<b>LLM-as-Judge Eval</b>	100 requests × 5k tokens	\$0.15/1M token	\$0.75
<b>Infrastructure</b>	AWS/Kubernetes	-	\$200
<b>Total</b>			<b>~\$212/month</b>

## 11.2 Cost Optimization Strategies

```
src/agent/cost_optimizer.py

class CostOptimizer:
 """Optimize LLM usage to reduce costs"""

 def __init__(self):
 self.cache = RedisCache(ttl=3600)

 async def get_cached_analysis(
 self,
 context_hash: str
) -> Optional[dict]:
 """
 Cache analysis results for similar contexts

 If two campaigns have nearly identical metrics,
 reuse analysis with minor adjustments
 """
 return await self.cache.get(f"analysis:{context_hash}")

 def select_model_by_complexity(
 self,
 context: CampaignContext
) -> str:
 """
```

## Use cheaper models for simple cases

```
- GPT-4o: Complex reasoning, edge cases
- GPT-4o-mini: Straightforward cases
"""
Calculate complexity score
complexity = 0

Many metrics changed significantly
if context['campaign_metrics']['cpa_change_pct'] > 50:
 complexity += 2

Conflicting signals
if has_conflicting_signals(context):
 complexity += 3

Novel scenario
if not has_similar_historical_case(context):
 complexity += 2

Select model
if complexity >= 5:
 return "gpt-4o" # $0.15/$0.60 per 1M tokens
else:
 return "gpt-4o-mini" # $0.03/$0.12 per 1M tokens

def compress_context(
 self,
 context: CampaignContext
) -> dict:
"""
Reduce token count by removing redundant info

- Round numbers to 2 decimals
- Remove null/empty fields
- Summarize historical data
"""
compressed = {}

Only include metrics with significant changes
metrics = context['campaign_metrics']
compressed['metrics'] = {
 k: round(v, 2) if isinstance(v, float) else v
 for k, v in metrics.items()
 if k.endswith('_change_pct') and abs(v) > 5 # >5% change
}

Summarize instead of full detail
if len(context.get('historical_patterns', [])) > 5:
 compressed['historical_summary'] = summarize_patterns(
 context['historical_patterns']
)

return compressed

Apply optimization
optimizer = CostOptimizer()
```

```

model = optimizer.select_model_by_complexity(context)
cached_result = await optimizer.get_cached_analysis(context_hash)

if cached_result:
 logger.info("Using cached analysis", savings_usd=0.05)
 return cached_result

```

## 11.3 Cost Tracking

```

src/monitoring/cost_tracker.py
from dataclasses import dataclass

@dataclass
class LLMUsage:
 model: str
 input_tokens: int
 output_tokens: int
 timestamp: datetime

class CostTracker:
 """Track and report LLM costs"""

 PRICING = {
 "gpt-4o": {"input": 0.15 / 1_000_000, "output": 0.60 / 1_000_000},
 "gpt-4o-mini": {"input": 0.03 / 1_000_000, "output": 0.12 / 1_000_000},
 "claude-3-5-sonnet": {"input": 0.15 / 1_000_000, "output": 0.60 / 1_000_000},
 }

 def calculate_cost(self, usage: LLMUsage) -> float:
 """Calculate cost for a single LLM call"""
 pricing = self.PRICING.get(usage.model, self.PRICING["gpt-4o"])

 input_cost = usage.input_tokens * pricing["input"]
 output_cost = usage.output_tokens * pricing["output"]

 return input_cost + output_cost

 async def generate_cost_report(
 self,
 start_date: datetime,
 end_date: datetime
) -> dict:
 """Generate cost report for period"""
 usages = await db.get_llm_usages(start_date, end_date)

 total_cost = sum(self.calculate_cost(u) for u in usages)

 # Breakdown by operation
 by_operation = {}
 for usage in usages:

```

```

 op = usage.operation
 if op not in by_operation:
 by_operation[op] = {"cost": 0, "calls": 0, "tokens": 0}

 by_operation[op]["cost"] += self.calculate_cost(usage)
 by_operation[op]["calls"] += 1
 by_operation[op]["tokens"] += usage.input_tokens + usage.output_tokens

 return {
 "period": {"start": start_date, "end": end_date},
 "total_cost": total_cost,
 "total_calls": len(usages),
 "by_operation": by_operation,
 "avg_cost_per_call": total_cost / len(usages) if usages else 0
 }

Usage in monitoring
cost_tracker = CostTracker()

After each LLM call
llm_cost_usd.inc(cost_tracker.calculate_cost(usage))

```

## 12. Team Structure & Workflows

### 12.1 Recommended Team

#### Phase 1-3 (Months 1-4): Foundation

- **1 GenAI Engineer (You) - Full-time**
  - Agent development
  - Prompt engineering
  - Evaluation framework
  - Infrastructure setup
- **0.5 Backend Engineer (Part-time support)**
  - API development
  - Data integration
  - Database design
- **0.25 Marketing Subject Matter Expert (Consultation)**
  - Requirements validation

- Test case creation
- Feedback on recommendations

## Phase 4-6 (Months 5-8): Production Launch

- **1 GenAI Engineer** - Full-time
- **1 Backend Engineer** - Full-time (API, integrations, scale)
- **0.5 Frontend Engineer** - Build review UI
- **0.5 DevOps Engineer** - Kubernetes, monitoring, CI/CD
- **Marketing Team** - Weekly feedback sessions

## Phase 7+ (Months 9+): Optimization & Scale

- **1 GenAI Engineer** - Improve reasoning, add features
- **0.5 Backend Engineer** - Maintenance, new integrations
- **Marketing Team** - Ongoing feedback loop

# 12.2 Weekly Workflow

## Monday:

- Review weekend metrics (acceptance rate, outcomes)
- Triage any critical issues
- Plan week's improvements

## Tuesday-Thursday:

- Development work
- Prompt iteration based on feedback
- Evaluation runs on every PR

## Friday:

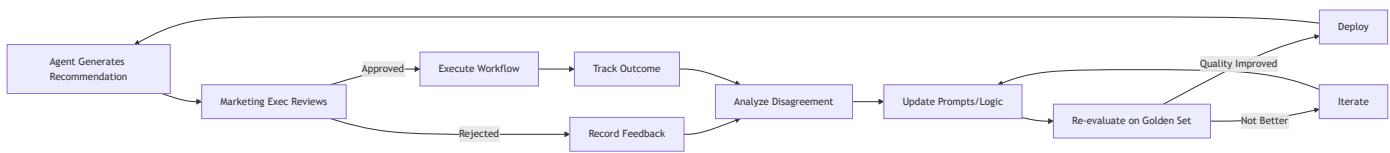
- Marketing team feedback session (30-60 min)
- Review flagged recommendations
- Discuss failure cases
- Identify patterns for improvement

## Continuous:

- Monitor dashboards

- Respond to alerts
- Document learnings

## 12.3 Feedback Loop Process



### Feedback Categories:

- 1. Correct Decision, Good Reasoning** - Reinforcement learning
- 2. Correct Decision, Poor Reasoning** - Improve explanation clarity
- 3. Wrong Decision, Understandable Why** - Edge case, add to training
- 4. Wrong Decision, Unclear Why** - Investigate failure mode

## 12.4 Cross-Team Collaboration: Data Science & MLOps

**Context:** The data science team is building MLOps infrastructure for traditional ML models (predictive models, recommendation systems, etc.). As you establish GenAI patterns, there's significant overlap in operational concerns but also critical differences.

### Areas of Alignment (Leverage Shared Infrastructure):

Component	Traditional ML	GenAI	Shared Approach
Monitoring	Prometheus, Grafana	Same + LangSmith	Share Prometheus/Grafana infrastructure, add GenAI-specific dashboards
CI/CD	GitHub Actions, testing	Same + LLM evaluation	Extend existing pipelines with evaluation gates
Feature Stores	Feast, Tecton	Context stores	Potentially share feature store for numerical features (CPA, CTR)

Component	Traditional ML	GenAI	Shared Approach
Experiment Tracking	MLflow	LangSmith + MLflow	Use MLflow for cost tracking, LangSmith for traces
Data Quality	Great Expectations	Same + prompt validation	Share data quality checks for campaign metrics
Infrastructure	Kubernetes, Docker	Same	Share K8s clusters with proper resource isolation

### Areas of Divergence (GenAI-Specific Patterns):

Aspect	Traditional ML	GenAI	Why Different
Evaluation	Numeric metrics (RMSE, AUC)	LLM-as-judge, human feedback	GenAI outputs are natural language, require semantic evaluation
Deployment	Model versioning, A/B testing	Prompt versioning, structured outputs	Prompts change more frequently than trained models
Latency	<100ms inference	~10-30s LLM calls	LLMs are slower; need async patterns
Cost Model	Fixed compute cost	Token-based pricing	GenAI has per-call variable costs
Observability	Model predictions, features	LLM traces, prompt-response pairs	Need to trace multi-step reasoning chains
Feedback Loop	Labeled data, retraining	Human approval, prompt iteration	GenAI improves via prompt engineering, not retraining

### Collaboration Protocol:

#### Monthly Sync (1 hour):

- Share learnings on monitoring patterns

- Discuss infrastructure evolution (K8s, observability)
- Identify opportunities for shared tooling
- Review cost optimization strategies

## Shared Documentation:

- Maintain a **PLATFORM\_HANDBOOK.md** covering:
  - Shared infrastructure (Kubernetes, Prometheus)
  - Deployment patterns
  - Incident response procedures
  - Cost management guidelines

## Divergent Documentation:

- Your team: **GenAI\_PATTERNS.md** covering:
  - Prompt engineering workflows
  - LLM evaluation methodologies
  - Human-in-the-loop patterns
  - Token cost optimization
- Their team: **ML\_PATTERNS.md** covering:
  - Model training pipelines
  - Feature engineering
  - A/B testing frameworks

## Practical Integration Points:

### 1. Shared Prometheus Metrics:

```
Both ML and GenAI use same metric format
from prometheus_client import Counter, Histogram

prediction_requests = Counter(
 'prediction_requests_total',
 'Total prediction requests',
 ['service', 'model_type'] # model_type: 'ml' or 'genai'
)
```

### 2. Shared Grafana Dashboards:

- Top-level: System health (CPU, memory, requests/sec)
- ML-specific: Model latency, prediction drift
- GenAI-specific: LLM latency, token usage, recommendation acceptance

### 3. Shared GitHub Actions Base:

```
.github/workflows/base-checks.yml (shared)
- name: Run Security Scan
- name: Check Code Quality
- name: Build Docker Image

.github/workflows/ml-pipeline.yml (ML team)
- name: Run Model Tests
- name: Evaluate on Test Set

.github/workflows/genai-pipeline.yml (Your team)
- name: Run LLM Evaluation
- name: Test Golden Dataset
```

### 4. Example Collaboration: Campaign Metrics

```
Shared: Data science team provides campaign features
src/shared/features.py (maintained by DS team)
class CampaignFeatures:
 def get_cpa_trend(self, campaign_id: str) -> float:
 """Predictive model for CPA trend"""
 # ML model trained by data science team

 # GenAI Usage: You consume these features
 # src/agent/marketing_agent.py
 features = CampaignFeatures()
 predicted_cpa = features.get_cpa_trend(campaign_id)

 # Use in agent prompt context
 context = f"Predicted CPA trend: {predicted_cpa:.2f}"
```

## Decision Framework: When to Converge vs. Diverge

### Use Shared Infrastructure When:

- It's a commodity (Kubernetes, Prometheus, Docker)
- No GenAI-specific requirements
- Reduces operational overhead
- Enables cross-team knowledge sharing

### Build Separate When:

- GenAI has unique requirements (LLM tracing, prompt versioning)
- Iteration speed matters (don't block on other team)
- Cost/complexity of integration > cost of duplication

**Key Principle:** "Share infrastructure, diverge on workflows."

## 12.5 Communication Channels

### Internal:

- **Slack Channel #marketing-agent** - Daily updates, questions
- **Slack Channel #ml-platform** - Cross-team collaboration with data science
- **Weekly Sync** - 30 min with marketing team
- **Monthly MLOps Sync** - 1 hour with data science team
- **Monthly Review** - Stakeholder presentation on progress

### Documentation:

- **README** - Setup instructions
- **Architecture Decision Records (ADRs)** - Key design choices
- **Prompt Changelog** - Version history with performance impact
- **Runbook** - Incident response procedures
- **PLATFORM\_HANDBOOK.md** - Shared infrastructure guide (co-maintained with DS team)

### External (Optional):

- **Blog Posts** - Share learnings (with data anonymized)
- **Conference Talks** - GenAI in production

---

## Appendices

---

### Appendix A: Glossary

**Agent:** AI system that can reason and take actions autonomously **Workflow:**

Predefined sequence of operations (less flexible than agent) **LangGraph:** Framework

for building stateful, multi-step AI workflows **LangSmith:** Observability platform for LLM

applications **Golden Dataset:** Curated test cases with known correct answers **LLM-as-**

**Judge:** Using an LLM to evaluate another LLM's outputs **Prompt Engineering:**

Crafting instructions for LLMs to perform tasks **Human-in-the-Loop (HITL):** System

that requires human approval/input **Graduated Autonomy:** Slowly reducing human

oversight as trust builds **Confidence Calibration**: Ensuring confidence scores match actual accuracy

## Appendix B: Risk Mitigation

Risk	Likelihood	Impact	Mitigation
<b>Poor recommendation quality</b>	Medium	High	<ul style="list-style-type: none"><li>- Extensive evaluation before launch</li><li>- Human approval for all recs initially</li><li>- Clear acceptance threshold (70%)</li></ul>
<b>LLM API downtime</b>	Low	Medium	<ul style="list-style-type: none"><li>- Fallback to rule-based system</li><li>- Retry logic with exponential backoff</li><li>- Circuit breaker pattern</li></ul>
<b>Cost overruns</b>	Medium	Low	<ul style="list-style-type: none"><li>- Budget alerts (\$100/day limit)</li><li>- Model selection by complexity</li><li>- Response caching</li></ul>
<b>Data quality issues</b>	Medium	Medium	<ul style="list-style-type: none"><li>- Validation on all collector outputs</li><li>- Graceful degradation with partial data</li><li>- Health checks on collectors</li></ul>
<b>Marketing team rejection</b>	Low	High	<ul style="list-style-type: none"><li>- Early and continuous involvement</li><li>- Weekly feedback sessions</li><li>- Iterate based on their input</li></ul>
<b>Prompt injection/manipulation</b>	Low	Low	<ul style="list-style-type: none"><li>- Input sanitization</li><li>- Output validation</li><li>- Audit logging</li></ul>

# Appendix C: Success Metrics Checklist

## Month 1-2: Foundation

- Dev environment working
- Data collectors functional
- First recommendation generated
- 5 golden test cases created

## Month 3-4: MVP

- Agent matches human decision >50% on golden set
- <30s latency consistently
- Reasoning is explainable
- Evaluation runs automatically

## Month 5-6: Launch

- UI deployed
- Marketing team trained
- 20+ real recommendations generated
- Acceptance rate >60%
- Monitoring dashboards live

## Month 7-8: Optimization

- Acceptance rate >70%
- Positive impact >80% when followed
- Cost <\$500/month
- Weekly iterations deployed

## Month 9+: Trust Building

- First graduated autonomy scenario identified
- 100+ recommendations with outcomes tracked
- Agreement with experts >75%
- Plan for next GenAI application

# Appendix D: Reference Architecture Diagram

[See full Mermaid diagram above]

## Appendix E: Prompt Templates Repository Structure

```
prompts/
 └── signal_analysis/
 ├── v1.yaml # Initial version
 ├── v2.yaml # Improved based on feedback
 ├── v3.yaml # Added few-shot examples
 └── latest.yaml -> v3.yaml
 └── recommendation_generation/
 ├── v1.yaml
 └── latest.yaml -> v1.yaml
 └── critique/
 ├── v1.yaml
 └── latest.yaml -> v1.yaml
└── README.md # Changelog and versioning guide
```

## Appendix F: Further Reading

### AI Agent Patterns:

- [Building Effective Agents - Anthropic](#)
- [LangGraph Documentation](#)

### Evaluation:

- [LangSmith Evaluation Guide](#)
- [promptfoo Documentation](#)

### Production GenAI:

- [GenAI Engineering Best Practices - OpenAI](#)
- [Prompt Engineering Guide](#)

### MLOps:

- [Building LLM Applications for Production - Chip Huyen](#)

# 13. Future Roadmap & Next Applications

---

## 13.1 Post-Launch Strategy

**Context:** Once the marketing agent is deployed and stable, you'll identify the next high-value GenAI application together with the business. We're not starting with a five-year roadmap because the best next problem will emerge from how the business actually uses the agent.

### Guiding Principles:

#### 1. Learn from Production:

- What works: Which reasoning patterns are most reliable?
- What's missing: Where does the agent still need human intervention?
- Where GenAI creates leverage: Which tasks benefit most from AI assistance?

#### 2. Separate Signal from Noise:

- Not every problem needs GenAI
- Focus on problems with:
  - High context complexity (many data sources to synthesize)
  - Reasoning requirements (not just pattern matching)
  - Contained blast radius (failures are recoverable)
  - Human verification possible (can validate outputs)
  - Clear evaluation criteria (can measure quality)

#### 3. Build on Foundations:

- Reuse evaluation frameworks
- Extend monitoring infrastructure
- Leverage learned patterns (prompt engineering, human-in-the-loop)

## 13.2 Candidate Next Applications

**These are speculation** - the actual choice depends on production learnings.

Examples:

## 1. Creative Performance Predictor

- **Problem:** Marketing team creates multiple ad variants, unsure which will perform
- **GenAI Solution:** Analyze creative elements (copy tone, visuals, CTA) + historical performance → predict which variants will succeed
- **Why Later:** Builds on marketing agent's campaign knowledge
- **Evaluation:** Predicted vs. actual performance correlation

## 2. Competitive Intelligence Summarizer

- **Problem:** Manual monitoring of competitor campaigns across platforms
- **GenAI Solution:** Aggregate competitor data → generate weekly strategic summary with actionable insights
- **Why Later:** Extends competitor analysis from marketing agent
- **Evaluation:** Marketing team finds insights actionable

## 3. Campaign Briefing Assistant

- **Problem:** Creating campaign briefs requires synthesizing brand guidelines, past performance, market insights
- **GenAI Solution:** Generate first-draft campaign brief from minimal input
- **Why Later:** Upstream of campaign execution (where agent currently operates)
- **Evaluation:** Draft quality, time saved

## 4. Multi-Channel Attribution Explainer

- **Problem:** Attribution models are black boxes to marketing team
- **GenAI Solution:** Explain attribution decisions in plain language with supporting evidence
- **Why Later:** Builds trust in data-driven decisions
- **Evaluation:** Understanding improvement, decision confidence

# 13.3 Discovery Process

## Months 6-9: Production Stabilization + Discovery

### Weekly Discovery Questions:

- Where is the marketing team still spending significant manual time?
- What decisions are they making that could benefit from AI reasoning?
- What feedback are they giving about the agent's gaps?

- What adjacent problems are they asking about?

## Monthly Review:

- Track time savings from current agent
- Identify 2-3 candidate next problems
- Rough scope each (complexity, value, fit for GenAI)

## Quarter-End Decision:

- Select next application based on:
  - Business value (time saved, revenue impact)
  - Technical feasibility (data available, evaluation possible)
  - Strategic fit (builds on existing capabilities)
  - Team readiness (lessons from first agent applied)

## 13.4 Scaling GenAI Capabilities

### As Additional Applications are Built:

#### Shared Components:

```

genai-platform/
├── core/
│ ├── llm_clients/ # Reusable LLM wrappers
│ ├── evaluation/ # Shared eval framework
│ ├── monitoring/ # Common observability
│ └── cost_tracking/ # Unified cost management
├── applications/
│ ├── marketing_agent/ # First application (this project)
│ ├── creative_predictor/ # Future application #2
│ └── competitive_intel/ # Future application #3
└── infrastructure/
 ├── kubernetes/ # Shared deployment configs
 ├── ci_cd/ # Reusable pipelines
 └── monitoring/ # Grafana dashboards

```

### Platform Team Evolution:

- **Months 1-6:** You own everything for marketing agent
- **Months 7-12:** Extract shared components as you build application #2
- **Year 2+:** Potentially grow to 2-3 person GenAI team owning platform + applications

# 13.5 Success Criteria for Platform Maturity

## Phase 1: Single Application (Months 1-6)

- Marketing agent in production
- Evaluation framework established
- Monitoring and CI/CD patterns proven

## Phase 2: Platform Emergence (Months 7-12)

- Second application using shared components
- Reusable evaluation framework
- Documented GenAI patterns
- Cost per application decreasing (leverage shared infra)

## Phase 3: Scaled Platform (Year 2+)

- 3+ applications in production
- Self-service GenAI toolkit for other teams
- Proven ROI on GenAI investment
- Clear decision framework: when to use GenAI vs. traditional ML vs. rules

# 13.6 What We're NOT Doing

## Avoiding Common Pitfalls:

### ✗ Building GenAI for Everything

- Not every problem needs an LLM
- Traditional ML, rules engines, and human processes still have their place

### ✗ Five-Year Roadmap Upfront

- Technology evolves too quickly
- Business needs change based on actual usage
- Better to earn the roadmap through shipping

### ✗ Autonomous Agents Without Guardrails

- Marketing agent has human approval for good reason
- Future applications should have similar safety mechanisms

- Trust is earned gradually, not assumed

## ✖ Optimizing Prematurely

- First agent focuses on quality, not cost
- Optimization happens once patterns are proven
- Don't sacrifice reliability for marginal cost savings

## 13.7 Long-Term Vision

### What Success Looks Like in 2 Years:

1. **Marketing Agent:** Fully automated with 80%+ acceptance rate, minimal human oversight
2. **GenAI Platform:** Reusable components enabling rapid new application development
3. **Team Capability:** 2-3 GenAI engineers owning multiple production applications
4. **Business Impact:** Measurable ROI through time savings, decision quality, efficiency gains
5. **Cultural Shift:** Marketing team trusts AI recommendations, uses GenAI as a thought partner

**Commitment:** We're establishing GenAI capabilities properly - the specific roadmap will be earned through shipping and learning. Focus is on systematic progress, quality, and building trust.

---

## Conclusion

---

This implementation guide provides a complete blueprint for building a production-grade Marketing Reasoning Agent. Key takeaways:

1. **Start Simple:** Workflow-based architecture, not fully autonomous agent
2. **Evaluation-Driven:** Build golden dataset, measure systematically
3. **Human Trust:** Human-in-the-loop initially, graduate to autonomy
4. **Production-Grade:** Monitoring, CI/CD, and evaluation from day one
5. **Iterative:** Weekly improvements based on real feedback
6. **Platform Thinking:** Build for today's needs, design for tomorrow's scale

The first 3-6 months focus on establishing quality foundations. Production deployment happens when the agent proves reliable through systematic evaluation, not because of arbitrary deadlines.

Success is measured by:

- Recommendation acceptance rate >70%
- Positive impact when followed >80%
- Marketing team trust and adoption
- Systematic, measurable progress
- Foundations for future GenAI applications

This is a marathon, not a sprint. Build trust through consistent, reliable recommendations. The roadmap will evolve based on what actually works in production - signal over noise, quality over speed.

### **Immediate Next Steps:**

1. Review this document with stakeholders
2. Set up development environment (Week 1)
3. Start data integration (Week 2-3)
4. First recommendation generation (Week 4)
5. Weekly marketing team sync (ongoing)

### **Long-Term Vision:**

- Marketing agent deployed and stable
- Discover next high-value application from production learnings
- Build GenAI platform capabilities through systematic iteration
- Establish patterns that work together with traditional ML (data science team)
- Create genuine business leverage through intelligent automation

Good luck building! 

---

**Document Version:** 1.0

**Last Updated:** February 11, 2026

**Maintained By:** GenAI Engineering Team

**Next Review:** March 2026