Unilever POC - Implementation Plan

C. Metrics to Track

Unilever Procurement GPT POC - Complete Implementation Plan

Executive Summary

Project: Build AI agents (Spend & Demand) with evaluation and monitoring framework **Timeline:** 8-10 weeks **Team Required:** 2-3 developers **Deliverables:** 1. Spend Agent (Text-to-SQL) 2. Demand Agent (Text-to-SQL) 3. Evaluation Framework (≥90% accuracy) 4. Monitoring Framework (Drift + Error Classification) 5. Dashboard & APIs

---

Phase 1: Setup & Data Discovery (Week 1)

1.1 Environment Setup

Duration: 2 days

Tasks: - [ ] Set up development environment - [ ] Install Ollama + Llama 3.1 8B - [ ] Set up PostgreSQL 15+ with pgvector - [ ] Create project structure - [ ] Set up Git repository

Deliverables:

```
Project initialized with:
- Python 3.11 virtual environment
- All dependencies installed
- Database running locally
- Version control configured
```

1.2 Data Access & Schema Discovery

Duration: 3 days

Tasks: - [ ] Obtain database credentials from Unilever - [ ] Connect to Spend database - [ ] Connect to Demand database - [ ] Document complete schema (tables, columns, relationships) - [ ] Extract sample data (anonymized if needed) - [ ] Understand business logic and constraints

Deliverables:

```
Documentation:
- spend_schema.md (complete ERD)
- demand_schema.md (complete ERD)
- business_rules.md (constraints, calculations)
- Sample data: 1000 rows per key table
```

Critical Questions to Ask Unilever: 1. Database connection strings and credentials? 2. Are there views or only raw tables? 3. Any stored procedures or functions we should know? 4. Data refresh frequency? 5. Any PII or sensitive data handling requirements?

Phase 2: Build AI Agents (Week 2-3)

2.1 Spend Agent Development

Duration: 5 days

Day 1-2: Basic Text-to-SQL Pipeline

Tasks: - [ ] Create base agent class with core pipeline - [ ] Implement query understanding module - [ ] Implement SQL generation with LLM - [ ] Implement SQL validation - [ ] Implement query execution - [ ] Add error handling

Code Structure:

```
class SpendAgent:
    def __init__(self, db_connection, schema):
        self.db = db_connection
        self.schema = schema
        self.llm = Ollama("llama3.1")
        self.validator = SQLValidator(schema)

    def process_query(self, user_query):
        # 1. Parse query intent
        # 2. Retrieve relevant context
        # 3. Generate SQL
        # 4. Validate SQL
        # 5. Execute SQL
        # 6. Format response
        pass
```

Testing: - Test with 10 simple queries (SELECT only) - Verify SQL syntax correctness - Check database execution success

Day 3-4: Add Context Retrieval (RAG)

Tasks: - [ ] Collect 50 example queries from Unilever - [ ] Create example query database - [ ] Implement embedding-based retrieval - [ ] Add few-shot prompting - [ ] Test with complex queries (JOINs, aggregations)

Implementation:

```
class ContextRetriever:
    def __init__(self):
        self.embeddings = SentenceTransformer('all-MiniLM-L6-v2')
        self.examples = self.load_examples()
        self.vector_store = FAISS.from_documents(self.examples)

    def get_relevant_examples(self, query, k=3):
        # Retrieve top-k similar examples
        pass
```

Testing: - Test with 20 medium-complexity queries - Measure SQL accuracy manually - Collect failed queries for improvement

Day 5: Add Self-Correction & Retry Logic

Tasks: - [ ] Implement error detection - [ ] Add SQL correction using LLM feedback - [ ] Add retry mechanism (max 3 attempts) - [ ] Log all attempts for analysis

Testing: - Deliberately create failing queries - Verify retry mechanism works - Check correction success rate

Deliverable: Functional Spend Agent with 70-80% success rate

2.2 Demand Agent Development

Duration: 5 days

Tasks: - [ ] Clone Spend Agent architecture - [ ] Customize for Demand schema - [ ] Collect Demand-specific examples - [ ] Add Demand-specific business logic - [ ] Test with 30 Demand queries

Note: Reuse most code from Spend Agent, customize prompts and examples

Deliverable: Functional Demand Agent with 70-80% success rate

---

Phase 3: Create Ground Truth Dataset (Week 3-4)

3.1 Query Collection

Duration: 2 days

Tasks: - [ ] Work with Unilever to collect real user queries - [ ] Categorize queries by complexity (simple, medium, hard) - [ ] Balance across different query types: - Aggregations (SUM, AVG, COUNT) - Filters (WHERE, HAVING) - Joins (INNER, LEFT, multiple tables) - Time-based queries (date ranges) - Top N queries (LIMIT, ORDER BY)

Target Distribution:

```
Total: 2,500 queries
- Simple (40%): 1,000 queries - Single table, basic filters
- Medium (40%): 1,000 queries - 2-3 table joins, aggregations
- Complex (20%): 500 queries - Multiple joins, subqueries, CTEs
```

3.2 Ground Truth Generation

Duration: 5 days

Approach A: Manual Creation (Recommended for accuracy) - Work with Unilever domain experts - Write expected SQL for each query - Execute SQL and store results - Manual verification

Approach B: Semi-Automated - Use agents to generate initial SQL - Expert review and correction - Execute and verify results - Store as ground truth

Tasks: - [ ] Create ground truth for 1,000 training queries - [ ] Create ground truth for 1,000 test queries - [ ] Create ground truth for 500 validation queries - [ ] Store in structured format (JSON/CSV)

Deliverable:

```
data/ground_truth/
├── train_1000.json
├── test_1000.json
└── validation_500.json

Format:
{
  "query_id": "Q001",
  "query_text": "What was Q1 2024 spend?",
  "expected_sql": "SELECT SUM(amount)...",
  "expected_result": {"total": 5200000},
  "agent_type": "spend",
  "complexity": "simple"
}
```

Phase 4: Build Evaluation Framework (Week 4-5)

4.1 Evaluation Pipeline

Duration: 3 days

Tasks: - [ ] Implement pre-processing module - [ ] Implement structural validation (SQL syntax, schema) - [ ] Implement semantic validation (compare with ground truth) - [ ] Implement LLM-as-Judge evaluation - [ ] Implement scoring algorithm - [ ] Create evaluation database schema

Code Structure:

```
class EvaluationFramework:
    def __init__(self):
        self.structural_validator = StructuralValidator()
        self.semantic_validator = SemanticValidator()
        self.llm_judge = LLMJudge()
        self.db = EvaluationDB()

    def evaluate(self, query, agent_response, ground_truth):
        # Step 1: Pre-process
        cleaned = self.preprocess(agent_response)

        # Step 2: Structural validation (30%)
        structural_score = self.structural_validator.validate(cleaned)

        # Step 3: Semantic validation (30%)
        semantic_score = self.semantic_validator.compare(
            agent_response, ground_truth
        )

        # Step 4: LLM judge (40%)
        llm_score = self.llm_judge.evaluate(
            query, agent_response, ground_truth
        )

        # Step 5: Calculate final score
        final_score = (0.3 * structural_score +
                       0.3 * semantic_score +
                       0.4 * llm_score)

        result = "PASS" if final_score >= 0.7 else "FAIL"

        # Step 6: Store result
        self.db.store_evaluation(...)

        return result, final_score
```

Testing: - Run on 100 queries from training set - Verify accuracy calculation - Check false positive/negative rates

4.2 Batch Evaluation System

Duration: 2 days

Tasks: - [ ] Create batch processing script - [ ] Add progress tracking - [ ] Implement parallel processing - [ ] Add result aggregation - [ ] Create evaluation reports

Implementation:

```
class BatchEvaluator:
    def evaluate_dataset(self, queries, batch_size=10):
        results = []

        for batch in chunks(queries, batch_size):
            # Process in parallel
            batch_results = self.process_batch(batch)
            results.extend(batch_results)

            # Log progress
            self.log_progress(len(results), len(queries))

        # Calculate metrics
        accuracy = self.calculate_accuracy(results)

        return results, accuracy
```

Deliverable: Evaluation system processing 2,500 queries in <2 hours

4.3 Iterative Agent Improvement

Duration: 5 days

Process: 1. Run evaluation on 1,000 training queries 2. Analyze failures 3. Identify patterns (missing examples, wrong prompts, schema issues) 4. Improve agent (add examples, fix prompts, clarify schema) 5. Re-evaluate 6. Repeat until ≥90% accuracy

Target Metrics:

```
Iteration 1: 70-75% accuracy (baseline)
Iteration 2: 80-85% accuracy (add examples, fix prompts)
Iteration 3: 85-90% accuracy (refine edge cases)
Iteration 4: ≥90% accuracy (final tuning)
```

Deliverable: Agents achieving ≥90% accuracy on test set

Phase 5: Build Monitoring Framework (Week 6-7)

5.1 Drift Detection System

Duration: 3 days

Tasks: - [ ] Create baseline from training queries - [ ] Implement query embedding generation - [ ] Implement drift score calculation - [ ] Implement drift classification - [ ] Add alert system

Implementation:

```
class DriftDetector:
    def __init__(self):
        self.embedder = SentenceTransformer('all-MiniLM-L6-v2')
        self.baseline = self.load_baseline()

    def create_baseline(self, training_queries):
        # Generate embeddings for 1000 training queries
        embeddings = self.embedder.encode(training_queries)

        # Calculate centroid
        centroid = np.mean(embeddings, axis=0)

        # Store baseline
        self.store_baseline(centroid, embeddings)

    def detect_drift(self, new_query):
        # 1. Generate embedding
        query_embedding = self.embedder.encode([new_query])[0]

        # 2. Calculate similarity with baseline
        similarity = cosine_similarity(
            query_embedding,
            self.baseline.centroid
        )

        # 3. Calculate drift score
        drift_score = 1 - similarity

        # 4. Classify
        if drift_score > 0.5:
            classification = "HIGH"
        elif drift_score > 0.3:
            classification = "MEDIUM"
        else:
            classification = "LOW"

        # 5. Store in DB
        self.db.store_drift(query, drift_score, classification)

        # 6. Alert if HIGH
        if classification == "HIGH":
            self.alert("High drift detected")

        return drift_score, classification
```

Testing: - Create synthetic drift (queries about new topics) - Verify detection accuracy - Test alert system

5.2 Error Classification System

Duration: 4 days

Tasks: - [ ] Define error taxonomy (5 categories) - [ ] Implement rule-based classifier - [ ] Implement LLM-based classifier - [ ] Add error enrichment (frequency, severity) - [ ] Create error database schema

Implementation:

```
class ErrorClassifier:
    def __init__(self):
        self.rule_classifier = RuleBasedClassifier()
        self.llm_classifier = LLMClassifier()

    def classify_error(self, query, response, error_msg, logs):
        # Step 1: Try rule-based first (fast)
        category = self.rule_classifier.classify(error_msg)

        if category:
            return category, "RULE_BASED", 1.0

        # Step 2: Use LLM for complex cases
        result = self.llm_classifier.classify(
            query=query,
            response=response,
            error=error_msg,
            logs=logs
        )

        return result['category'], "LLM_BASED", result['confidence']

class RuleBasedClassifier:
    def classify(self, error_msg):
        rules = {
            "syntax error": "SQL_GENERATION_ERROR",
            "table.*not exist": "SQL_SCHEMA_ERROR",
            "timeout": "INTEGRATION_TIMEOUT",
            "authentication": "INTEGRATION_AUTH_ERROR",
            "null": "DATA_MISSING_ERROR"
        }

        for pattern, category in rules.items():
            if re.search(pattern, error_msg, re.IGNORECASE):
                return category

        return None
```

Testing: - Create test cases for each error category - Verify classification accuracy (aim for >85%) - Test LLM fallback for complex errors

Deliverable: Error classification system with >85% accuracy

Phase 6: Database & Storage Layer (Week 7)

6.1 Database Schema Creation

Duration: 2 days

Tasks: - [ ] Design all 5 tables (queries, evaluations, errors, drift_monitoring, baseline) - [ ] Create migration scripts - [ ] Add indexes for performance - [ ] Set up foreign key relationships - [ ] Add pgvector extension for embeddings

Schema:

```sql
-- 1. queries table
CREATE TABLE queries (
    query_id SERIAL PRIMARY KEY,
    query_text TEXT NOT NULL,
    agent_type VARCHAR(20) NOT NULL,
    agent_response TEXT,
    generated_sql TEXT,
    execution_time_ms INTEGER,
    status VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    user_id VARCHAR(100),
    session_id VARCHAR(100)
);

-- 2. evaluations table
CREATE TABLE evaluations (
    evaluation_id SERIAL PRIMARY KEY,
    query_id INTEGER REFERENCES queries(query_id),
    agent_type VARCHAR(20) NOT NULL,
    structural_score FLOAT,
    semantic_score FLOAT,
    llm_score FLOAT,
    final_score FLOAT,
    evaluation_result VARCHAR(10),
    confidence FLOAT,
    reasoning TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 3. errors table
CREATE TABLE errors (
    error_id SERIAL PRIMARY KEY,
    evaluation_id INTEGER REFERENCES evaluations(evaluation_id),
    query_id INTEGER REFERENCES queries(query_id),
    error_category VARCHAR(50),
    error_subcategory VARCHAR(50),
    error_message TEXT,
    stack_trace TEXT,
    severity VARCHAR(20),
    frequency_count INTEGER DEFAULT 1,
    confidence FLOAT,
    first_seen TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_seen TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 4. drift_monitoring table
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE drift_monitoring (
    drift_id SERIAL PRIMARY KEY,
    query_id INTEGER REFERENCES queries(query_id),
    query_embedding VECTOR(384),
    drift_score FLOAT,
    drift_classification VARCHAR(20),
    similarity_to_baseline FLOAT,
    is_anomaly BOOLEAN DEFAULT FALSE,
```

```
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 5. baseline table
CREATE TABLE baseline (
    baseline_id SERIAL PRIMARY KEY,
    agent_type VARCHAR(20) NOT NULL,
    centroid_embedding VECTOR(384),
    num_queries INTEGER,
    avg_query_length FLOAT,
    common_keywords JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    version INTEGER DEFAULT 1
);

-- Indexes
CREATE INDEX idx_query_agent ON queries(agent_type);
CREATE INDEX idx_eval_result ON evaluations(evaluation_result);
CREATE INDEX idx_error_category ON errors(error_category);
CREATE INDEX idx_drift_score ON drift_monitoring(drift_score);
```

6.2 Data Access Layer

Duration: 1 day

Tasks: - [ ] Create database connection manager - [ ] Implement CRUD operations for all tables - [ ] Add connection pooling - [ ] Add transaction support - [ ] Create database utilities

Deliverable: Complete database layer with all tables operational

---

Phase 7: API & Integration Layer (Week 8)

7.1 REST API Development

Duration: 3 days

Tasks: - [ ] Set up FastAPI application - [ ] Implement 6 core endpoints - [ ] Add request validation (Pydantic models) - [ ] Add error handling - [ ] Add CORS support - [ ] Generate API documentation (Swagger)

Endpoints:

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI(title="Unilever Procurement GPT API")

# 1. Query endpoint (for agents)
@app.post("/api/v1/query")
async def process_query(
    query: str,
    agent_type: str  # "spend" or "demand"
):
    """Process natural language query using AI agent"""
    agent = get_agent(agent_type)
    result = agent.process_query(query)
    return result

# 2. Evaluate endpoint
@app.post("/api/v1/evaluate")
async def evaluate_response(
    query: str,
    agent_response: str,
    ground_truth: str,
    agent_type: str
):
    """Evaluate agent response against ground truth"""
    evaluator = EvaluationFramework()
    result = evaluator.evaluate(query, agent_response, ground_truth)
    return result

# 3. Metrics endpoint
@app.get("/api/v1/metrics")
async def get_metrics(
    agent_type: Optional[str] = None,
    time_period: str = "24h"
):
    """Get evaluation metrics"""
    metrics = calculate_metrics(agent_type, time_period)
    return metrics

# 4. Drift endpoint
@app.get("/api/v1/drift")
async def get_drift_status(
    agent_type: Optional[str] = None,
    period: str = "24h"
):
    """Get drift detection status"""
    drift_data = get_drift_data(agent_type, period)
    return drift_data

# 5. Errors endpoint
@app.get("/api/v1/errors")
async def get_errors(
    category: Optional[str] = None,
    limit: int = 100
):
    """Get error summary"""
```

```
    errors = get_error_data(category, limit)
    return errors

# 6. Baseline update endpoint
@app.post("/api/v1/baseline/update")
async def update_baseline(
    agent_type: str,
    queries: List[str]
):
    """Update drift detection baseline"""
    drift_detector = DriftDetector()
    baseline_id = drift_detector.create_baseline(queries)
    return {"status": "success", "baseline_id": baseline_id}
```

Testing: - Test all endpoints with Postman/curl - Verify request/response format - Test error cases

7.2 Async Processing Pipeline

Duration: 2 days

Tasks: - [ ] Set up background task queue (Celery/Redis) - [ ] Implement async evaluation - [ ] Implement async drift detection - [ ] Add job status tracking

Implementation:

```
        from celery import Celery

        celery_app = Celery('tasks', broker='redis://localhost:6379/0')

        @celery_app.task
        def async_evaluate(query_id):
            """Evaluate query in background"""
            # Get query from DB
            query_data = db.get_query(query_id)

            # Run evaluation
            evaluator = EvaluationFramework()
            result = evaluator.evaluate(
                query_data['query'],
                query_data['response'],
                query_data['ground_truth']
            )

            # Store result
            db.store_evaluation(query_id, result)

            return result

        @celery_app.task
        def async_drift_detection(query_id):
            """Detect drift in background"""
            query_data = db.get_query(query_id)

            detector = DriftDetector()
            drift_score, classification =
detector.detect_drift(query_data['query'])

            db.store_drift(query_id, drift_score, classification)

            return drift_score, classification
```

Deliverable: Complete API with async processing

Phase 8: Dashboard & Visualization (Week 9)

8.1 Dashboard Development

Duration: 3 days

Tasks: - [ ] Set up Streamlit application - [ ] Create 3 main panels (Evaluation, Drift, Errors) - [ ] Add charts and visualizations - [ ] Add real-time data refresh - [ ] Add filters and date range selection

Dashboard Structure:

```
        import streamlit as st
        import plotly.express as px

        def main():
            st.set_page_config(page_title="Procurement GPT Monitor",
layout="wide")
            st.title("🤖 Procurement GPT POC - Monitoring Dashboard")

            # Sidebar filters
            agent_filter = st.sidebar.selectbox("Agent", ["All", "Spend",
"Demand"])
            date_range = st.sidebar.date_input("Date Range", [])

            # Main tabs
            tab1, tab2, tab3 = st.tabs(["Evaluation", "Drift Detection",
"Errors"])

            with tab1:
                show_evaluation_panel(agent_filter, date_range)

            with tab2:
                show_drift_panel(agent_filter, date_range)

            with tab3:
                show_error_panel(agent_filter, date_range)

        def show_evaluation_panel(agent_filter, date_range):
            # Metrics
            col1, col2, col3, col4 = st.columns(4)

            with col1:
                st.metric("Overall Accuracy", "92.5%", "+2.3%")
            with col2:
                st.metric("Spend Agent", "94.0%", "+1.5%")
            with col3:
                st.metric("Demand Agent", "91.0%", "+3.1%")
            with col4:
                st.metric("Pass Rate", "88.0%", "+0.8%")

            # Charts
            st.subheader("Accuracy Trends")
            accuracy_data = fetch_accuracy_trends(agent_filter, date_range)
            fig = px.line(accuracy_data, x='date', y='accuracy', color='agent')
            st.plotly_chart(fig, use_container_width=True)

            st.subheader("Pass/Fail Distribution")
            distribution_data = fetch_pass_fail_distribution(agent_filter)
            fig = px.pie(distribution_data, names='result', values='count')
            st.plotly_chart(fig)
```

Charts to Include: 1. **Evaluation Panel:** - Accuracy over time (line chart) - Pass/Fail distribution (pie chart) - Accuracy by agent (bar chart) - Recent evaluations table

2. **Drift Panel:**
    - Drift score timeline (area chart)
    - Query distribution shift (histogram)
    - Anomaly detection (scatter plot)
    - Drift alerts list
3. **Error Panel:**
    - Error distribution by category (pie chart)
    - Error frequency over time (line chart)
    - Top 10 errors (bar chart)
    - Recent errors table with details

8.2 Alerting System

Duration: 2 days

Tasks: - [ ] Implement email alerts (SMTP) - [ ] Implement Slack/Teams webhooks - [ ] Set up alert rules - [ ] Add alert history

Implementation:

```python
class AlertManager:
    def __init__(self):
        self.email_sender = EmailSender()
        self.slack_sender = SlackSender()

    def check_and_alert(self):
        # Check drift
        high_drift = self.check_high_drift()
        if high_drift:
            self.send_alert(
                level="HIGH",
                message="High drift detected",
                data=high_drift
            )

        # Check accuracy drop
        accuracy = self.get_current_accuracy()
        if accuracy < 0.85:
            self.send_alert(
                level="CRITICAL",
                message=f"Accuracy dropped to {accuracy:.1%}",
                data={"accuracy": accuracy}
            )

        # Check error spike
        error_count = self.get_recent_errors(hours=1)
        if error_count > 10:
            self.send_alert(
                level="HIGH",
                message=f"Error spike: {error_count} errors in 1 hour",
                data={"error_count": error_count}
            )

    def send_alert(self, level, message, data):
        if level == "CRITICAL":
            self.email_sender.send(...)
            self.slack_sender.send(...)
        elif level == "HIGH":
            self.slack_sender.send(...)

        # Log alert
        self.db.store_alert(level, message, data)
```

Deliverable: Complete dashboard with alerting system

Phase 9: Testing & Validation (Week 9-10)

9.1 System Integration Testing

Duration: 2 days

Tasks: - [ ] End-to-end testing of complete flow - [ ] Test agent → evaluation → storage → dashboard - [ ] Test drift detection in real-time - [ ] Test error classification on various error types - [ ] Load testing (simulate multiple concurrent users) - [ ] API endpoint testing

9.2 Accuracy Validation

Duration: 2 days

Tasks: - [ ] Run evaluation on full validation set (500 queries) - [ ] Calculate final accuracy metrics - [ ] Analyze false positives and false negatives - [ ] Generate accuracy report - [ ] Verify ≥90% accuracy target met

Validation Report:

```
Validation Results (500 queries):
- Overall Accuracy: 91.2%
- Spend Agent: 92.5%
- Demand Agent: 89.9%
- True Positives: 410
- True Negatives: 46
- False Positives: 22
- False Negatives: 22
- Precision: 94.9%
- Recall: 94.9%
- F1-Score: 94.9%
```

9.3 Performance Testing

Duration: 1 day

Tasks: - [ ] Measure agent response time (target: <5s) - [ ] Measure evaluation time (target: <2s per query) - [ ] Test concurrent query handling - [ ] Database query performance - [ ] API response time

Deliverable: Complete test report with all metrics

Phase 10: Documentation & Handover (Week 10)

10.1 Technical Documentation

Duration: 2 days

Tasks: - [ ] Architecture documentation - [ ] API documentation (Swagger + README) - [ ] Database schema documentation - [ ] Agent implementation guide - [ ] Deployment guide

Documents:

```
docs/
├── architecture.md
├── api_reference.md
├── database_schema.md
├── agent_guide.md
├── deployment_guide.md
├── troubleshooting.md
└── maintenance_guide.md
```

10.2 User Documentation

Duration: 1 day

Tasks: - [ ] Dashboard user guide - [ ] API usage examples - [ ] Common queries guide - [ ] Alert configuration guide

10.3 Deployment to Azure

Duration: 2 days

Tasks: - [ ] Set up Azure resources - [ ] Deploy database to Azure PostgreSQL - [ ] Deploy API to Azure App Service - [ ] Deploy dashboard to Azure App Service - [ ] Configure monitoring (Application Insights) - [ ] Set up CI/CD pipeline

Deliverable: Production-ready POC deployed on Azure

Project Timeline Summary

```
Week 1: Setup & Data Discovery
Week 2-3: Build AI Agents
Week 3-4: Create Ground Truth Dataset
Week 4-5: Build Evaluation Framework
Week 6-7: Build Monitoring Framework
Week 7: Database & Storage Layer
Week 8: API & Integration Layer
Week 9: Dashboard & Visualization
Week 9-10: Testing & Validation
Week 10: Documentation & Handover
```

Resource Requirements

Team Structure

```
Role 1: Senior ML Engineer (Lead)
- Agent development
- Evaluation framework
- LLM prompt engineering
- 40 hours/week

Role 2: Backend Developer
- API development
- Database design
- Integration layer
- 40 hours/week

Role 3: Data Engineer (Part-time)
- Data collection
- Ground truth generation
- Schema documentation
- 20 hours/week
```

Infrastructure

```
Development:
- 3 development machines
- Local PostgreSQL instances
- Ollama running locally

Production (Azure):
- Azure PostgreSQL (Standard tier)
- 2x Azure App Service (B2 tier)
- Azure Application Insights
- Azure Storage (for logs)

Estimated Cost: $200-300/month
```

Tools & Software

```
Free/Open Source:
- Python 3.11
- Ollama + Llama 3.1
- PostgreSQL + pgvector
- FastAPI
- Streamlit
- Sentence Transformers
- FAISS

Paid (Optional):
- GitHub (for version control)
- Azure services
```

Risk Management

Risk 1: Agent Accuracy Below 90%

Probability: Medium **Impact:** High **Mitigation:** - Collect more high-quality examples - Fine-tune prompts iteratively - Consider using better LLM (GPT-4) if budget allows - Add more validation layers - Increase ground truth quality

Risk 2: Data Access Delays

Probability: Medium **Impact:** High **Mitigation:** - Start with sample/dummy data - Build agent architecture in parallel - Use synthetic queries for initial testing - Escalate to Unilever management early

Risk 3: LLM Performance Issues

Probability: Low **Impact:** Medium **Mitigation:** - Have backup LLM options (Groq, HuggingFace) - Optimize prompts for smaller models - Cache common queries - Use GPU if available

Risk 4: Scope Creep

Probability: High **Impact:** Medium **Mitigation:** - Strict adherence to POC scope - Document all feature requests for post-POC - Regular stakeholder alignment - Clear acceptance criteria

---

Success Criteria

Must Have (POC Acceptance)

- ✅ Spend Agent operational with ≥90% accuracy
- ✅ Demand Agent operational with ≥90% accuracy
- ✅ Evaluation Framework with ≥90% prediction accuracy
- ✅ Drift Detection system functional
- ✅ Error Classification system (5 categories)
- ✅ Dashboard showing all metrics
- ✅ REST API with 6 endpoints
- ✅ Evaluation of 2,500 prompts completed
- ✅ Complete documentation

Nice to Have (Post-POC)

- Automatic baseline refresh
- Multi-model ensemble evaluation

- Advanced analytics and predictions
- Integration with Unilever's existing systems
- Real-time streaming dashboard

Deliverables Checklist

Code

- [ ] Spend Agent (Python)
- [ ] Demand Agent (Python)
- [ ] Evaluation Framework (Python)
- [ ] Monitoring Framework (Python)
- [ ] REST API (FastAPI)
- [ ] Dashboard (Streamlit)
- [ ] Database migrations
- [ ] Tests (pytest)

Data

- [ ] 2,500 queries with ground truth
- [ ] Schema documentation
- [ ] Example query database
- [ ] Baseline embeddings

Documentation

- [ ] Architecture diagrams
- [ ] API documentation
- [ ] User guides
- [ ] Deployment guide
- [ ] Maintenance guide

Infrastructure

- [ ] Azure deployment
- [ ] CI/CD pipeline
- [ ] Monitoring setup

Next Steps

Immediate Actions (Week 1)

1. **Get Unilever Approval:**
   - Review this plan with stakeholders
   - Confirm timeline and resources
   - Get database access credentials
2. **Team Assembly:**
   - Assign roles
   - Set up communication channels
   - Schedule daily standups
3. **Environment Setup:**
   - Create Git repository
   - Set up development environments
   - Install all dependencies
4. **Kickoff Meeting:**
   - Review plan with team
   - Assign initial tasks
   - Set up project tracking (Jira/Trello)

Contact Points at Unilever

- **Data Access:** [Name, Email]
- **Business Logic:** [Name, Email]
- **Ground Truth Review:** [Name, Email]

- **POC Acceptance:** [Name, Email]

Appendix

A. Technology Stack Details

```
Backend:
- Python 3.11
- FastAPI 0.109+
- Uvicorn (ASGI server)

Database:
- PostgreSQL 15+
- pgvector extension
- SQLAlchemy (ORM)

AI/ML:
- Ollama (LLM runtime)
- Llama 3.1 8B (free model)
- Sentence Transformers (embeddings)
- FAISS (vector search)

Data Processing:
- pandas
- numpy
- scipy
- scikit-learn

Frontend:
- Streamlit 1.30+
- Plotly (charts)

DevOps:
- Docker
- Azure App Service
- Azure PostgreSQL
- Application Insights
```

B. File Structure

```
unilever-procurement-poc/
├── agents/
│   ├── spend_agent.py
│   ├── demand_agent.py
│   └── base_agent.py
├── evaluation/
│   ├── evaluator.py
│   ├── validators.py
│   └── llm_judge.py
├── monitoring/
│   ├── drift_detector.py
│   ├── error_classifier.py
│   └── alerting.py
├── api/
│   ├── main.py
│   ├── routes.py
│   └── models.py
├── database/
│   ├── connection.py
│   ├── models.py
│   └── migrations/
├── dashboard/
│   └── app.py
├── data/
│   ├── ground_truth/
│   ├── examples/
│   └── schemas/
├── tests/
├── docs/
├── config/
├── requirements.txt
└── README.md
```

C. Metrics to Track

```
Daily:
- Agent query count
- Success/failure rate
- Average response time
- Error distribution

Weekly:
- Evaluation accuracy
- Drift score trends
- Error frequency
- Performance benchmarks

Monthly:
- Overall system health
- Cost analysis
- Usage patterns
- Improvement opportunities
```

Document Version: 1.0 **Last Updated:** February 2026 **Author:** Implementation Team **Status:** Ready for Review