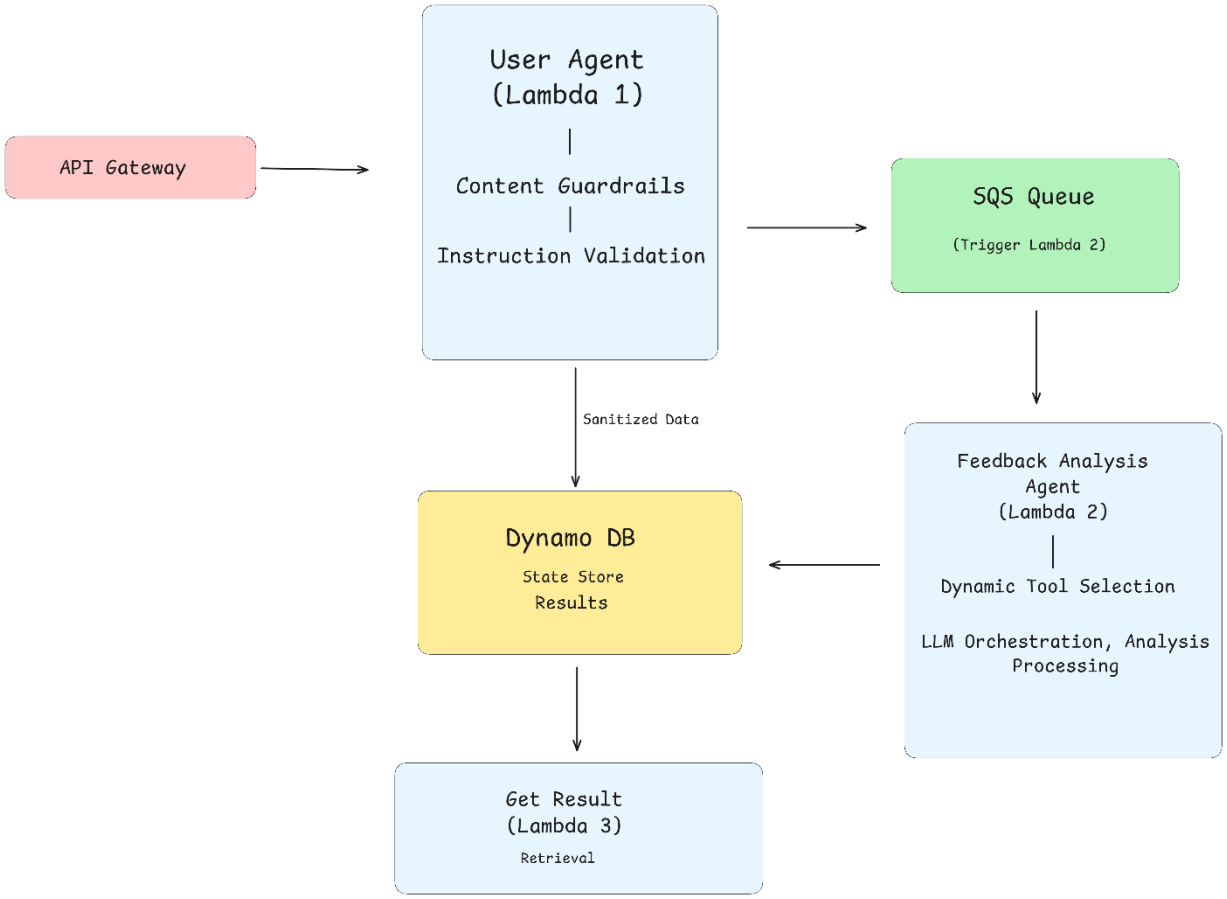
Intelligent LLM Agent System - Architecture and Design Decisions & Setup Instructions

System Overview

The Intelligent LLM Agent System is a multi-agent solution designed to process customer feedback through dynamic tool selection and intelligent analysis. The system employs a guardrail-first approach with asynchronous processing to ensure local safety and reliability.

System Architecture Overview

High-Level Architecture Diagram



Component Interaction Flow

1. **Request Initiation**: Client sends feedback via API Gateway
2. **Initial Processing**: User Agent validates and applies guardrails
3. **Asynchronous Queuing**: Validated data queued in SQS
4. **Tool Processing**: Tool Agent processes with dynamic tool selection
5. **State Management**: Results stored in DynamoDB
6. **Result Retrieval**: Results Agent formats and returns analysis

Multi-Agent Design

Agent Responsibilities Matrix

Agent	Primary Role	Secondary Roles	Technologies Used
User Agent	User Interaction & Guardrails	Input Validation, State Initialization	OpenAI GPT, AWS Lambda, DynamoDB, SQS
Tool Agent	Dynamic Tool Execution	LLM Orchestration, Analysis Processing	OpenAI Agents Framework, TextBlob, Custom Tools
Results Agent	Data Retrieval & Formatting	State Management, Response Structuring	DynamoDB Query, JSON Processing

Core Components

User Agent (Guardrail Agent) Design

Purpose & Responsibilities

The User Agent serves as the system's entry point and security gateway, implementing comprehensive guardrails for safe and reliable processing.

Core Components

1. Guardrail Agent Class

```
class GuardrailAgent:
    """User interaction agent with guardrails"""

    def __init__(self):
        self.guardrail_prompts = {
            "content_filter": "...",
            "instruction_validator": "..."
        }
```

2. Content Filtering System

- **Purpose:** Detect and filter inappropriate content
- **Methodology:** LLM-based content analysis using structured prompts
- **Categories Checked:**
    - Hate speech and harassment
    - Spam or irrelevant content
    - Personal information requiring redaction

3. Instruction Validation System

- **Purpose**: Validate user instructions for safety and scope
- **Checks Performed:**
    - Malicious request detection
    - Scope boundary enforcement
    - System manipulation attempts

Decision Logic

Content Safety Decision Tree

```
Input Content
```

- ├── Contains hate speech/threats? → REJECT
- ├── Contains PII that needs redaction? → SANITIZE
- ├── Spam/irrelevant content? → REJECT
- └── Safe content → PROCEED

Instruction Validation Logic

```
User Instructions
```

- ├── Empty/None? → USE DEFAULT TOOLS
- ├── Contains malicious requests? → REJECT
- ├── Out of scope operations? → REJECT
- ├── System manipulation attempts? → REJECT
- └── Valid instructions → SANITIZE & PROCEED

State Management Strategy

DynamoDB Record Structure

```
{
  "feedback_id": "unique_identifier",
  "timestamp": "unix_timestamp_for_sort",
  "status": "processing|completed|failed",
  "original_data": {
    "feedback_text": "sanitized_content",
    "instructions": "validated_instructions",
    "customer_name": "name",
    "timestamp": "iso_timestamp"
  },
  "created_at": "iso_timestamp",
  "updated_at": "iso_timestamp"
}
```

Tool Agent (Feedback Analysis Agent) Design

Purpose & Responsibilities

The Tool Agent implements intelligent analysis capabilities with dynamic tool selection based on user instructions and content analysis.

Architecture Components

1. OpenAI Agents Framework Integration

```
from agents import Agent, Runner, function_tool
from pydantic import BaseModel

feedback_analysis_agent = Agent(
    name="Feedback Processing Agent",
    instructions="...",
    tools=[sentiment_analysis, topic_categorization,
           keyword_contextualization, summarization],
    output_type=SummaryAnalysis
)
```

2. Tool Implementation Strategy

Each tool follows a consistent pattern:

- **Pydantic Models:** Strict schema compliance for inputs/outputs
- **Error Handling:** Graceful degradation with meaningful error messages
- **Fallback Logic:** Basic processing when advanced analysis fails

Tool Specifications

1. Sentiment Analysis Tool

Processing Logic:

1. Basic sentiment using TextBlob (polarity/subjectivity)
2. Enhanced analysis via OpenAI for emotional context
3. Confidence scoring based on analysis consistency
4. Fallback to basic analysis if enhanced processing fails

2. Topic Categorization Tool

Categorization Logic:

1. Use predefined business-relevant topics
2. LLM-based classification with confidence scoring
3. Multi-topic support for complex feedback
4. Reasoning explanation for transparency

Predefined Topics:

- Product Quality, Delivery, Customer Support
- Pricing, Website/App, Billing
- Returns, Shipping, User Experience

3. Keyword Contextualization Tool

Analysis Strategy:

1. Statistical frequency analysis (Counter)
2. Context-aware extraction via LLM
3. Relevance scoring (0-1 scale)
4. Entity recognition and phrase extraction

4. Summarization Tool

Summarization Logic:

1. Synthesize insights from previous tool outputs
2. Generate actionable business recommendations
3. Prioritize actions by impact and urgency
4. Assign departmental responsibility

Dynamic Tool Selection Logic

Instruction Interpretation Engine

The agent uses LLM-based instruction parsing to determine tool execution strategy:

Decision Matrix

Instruction Type	Tools Executed	Reasoning
No instructions	All tools → Summarization	Comprehensive analysis
"Sentiment only"	Sentiment → Summarization	Focused sentiment analysis
"Topics and keywords"	Topic + Keyword → Summarization	Content categorization focus
"Actionable insights"	All tools → Enhanced Summarization	Business-focused analysis

Tool Sequencing Strategy

1. Parse Instructions: LLM determines required tools
2. Execute Tools: Sequential execution based on dependencies
3. Context Passing: Previous tool outputs inform subsequent tools
4. Final Summarization: Always executed to provide structured output

Example Instruction Interpretations:

- "Focus on sentiment analysis only" → Execute only sentiment_analysis tool
- "Analyze sentiment and suggest improvements" → Execute sentiment_analysis + summarization
- "Identify key topics and summarize actionable points" → Execute topic_categorization + summarization
- No instructions → Execute all tools in logical sequence

Results Agent Design

Response Formatting System

Multiple formatting layers ensure consistent, user-friendly outputs:

1. Status Detection: Determine current processing state
2. Data Extraction: Parse analysis results from DynamoDB
3. Structure Formatting: Organize results into logical sections
4. Recommendation Grouping: Group by priority and department

Retrieve Analysis Results


```
{
  "feedback_id": "12345",
  "status": "completed",
  "message": "Analysis completed successfully",
  "results": {
    "executive_summary": "Customer expresses satisfaction with product quality but frustration with delivery delays.",
    "key_insights": {
      "main_points": [
        "Positive product feedback",
        "Delivery service issues",
        "Overall mixed sentiment"
      ],
      "customer_impact_assessment": "Moderate impact - satisfied with product but delivery issues may affect future
purchases"
    },
    "actionable_recommendations": {
      "total_recommendations": 2,
      "immediate_actions": [
        {
          "action": "Review delivery logistics and identify bottlenecks",
          "department": "Operations",
          "timeline": "Within 1 week",
          "priority": "high"
        }
      ]
    }
```


Current Limitations

- **Caching:** Redis caching not implemented (planned enhancement)
- **Infrastructure as Code:** Manual deployment (Terraform planned)
- **Advanced Analytics:** Limited to predefined analysis tools

Future Enhancements

1. **Redis Caching:** Implement result caching for improved performance
2. **Terraform Deployment:** Complete IaC implementation

Setup Instructions

Prerequisites

- AWS Account with appropriate permissions
- Python 3.9+
- Docker (for building Lambda layers)
- OpenAI API Key

Environment Variables

Create a .env file with the following variables:

```
OPENAI_API_KEY=your_openai_api_key_here
DYNAMODB_TABLE=feedback-state-table
SQS_QUEUE_URL_FEEDBACK=your_sqs_queue_url
```

AWS Resources Setup

1. DynamoDB Table

```
# Create DynamoDB table
aws dynamodb create-table \
  --table-name feedback-state-table \
  --attribute-definitions \
    AttributeName=feedback_id,AttributeType=S \
    AttributeName=timestamp,AttributeType=N \
  --key-schema \
    AttributeName=feedback_id,KeyType=HASH \
    AttributeName=timestamp,KeyType=RANGE \
  --provisioned-throughput \
    ReadCapacityUnits=5,WriteCapacityUnits=5
```

2. SQS Queue

```
aws sqs create-queue --queue-name feedback-processing-queue
```

Lambda Layers Creation

User Agent Layer

```
cd create_layers_using_docker/user_agent_layer
docker build -t openai-layer .
docker run --rm -v $(pwd):/output openai-layer
```

Feedback Analysis Agent Layer

```
cd create_layers_using_docker/feedback_analysis_agent
docker build -t feedback-layer .
docker run --rm -v $(pwd):/output feedback-layer
```

3. AWS DynamoDB Table Setup

Create the DynamoDB table:

```
aws dynamodb create-table \
  --table-name feedback-analysis-state \
  --attribute-definitions \
    AttributeName=feedback_id,AttributeType=S \
    AttributeName=timestamp,AttributeType=N \
  --key-schema \
    AttributeName=feedback_id,KeyType=HASH \
    AttributeName=timestamp,KeyType=RANGE \
  --billing-mode PAY_PER_REQUEST \
  --region us-east-1
```

Table Schema:

- **Primary Key:** feedback_id (String)
- **Sort Key**: timestamp (Number)
- **Billing Mode:** On-Demand
- **Attributes:**
    - status (String): processing, completed, failed
    - original_data (Map): Original feedback input
    - results (Map): Analysis results
    - created_at (String): ISO timestamp
    - updated_at (String): ISO timestamp

4. Amazon SQS Queue Setup

Create the SQS queue:

```
aws sqs create-queue \
  --queue-name feedback-processing \
  --attributes VisibilityTimeoutSeconds=300,MessageRetentionPeriod=1209600 \
  --region us-east-1
```

Queue Configuration:

- Name: feedback-processing
- Visibility Timeout: 300 seconds (5 minutes)

5. IAM Roles and Policies

User Agent Role

Create IAM role for User Agent Lambda:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:UpdateItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/feedback-analysis-state"
    },
    {
      "Effect": "Allow",
      "Action": [
        "sqs:SendMessage"
      ],
      "Resource": "arn:aws:sqs:*:*:feedback-processing"
    }
  ]
}
```

Tool Agent Role

Create IAM role for Tool Agent Lambda:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateItem",
        "dynamodb:GetItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/feedback-analysis-state"
    },
    {
      "Effect": "Allow",
      "Action": [
        "sqs:ReceiveMessage",
        "sqs:DeleteMessage",
        "sqs:GetQueueAttributes"
      ],
      "Resource": "arn:aws:sqs:*:*:feedback-processing"
    }
  ]
}
```

Results Agent Role

Create IAM role for Results Agent Lambda:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:Query",
        "dynamodb:GetItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/feedback-analysis-state"
    }
  ]
}
```

6. Lambda Functions Deployment

User Agent Lambda (Lambda 1)

1. **Create Lambda Function:**
   - Runtime: Python 3.9
   - Handler: user_agent.lambda_handler
   - Timeout: 60 seconds
   - Memory: 256 MB
2. **Upload Code:**

```
zip -r user-agent.zip handler/user_agent.py
aws lambda update-function-code \
  --function-name user-agent \
  --zip-file fileb://user-agent.zip
```

3. **Add Layer:**

```
aws lambda update-function-configuration \
  --function-name user-agent \
  --layers arn:aws:lambda:us-east-1:YOUR_ACCOUNT:layer:openai-layer:1
```

4. Set Environment Variables:

```
aws lambda update-function-configuration \
  --function-name user-agent \
  --environment Variables='{
    "OPENAI_API_KEY":"your_key_here",
    "DYNAMODB_TABLE":"feedback-analysis-state",
    "SQS_QUEUE_URL_FEEDBACK":"your_queue_url_here"
  }'
```

Tool Agent Lambda (Lambda 2)

1. Create Lambda Function:
   - Runtime: Python 3.9
   - Handler: feedback_analysis_agent.lambda_handler
   - Timeout: 300 seconds
   - Memory: 1024 MB
2. Upload Code and Configure (similar to User Agent)
3. Add SQS Trigger:

```
aws lambda create-event-source-mapping \
  --function-name tool-agent \
  --event-source-arn arn:aws:sqs:us-east-1:YOUR_ACCOUNT:feedback-processing \
  --batch-size 10
```

Results Agent Lambda (Lambda 3)

1. Create Lambda Function:
   - Runtime: Python 3.9
   - Handler: get_results.lambda_handler
   - Timeout: 30 seconds
   - Memory: 256 MB
2. Upload Code and Configure (similar to others)

7. API Gateway Setup

Create REST API

3. Create API:

```
aws apigateway create-rest-api \
  --name feedback-analysis-api \
  --description "Intelligent LLM Agent API"
```

4. Create Resources and Methods: POST /feedback (User Agent):
   - Integration: Lambda Function (user-agent)
   - Enable CORS
   - Request validation enabled

   GET /results/{feedback_id} (Results Agent):

   - Integration: Lambda Function (results-agent)
   - Path parameter: feedback_id
   - Enable CORS
5. Deploy API:

```
aws apigateway create-deployment \
  --rest-api-id YOUR_API_ID \
  --stage-name prod
```

8. Testing the Setup

Test Script

Run the test_api.py script inside the Test_API directory.

9. Deployment Automation (Future Enhancement)

Note: Not tested

For automated deployment, use the provided serverless.yaml:

bash

 # Install Serverless Framework

npm install -g serverless

 # Deploy

serverless deploy --stage prod

Verification Checklist

- DynamoDB table created with correct schema
- SQS queue created and accessible
- Lambda functions deployed with correct roles
- API Gateway configured with proper endpoints
- Environment variables set correctly
- Test successful end-to-end flow
- CloudWatch logs accessible
- Error handling tested

Support

For issues during setup:

6. Check CloudWatch logs for detailed error messages
7. Verify IAM permissions and roles
8. Test each component individually
9. Ensure all environment variables are set correctly
10. Validate API keys and AWS credentials

📁 Project Structure

```
INTELLIGENT_LLM_AGENTS/
├── create_layers_using_docker/        # Lambda layers for dependencies
|   ├── feedback_analysis_agent/
| |   ├── build_layer.bat
| |   ├── Dockerfile
| |   ├── openai-agents-layer.zip
| |   └── requirements.txt
|   └── user_agent_layer/
|       ├── build_layer.bat
|       ├── Dockerfile
|       ├── openai-layer.zip
|       └── requirements.txt
├── handler/                    # Lambda function handlers
|   ├── feedback_analysis_agent.py     # Tool agent implementation
|   ├── get_results.py            # Results retrieval agent
|   └── user_agent.py             # User interaction agent
├── Test_API/                   # Testing utilities
|   └── test_api.py               # API testing script
├── .env                        # Environment variables
├── README.md                   # This file
├── requirements.txt            # Python dependencies
└── serverless.yaml             # Serverless configuration (optional)
```