Omnichannel Customer Support System

Project Documentation

Version: 1.0
Last Updated: April 7, 2025
Status: Planning Phase

Table of Contents

Project Overview

The Omnichannel Customer Support System is designed to streamline customer support operations by intelligently routing customer queries across multiple communication channels to the most appropriate support staff based on skills and workload. The system incorporates AI-powered classification, RAG-based knowledge retrieval, and data collection for continuous improvement.

Objectives

- Create a unified backend for handling customer support queries from multiple channels
- Implement intelligent query classification and routing
- Provide AI-assisted solutions for customer support staff
- Build a comprehensive data collection system for future ML model training
- Design with scalability and extensibility in mind

Current Scope

- Email as the initial communication channel
- Backend systems only (no frontend development)
- AI-powered classification using Groq
- RAG implementation with vector databases
- Skills-based and queue-based routing

Future Extensions

- Integration with Slack, Twitter, WhatsApp
- Custom in-house NLP model to replace Groq
- Advanced analytics dashboard
- Automated reply suggestions

High-Level Components

1. **Channel Integration Layer**
    - Email Processor
    - Future Channel Adapters (Slack, Twitter, WhatsApp)
    - Unified Message Format Converter
2. **Classification Engine**
    - Query Analyzer
    - Category Classifier
    - Criticality Assessor
    - Routing Decision Maker
3. **Routing & Queue System**
    - Skills Database
    - Agent Availability Tracker
    - Load Balancer
    - Priority Queue Manager
4. **Knowledge Base System**

    - Document Processor
    - Vector Store
    - Reranking Engine
    - Response Generator

5. **Data Collection & Storage**

    - Query Tracker
    - Resolution Recorder
    - Analysis Database
    - ML Training Data Extractor

6. **Monitoring & Admin Tools**

    - System Health Monitor
    - Performance Analytics
    - Configuration Management
    - Reporting Tools

Development Phases

Phase 1: Foundation (Weeks 1-4)

- Set up core FastAPI backend structure
- Implement email integration
- Design and create database schemas
- Set up message queue infrastructure
- Implement basic authentication and security

Deliverables:

- Working backend that can receive and process emails
- Database for storing support tickets and agent data
- Basic message queue integration
- Project documentation framework

Phase 2: Classification & Routing (Weeks 5-8)

- Integrate Groq LLM for query classification
- Build skills database and matching algorithm
- Implement routing logic and queue management
- Develop agent workload monitoring

Deliverables:

- Functional classification system
- Skills-based routing mechanism
- Round-robin load balancing
- Agent capacity management

Phase 3: RAG Implementation (Weeks 9-12)

- Set up document acquisition pipeline
- Implement vector database integration
- Build reranking system
- Create solution recommendation engine

Deliverables:

- Document ingestion system
- Vector search functionality
- Top-3 solution recommendations for agents
- Direct resolution for low-criticality issues

Phase 4: Data Collection & Analytics (Weeks 13-16)

- Design and implement data collection pipeline
- Create analytics database
- Set up monitoring and alerting
- Implement feedback collection system

Deliverables:

- Complete data pipeline for query → assignment → solution
- Basic analytics dashboard
- Monitoring system with alerts
- Feedback collection mechanisms

Phase 5: Optimization & Testing (Weeks 17-20)

- Performance optimization
- Load testing
- Security auditing

- Documentation finalization

Deliverables:

- Optimized system capable of handling production load
- Comprehensive test coverage
- Security compliance verification
- Complete system documentation

Technology Stack

Core Framework

- **FastAPI**: Main backend framework
- **Pydantic**: Data validation and settings management
- **Uvicorn/Gunicorn**: ASGI server

Data Processing & AI

- **LangChain**: For RAG pipelines and LLM orchestration
- **CrewAI**: For agent-based workflow automation
- **Groq**: LLM provider for classification and generation
- **Crawl4AI**: Document acquisition and processing

Databases

- **MySQL**: Primary relational database
- **Vector Database**: One of (Qdrant / Pinecone / Chroma Cloud)
- **Redis**: Caching and session management

Message Queue & Async

- **RabbitMQ**: Message queue for ticket distribution
- **Celery**: Distributed task queue for async processing

Infrastructure & DevOps

- **AWS Services**:
  - S3: Document storage
  - Lambda: Serverless functions
  - SQS/SNS: Additional queue capabilities
  - CloudWatch: Basic monitoring
- **Docker**: Containerization
- **Kubernetes**: Container orchestration (if needed)

Monitoring & Observability

- **Grafana**: Dashboards and visualization
- **Prometheus**: Metrics collection
- **Sentry**: Error tracking

Integration Tools

- **Zapier/n8n/MCP**: For external integrations (if needed)

API Specifications

Core API Endpoints

Authentication & User Management

```
 POST /api/v1/auth/login
POST /api/v1/auth/logout
GET /api/v1/users/me
PUT /api/v1/users/me
```

Ticket Management

```
POST /api/v1/tickets
GET /api/v1/tickets
GET /api/v1/tickets/{ticket_id}
PUT /api/v1/tickets/{ticket_id}
GET /api/v1/tickets/queue
POST /api/v1/tickets/{ticket_id}/assign
POST /api/v1/tickets/{ticket_id}/resolve
```

Agent Management

```
GET /api/v1/agents
GET /api/v1/agents/{agent_id}
PUT /api/v1/agents/{agent_id}
POST /api/v1/agents/{agent_id}/skills
GET /api/v1/agents/{agent_id}/queue
PUT /api/v1/agents/{agent_id}/status
```

Category Management

```
GET /api/v1/categories
POST /api/v1/categories
PUT /api/v1/categories/{category_id}
DELETE /api/v1/categories/{category_id}
```

Knowledge Base

```
POST /api/v1/documents
GET /api/v1/documents
GET /api/v1/documents/{document_id}
PUT /api/v1/documents/{document_id}
DELETE /api/v1/documents/{document_id}
POST /api/v1/documents/crawl
POST /api/v1/documents/upload
```

RAG Endpoints

```
POST /api/v1/rag/query
GET /api/v1/rag/suggestions/{ticket_id}
```

Analytics

```
GET /api/v1/analytics/tickets
GET /api/v1/analytics/agents
GET /api/v1/analytics/performance
GET /api/v1/analytics/categories
```

Admin Endpoints

```
GET /api/v1/admin/system/status
POST /api/v1/admin/system/config
GET /api/v1/admin/logs
POST /api/v1/admin/reindex
```

API Authentication

- JWT-based authentication
- Role-based access control
- API keys for service-to-service communication

API Documentation

- OpenAPI/Swagger documentation
- Endpoint-specific rate limits
- Versioning strategy

Database Design

Main Tables

Users

```
CREATE TABLE users (
    user_id VARCHAR(36) PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    role ENUM('admin', 'agent', 'supervisor') NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

Agents

```
CREATE TABLE agents (
    agent_id VARCHAR(36) PRIMARY KEY,
    user_id VARCHAR(36) NOT NULL,
    status ENUM('available', 'busy', 'away', 'offline') DEFAULT 'offline',
    max_concurrent_tickets INT DEFAULT 5,
    current_load INT DEFAULT 0,
    last_activity TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Skills

```
CREATE TABLE skills (
    skill_id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

AgentSkills

```
CREATE TABLE agent_skills (
    agent_id VARCHAR(36),
    skill_id VARCHAR(36),
    proficiency_level INT DEFAULT 1,
    PRIMARY KEY (agent_id, skill_id),
    FOREIGN KEY (agent_id) REFERENCES agents(agent_id),
    FOREIGN KEY (skill_id) REFERENCES skills(skill_id)
);
```

Categories

```
CREATE TABLE categories (
    category_id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    parent_category_id VARCHAR(36),
    required_skill_id VARCHAR(36),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (parent_category_id) REFERENCES categories(category_id),
    FOREIGN KEY (required_skill_id) REFERENCES skills(skill_id)
);
```

Channels

```
CREATE TABLE channels (
    channel_id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    type ENUM('email', 'slack', 'twitter', 'whatsapp') NOT NULL,
    config JSON,
    active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Tickets

```
CREATE TABLE tickets (
    ticket_id VARCHAR(36) PRIMARY KEY,
    external_id VARCHAR(255),
    channel_id VARCHAR(36) NOT NULL,
    customer_id VARCHAR(36),
    subject VARCHAR(255),
    content TEXT,
    category_id VARCHAR(36),
    criticality INT DEFAULT 1,
    status ENUM('new', 'open', 'in_progress', 'resolved', 'closed') DEFAULT 'new',
    assigned_agent_id VARCHAR(36),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (channel_id) REFERENCES channels(channel_id),
    FOREIGN KEY (category_id) REFERENCES categories(category_id),
    FOREIGN KEY (assigned_agent_id) REFERENCES agents(agent_id)
);
```

Customers

```
CREATE TABLE customers (
    customer_id VARCHAR(36) PRIMARY KEY,
    email VARCHAR(255),
    name VARCHAR(255),
    metadata JSON,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

Messages

```
CREATE TABLE messages (
    message_id VARCHAR(36) PRIMARY KEY,
    ticket_id VARCHAR(36) NOT NULL,
    sender_type ENUM('customer', 'agent', 'system') NOT NULL,
    sender_id VARCHAR(36),
    content TEXT,
    metadata JSON,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (ticket_id) REFERENCES tickets(ticket_id)
);
```

Documents

```
CREATE TABLE documents (
    document_id VARCHAR(36) PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    content TEXT,
    metadata JSON,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

DocumentChunks

```
CREATE TABLE document_chunks (
    chunk_id VARCHAR(36) PRIMARY KEY,
    document_id VARCHAR(36) NOT NULL,
    content TEXT NOT NULL,
    embedding_id VARCHAR(255),
    metadata JSON,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (document_id) REFERENCES documents(document_id)
);
```

Solutions

```
CREATE TABLE solutions (
    solution_id VARCHAR(36) PRIMARY KEY,
    ticket_id VARCHAR(36) NOT NULL,
    content TEXT NOT NULL,
    agent_id VARCHAR(36),
    used_rag BOOLEAN DEFAULT FALSE,
    auto_resolved BOOLEAN DEFAULT FALSE,
    feedback_score INT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (ticket_id) REFERENCES tickets(ticket_id),
    FOREIGN KEY (agent_id) REFERENCES agents(agent_id)
);
```

Indexing Strategy

- Appropriate indexes on foreign keys
- Text indexes for search functionality
- Performance-optimized indexing for queue operations

Message Queue Architecture

RabbitMQ Configuration

Exchanges

- **ticket.incoming**: Fanout exchange for new tickets

- **ticket.classification**: Direct exchange for classified tickets
- **ticket.assignment**: Direct exchange for ticket assignment
- **notification**: Topic exchange for system notifications

Queues

- **email.incoming**: New email messages
- **classification.queue**: Tickets awaiting classification
- **agent.{skill_id}**: Skill-specific queues
- **direct.resolution**: Low criticality tickets for auto-resolution
- **notification.{type}**: Type-specific notification queues

Routing Logic

- New tickets → classification queue
- Classified tickets → skill-specific queues based on category
- Agent availability updates → queue rebalancing
- Priority tickets → priority queues with higher processing preference

Load Balancing Strategy

- Round-robin distribution within skill groups
- Agent load factor consideration
- SLA-based prioritization
- Backpressure mechanisms for system protection

RAG Implementation

Document Processing Pipeline

1. **Acquisition**
    - Crawl4AI for web content scraping
    - File upload API for direct document addition
    - Scheduled crawling of specified resources
2. **Processing**
    - Text extraction and cleaning
    - Metadata generation
    - Chunking strategy (by paragraph, fixed token size, semantic units)
3. **Embedding Generation**
    - Embedding model selection
    - Batch processing approach
    - Update strategy for changed documents
4. **Storage**
    - Vector DB schema
    - Metadata fields and filtering capabilities
    - Versioning approach

Query Processing

1. **Query Understanding**
    - Query parsing
    - Intent identification
    - Key concept extraction
2. **Retrieval**
    - Vector similarity search
    - Metadata filtering
    - Top-k selection
3. **Reranking**
    - Cross-encoder implementation
    - Context-aware relevance scoring
    - Result diversification
4. **Response Generation**
    - Template-based for simple queries
    - LLM-assisted for complex queries
    - Source citation and confidence scoring

Vector Database Configuration

- **Database Choice**: [Qdrant/Pinecone/Chroma]

- **Collection Structure**:
    - Namespaces for different document types
    - Metadata schema for filtering
    - Index configuration for performance
- **Scaling Strategy**:
    - Horizontal scaling approach
    - Sharding strategy if applicable
    - Replication for redundancy

LLM Integration

Groq Integration

- **API Configuration**:
    - Authentication method
    - Rate limiting consideration
    - Fallback strategy
    - Model selection (Mixtral, Llama, etc.)
- **Classification Pipeline**:
    - Prompt templates for ticket classification
    - Category mapping logic
    - Confidence thresholds
    - Human-in-the-loop for low confidence cases
- **Criticality Assessment**:
    - Severity determination prompts
    - Urgency factors consideration
    - Business impact evaluation
    - Priority score calculation
- **Knowledge Augmentation**:
    - RAG integration points
    - Context window management
    - Answer synthesis approach

Future In-house Model

- **Training Data Collection**:
    - Query-solution pairs storage
    - Classification examples collection
    - Performance tracking for improvements
    - Annotation pipeline for training data
- **Model Architecture Planning**:
    - Fine-tuning approach
    - Model size considerations
    - Specialized classification heads
    - Deployment infrastructure requirements

Monitoring & Observability

Grafana Dashboards

1. **System Health Dashboard**
    - API endpoint response times
    - Queue depths and processing rates
    - Database performance metrics
    - Error rates and types
2. **Business Metrics Dashboard**
    - Ticket volume by channel
    - Resolution times by category
    - Agent performance metrics
    - Customer satisfaction scores
3. **ML/AI Performance Dashboard**
    - Classification accuracy
    - RAG relevance scores
    - LLM response times
    - Auto-resolution success rates

Alerting Configuration

- Critical system failures
- Queue depth thresholds
- Response time degradation
- Classification confidence thresholds
- Agent availability issues

Logging Strategy

- Structured logging
- Log levels and retention policy
- Request ID tracing
- PII handling in logs

Security Considerations

Authentication & Authorization

- JWT implementation details
- Role-based access control matrix
- API key management
- Session handling

Data Protection

- PII identification and handling
- Encryption strategy (at rest and in transit)
- Data retention policies
- Backup procedures

Compliance Requirements

- GDPR considerations
- SOC 2 compliance path
- Industry-specific requirements

Security Testing

- Regular penetration testing
- Dependency vulnerability scanning
- Code security reviews
- Authentication bypass testing

Testing Strategy

Unit Testing

- Test framework: Pytest
- Coverage targets: 80%+ for core functionality
- Mocking strategy for external dependencies
- CI integration

Integration Testing

- API endpoint testing
- Database interaction testing
- Queue processing verification
- External service mocking

Performance Testing

- Load testing methodology
- Benchmarking key operations
- Scalability testing approach
- Stress testing scenarios

Acceptance Testing

- Ticket classification accuracy
- End-to-end flow validation
- Agent experience verification
- Customer journey simulation

Deployment Pipeline

Environment Strategy

- Development environment
- Testing/QA environment
- Staging environment
- Production environment

CI/CD Pipeline

- GitHub Actions/Jenkins configuration
- Automated testing gates
- Deployment approval process
- Rollback procedures

Infrastructure as Code

- Terraform for AWS resources
- Docker Compose for local development
- Kubernetes manifests if applicable
- Environment variable management

Project Timeline

Key Milestones

- **Week 4**: Email integration and basic backend completed
- **Week 8**: Classification and routing system operational
- **Week 12**: RAG system implemented and integrated
- **Week 16**: Data collection pipeline complete
- **Week 20**: System optimization and testing completed
- **Week 22**: Production deployment
- **Week 24**: Project retrospective and planning for next phases

Gantt Chart

[Gantt chart would be inserted here in a real document]

Resource Requirements

Development Team

- 1 Backend Lead Developer (You)
- 1-2 Backend Developers
- 1 AI/ML Specialist (part-time)
- 1 DevOps Engineer (part-time)
- 1 QA Engineer

Infrastructure Requirements

- AWS account with appropriate permissions
- Development environments for team members
- CI/CD pipeline setup
- Test data generation tools
- Monitoring setup

External Dependencies

- Groq API credits
- Vector database service subscription
- Email sending service (if not using AWS SES)
- Testing tools and environments