Banking AI Chatbot Documentation

Table of Contents

System Overview

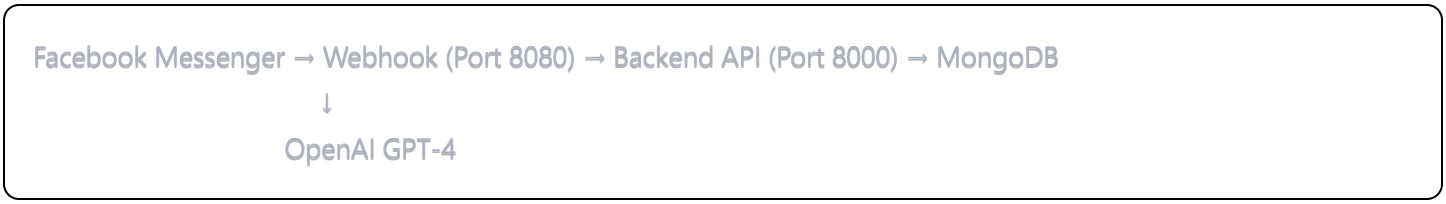
A sophisticated AI-powered banking chatbot system that provides contextual conversation capabilities for banking operations. The system integrates with Facebook Messenger and uses OpenAI's GPT-4 for natural language understanding and MongoDB for data storage.

Key Capabilities

- **Contextual Conversations**: Remembers previous queries and understands references like "from this" and "out of that"
- **Query Completeness Analysis**: Asks for clarification when information is missing
- **Multi-Intent Support**: Balance inquiries, transaction history, spending analysis, money transfers
- **Dynamic MongoDB Pipeline Generation**: Creates complex database queries from natural language
- **Real-time Processing**: Instant responses with comprehensive logging

Architecture

High-Level Architecture



Component Breakdown

1. **Webhook Service (webhook.py)**

- **Port**: 8080
- **Purpose**: Handles Facebook Messenger integration
- **Responsibilities**:
  - Receives messages from Facebook
  - User verification
  - Rate limiting
  - API calls to backend
  - Message sending to users

2. **Backend API Service (api_routes.py + app.py)**

- **Port**: 8000
- **Purpose**: Core business logic and AI processing
- **Responsibilities**:
  - AI query processing
  - Context management
  - MongoDB operations
  - Pipeline generation and execution

3. **AI Agent (ai_agent.py)**

- **Purpose**: Intelligent query processing with context awareness
- **Responsibilities**:
  - Natural language understanding
  - Context resolution
  - Intent classification
  - MongoDB pipeline generation
  - Response formatting

4. Database Layer (MongoDB)

- **Collections**:
  - `users`: User profiles and credentials
  - `bank_statements`: Transaction data

Features

1. Contextual Conversation Management

Context Storage

Each user has a persistent conversation context containing:

- Last query and intent
- Previous filters and pipeline
- Last result and response
- Timestamp for context expiry

Context Reference Detection

Automatically detects phrases like:

- "from this", "from that"
- "out of this", "out of that"
- "from these", "from those"
- "of this", "of that"
- "in this", "in that"

Context Resolution

Combines current query with previous context to create complete, standalone queries.

Example:

```
User: "How much did I spend in June?"
Bot: "You spent $1,234.56 in June"
User: "from this how much on groceries?"
Bot: Resolves to "How much did I spend on groceries in June?"
```

2. Query Completeness Analysis

Automatic Validation

Before processing any query, the system checks for:

- Time periods (for spending/transaction queries)
- Categories (for spending analysis)
- Limits (for transaction lists)
- Required parameters (for transfers)

Smart Clarification

Asks specific clarification questions when information is missing:

- "Could you please specify the time period? For example: 'in June', 'last month', 'this year', or 'in the last 30 days'."
- "Could you please specify how many transactions or what time period? For example: 'last 10 transactions', 'transactions in June', or 'recent transactions'."

3. Multi-Intent Support

Supported Intents

1. **Balance Inquiry**
   - Keywords: balance, money, amount, funds, account, cash
   - Example: "What is my balance?"
2. **Transaction History**
   - Keywords: transaction, history, recent, last, show, list, activities
   - Example: "Show me my last 10 transactions"
3. **Spending Analysis**
   - Keywords: spend, spent, spending, expense, cost, paid, purchase
   - Example: "How much did I spend last month?"
4. **Category Spending**
   - Spending analysis with specific categories
   - Example: "How much did I spend on groceries in June?"
5. **Money Transfer**
   - Keywords: transfer, send, pay, wire, remit, move money
   - Example: "Transfer 500 USD to John"

6. **General**
    - Fallback for unrecognized queries
    - Provides help and available options

4. Dynamic MongoDB Pipeline Generation

The system automatically generates complex MongoDB aggregation pipelines based on user queries:

Pipeline Components

- **$match**: Filters by account, dates, categories, amounts
- **$group**: Aggregates totals by currency
- **$sort**: Orders by date or other criteria
- **$limit**: Restricts result count
- **$project**: Selects specific fields

Example Pipeline Generation

Query: "How much did I spend on groceries in June 2024?"

Generated Pipeline:

```json
```

```
[
  {
    "$match": {
      "account_number": "1001",
      "type": "debit",
      "category": {"$regex": "groceries", "$options": "i"},
      "date": {
        "$gte": {"$date": "2024-06-01T00:00:00Z"},
        "$lte": {"$date": "2024-06-30T23:59:59Z"}
      }
    }
  },
  {
    "$group": {
      "_id": null,
      "total_usd": {"$sum": "$amount_usd"},
      "total_pkr": {"$sum": "$amount_pkr"}
    }
  }
]
```

API Endpoints

Authentication Endpoints

POST /verify

Verify user credentials for chatbot access.

Request:

```
json

{
  "account_number": "1001",
  "dob": "1990-01-01",
  "mother_name": "Jane Doe",
  "place_of_birth": "New York"
}
```

Response:

```
json
```

```
{
  "status": "success",
  "user": {
    "first_name": "John",
    "account_number": "1001"
  }
}
```


Core Banking Endpoints

POST /user_balance

Get user's current account balance.

Request:


```
json

{
  "account_number": "1001"
}
```


Response:


```
json

{
  "status": "success",
  "user": {
    "first_name": "John",
    "last_name": "Doe",
    "account_number": "1001",
    "current_balance_usd": 5000.50,
    "current_balance_pkr": 375000.00
  }
}
```


POST /execute_pipeline

Execute custom MongoDB aggregation pipelines.

Request:


```
json
```

```
{
  "account_number": "1001",
  "pipeline": [
    {"$match": {"account_number": "1001", "type": "debit"}},
    {"$group": {"_id": null, "total": {"$sum": "$amount_usd"}}}
  ]
}
```

Response:

```
json

{
  "status": "success",
  "data": [{"_id": null, "total": 1234.56}],
  "count": 1
}
```

POST /transfer_money

Process money transfers between accounts.

Request:

```
json

{
  "from_account": "1001",
  "to_recipient": "John Smith",
  "amount": 500.00,
  "currency": "USD"
}
```

Response:

```
json
```

```
{
  "status": "success",
  "message": "Successfully transferred 500.0 USD to John Smith",
  "transaction_id": "64f8a9b2c3d4e5f6g7h8i9j0",
  "new_balance_usd": 4500.50,
  "new_balance_pkr": 375000.00,
  "transfer_details": {
    "amount": 500.0,
    "currency": "USD",
    "recipient": "John Smith",
    "timestamp": "2024-07-19T10:30:00"
  }
}
```


AI Processing Endpoints

POST /process_query

Process natural language banking queries with contextual awareness.

Request:

json


```
{
  "user_message": "How much did I spend on groceries last month?",
  "account_number": "1001",
  "first_name": "John"
}
```


Response:

json


```
{
  "status": "success",
  "response": "Last month, you spent a total of $245.67 on groceries."
}
```


Utility Endpoints

GET /health

Health check for service monitoring.

Response:

```json
{
  "status": "healthy",
  "timestamp": "2024-07-19T10:30:00",
  "service": "banking_ai_backend"
}
```

POST /debug_pipeline

Debug MongoDB pipeline processing.

Request:

```json
{
  "account_number": "1001",
  "pipeline": [{"$match": {"account_number": "1001"}}]
}
```

Response:

```json
{
  "status": "success",
  "original_pipeline": [{"$match": {"account_number": "1001"}}],
  "processed_pipeline": [{"$match": {"account_number": "1001"}}]
}
```

Context Management

Context Storage Structure

```python
```

```
{
  "last_query": "How much did I spend in June?",
  "last_intent": "spending_analysis",
  "last_filters": {
    "month": "june",
    "year": 2024,
    "transaction_type": "debit"
  },
  "last_pipeline": [...],
  "last_result": {...},
  "last_response": "You spent $1,234.56 in June",
  "timestamp": "2024-07-19T10:30:00"
}
```


Context Lifecycle

1. **Creation**: New context created on first successful query
2. **Update**: Context updated after each successful query processing
3. **Expiry**: Context expires after 10 minutes of inactivity
4. **Cleanup**: Periodic cleanup prevents memory leaks

Context Resolution Process

1. Detect contextual reference phrases
2. Validate context exists and is recent
3. Combine current query with previous context
4. Generate resolved, standalone query
5. Process resolved query normally

Query Processing Flow

Complete Query Processing Pipeline

mermaid

```
graph TD
    A[User Message] --> B[Contextual Analysis]
    B --> C{Has Context Reference?}
    C -->|Yes| D[Check Context Exists]
    C -->|No| E[Completeness Analysis]
    D --> F{Context Valid?}
    F -->|Yes| G[Resolve with Context]
    F -->|No| H[Request Complete Info]
    E --> I{Query Complete?}
    I -->|No| H
    I -->|Yes| J[Process Query]
    G --> J
    J --> K[Extract Filters]
    K --> L[Detect Intent]
    L --> M[Generate Pipeline]
    M --> N[Execute Pipeline]
    N --> O[Format Response]
    O --> P[Update Context]
    P --> Q[Return Response]
    H --> Q
```

Step-by-Step Processing

1. **Message Reception**
   - Webhook receives message from Facebook
   - Rate limiting check
   - User authentication validation
2. **Contextual Analysis**
   - Check for context reference phrases
   - Validate previous context exists and is recent
   - Resolve contextual queries if applicable
3. **Completeness Analysis**
   - Analyze if query has all required information
   - Check for missing time periods, categories, limits
   - Request clarification if incomplete
4. **Query Processing**
   - Extract filters using LLM

- Classify intent (balance, transactions, spending, transfer)
- Generate MongoDB aggregation pipeline
- Validate pipeline structure

5. **Execution**
   - Execute pipeline against MongoDB
   - Handle different intent types appropriately
   - Format response using LLM
6. **Context Update**
   - Store query, intent, filters, pipeline, result
   - Update timestamp for context expiry
   - Return formatted response to user

Use Cases & Examples

1. Simple Balance Inquiry

Conversation:

```
User: "What is my balance?"
Bot: "Your current balance is $5,000.50 USD and ₨375,000.00 PKR."
```

Technical Flow:

- Intent: balance_inquiry
- Pipeline: Direct balance query
- No context needed

2. Transaction History with Context

Conversation:

```
User: "Show me my last 5 transactions"
Bot: "Here are your last 5 transactions: [transaction list]"
User: "from these which ones are over $100?"
Bot: "From your last 5 transactions, here are the ones over $100: [filtered list]"
```

Technical Flow:

- First query: intent=transaction_history, limit=5
- Second query: Context detected, resolved to "show me transactions from last 5 that are over $100"
- Modified pipeline with amount filter

3. Spending Analysis with Follow-up

Conversation:

```
User: "How much did I spend in June?"
Bot: "In June, you spent a total of $1,234.56 USD and ₨95,000 PKR."
User: "from this how much on groceries?"
Bot: "From your June spending, you spent $245.67 on groceries."
```

Technical Flow:

- First query: intent=spending_analysis, month=june, year=2024
- Context stored with date filters
- Second query: Context resolved to "how much did I spend on groceries in June?"
- Pipeline updated with category filter

4. Incomplete Query Handling

Conversation:

```
User: "How much did I spend on groceries?"
Bot: "Could you please specify the time period? For example: 'in June', 'last month', 'this year', or 'in the last 30 days'."
User: "last month"
Bot: "Last month, you spent $245.67 on groceries."
```

Technical Flow:

- First query: Completeness analysis detects missing time period
- Clarification requested
- Second query: Combined with previous to form complete query

5. Money Transfer

Conversation:

```
User: "Transfer 500 USD to John Smith"
Bot: "Successfully transferred $500.00 USD to John Smith. Your new balance is $4,500.50 USD and ₨375,000.00 PKR.
Transaction ID: 64f8a9b2c3d4e5f6g7h8i9j0"
```

Technical Flow:

- Intent: transfer_money
- Extract amount, currency, recipient
- Validate sufficient balance
- Create transaction record
- Update balance

6. Complex Contextual Chain

Conversation:

```
User: "Show me transactions from first week of June"
Bot: "Here are your transactions from June 1-7: [transaction list]"
User: "from these how much on utilities?"
Bot: "From your June 1-7 transactions, you spent $85.50 on utilities."
User: "what about food?"
Bot: "From the same period (June 1-7), you spent $156.78 on food."
```

Technical Flow:

- Context maintained across multiple queries
- Each follow-up resolves with previous date range
- Different category filters applied to same base query

Setup & Installation

Prerequisites

- Python 3.8+
- MongoDB
- OpenAI API Key
- Facebook Developer Account
- Facebook Page Access Token

Environment Setup

1. **Clone Repository**

```bash
git clone <repository-url>
cd banking-ai-chatbot
```

2. **Install Dependencies**

```bash
pip install -r requirements.txt
```

3. **Environment Variables** Create `.env` file:

```env
OPENAI_API_KEY=your_openai_api_key
MONGODB_URI=mongodb://localhost:27017/banking_db
PAGE_ACCESS_TOKEN=your_facebook_page_token
VERIFY_TOKEN=your_webhook_verify_token
```

4. **MongoDB Setup**

```bash
# Start MongoDB
mongod

# Create database and collections
mongo
use banking_db
db.users.createIndex({"account_number": 1})
db.bank_statements.createIndex({"account_number": 1, "date": -1})
```

5. **Facebook Webhook Setup**

- Create Facebook App
- Setup Messenger webhook pointing to your server
- Subscribe to page events

- Set webhook URL: `https://yourdomain.com/webhook`

Running the Services

1. **Start Backend API (Port 8000)**

```bash
python app.py
```

2. **Start Webhook Service (Port 8080)**

```bash
python webhook.py
```

3. **Verify Services**

```bash
# Check backend health
curl http://localhost:8000/health

# Check webhook health
curl http://localhost:8080/health
```

Configuration

AI Agent Configuration

LLM Settings

```python
llm = ChatOpenAI(
    model="gpt-4o",
    api_key=os.getenv("OPENAI_API_KEY"),
    temperature=0.1  # Low temperature for consistent responses
)
```

Context Settings

```python
```

```python
CONTEXT_EXPIRY_MINUTES = 10  # Context expires after 10 minutes
MAX_CONTEXT_HISTORY = 5      # Keep last 5 query contexts
```

Rate Limiting

```python
MESSAGE_RATE_LIMIT = 2  # Seconds between messages per user
API_TIMEOUT = 30        # Seconds for AI processing timeout
```

MongoDB Configuration

Collections Schema

Users Collection:

```json
{
  "_id": ObjectId,
  "account_number": "1001",
  "first_name": "John",
  "last_name": "Doe",
  "dob": "1990-01-01",
  "mother_name": "Jane Doe",
  "place_of_birth": "New York",
  "current_balance_usd": 5000.50,
  "current_balance_pkr": 375000.00
}
```

Bank Statements Collection:

```json
```

```
{
  "_id": ObjectId,
  "account_number": "1001",
  "date": ISODate("2024-07-19T10:30:00Z"),
  "type": "debit",
  "description": "Grocery Store Purchase",
  "category": "Groceries",
  "amount_usd": 45.67,
  "amount_pkr": 0,
  "balance_usd": 4954.83,
  "balance_pkr": 375000.00
}
```

Logging Configuration

Log Levels

- INFO: Normal operations, query processing
- WARNING: Incomplete queries, validation issues
- ERROR: Processing failures, API errors
- DEBUG: Detailed pipeline generation, context resolution

Log Format

```python
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(name)s - %(message)s'
)
```

Error Handling

Error Categories

1. **User Input Errors**

- Incomplete queries → Request clarification
- Invalid formats → Provide examples
- Missing context → Ask for complete information

2. **API Errors**

- Timeout errors → "Request timed out, please try simpler query"
- Rate limiting → "Please wait before sending another message"
- Authentication failures → "Please verify your identity"

3. **Database Errors**

- Connection failures → "Database temporarily unavailable"
- Query errors → "Error processing request, please try again"
- Data corruption → Log error, return generic message

4. **AI Processing Errors**

- LLM failures → Fallback to rule-based processing
- Pipeline generation errors → Use fallback pipeline
- Context resolution errors → Request complete query

Error Response Structure

```json
{
  "status": "error",
  "response": "User-friendly error message",
  "error": "Technical error details (logged only)"
}
```

Recovery Mechanisms

Graceful Degradation

1. LLM fails → Rule-based processing
2. Context resolution fails → Request complete query
3. Pipeline generation fails → Use fallback pipeline
4. Database fails → Cache responses temporarily

Retry Logic

- API calls: 3 retries with exponential backoff
- Database operations: 2 retries with 1-second delay
- Context resolution: Single attempt, fallback to complete query

Monitoring & Logging

Key Metrics to Monitor

Performance Metrics

- Query processing time
- API response time
- Database query execution time
- Context resolution success rate

Business Metrics

- Total queries processed
- Intent distribution
- Context usage rate
- Error rates by type

System Metrics

- Memory usage (context storage)
- Database connection pool
- API rate limiting hits
- Facebook webhook reliability

Logging Structure

Query Processing Logs

```json
{
  "timestamp": "2024-07-19T10:30:00Z",
  "action": "process_query",
  "account_number": "1001",
  "user_message": "How much did I spend on groceries?",
  "intent": "spending_analysis",
  "has_context": false,
  "processing_time_ms": 1250,
  "status": "success"
}
```

Context Resolution Logs

```json
{
  "timestamp": "2024-07-19T10:30:00Z",
  "action": "resolve_contextual_query",
  "original_query": "from this how much on groceries",
  "resolved_query": "how much did I spend on groceries in June",
  "context_age_seconds": 45,
  "status": "success"
}
```

Error Logs

```json
{
  "timestamp": "2024-07-19T10:30:00Z",
  "action": "pipeline_generation_error",
  "account_number": "1001",
  "error_type": "json_parse_error",
  "error_message": "Could not parse LLM response",
  "fallback_used": true
}
```

Monitoring Dashboard Recommendations

Real-time Metrics

- Active conversations
- Query processing queue
- Error rate (last 5 minutes)
- Average response time

Daily Reports

- Total queries by intent
- Context usage statistics
- Top error categories
- User engagement metrics

Alerts

- Error rate > 5%
- Response time > 10 seconds
- Database connection failures
- OpenAI API quota exhaustion

Troubleshooting

Common Issues

1. Context Not Working

Symptoms: Follow-up queries not understanding context **Causes:**

- Context expired (>10 minutes)
- Context storage failure
- Reference phrase not detected

Solutions:

```python
# Check context storage
context = ai_agent.get_user_context(account_number)
print(f"Last query: {context.last_query}")
print(f"Timestamp: {context.timestamp}")

# Check reference phrase detection
context_phrases = ["from this", "from that", "out of this"]
has_reference = any(phrase in user_message.lower() for phrase in context_phrases)
```

2. Pipeline Generation Failures

Symptoms: "Error processing query" messages **Causes:**

- LLM response parsing errors
- Invalid MongoDB pipeline structure
- Date formatting issues

Solutions:

```python
# Enable debug logging
logger.setLevel(logging.DEBUG)

# Test pipeline generation
filters = ai_agent.extract_filters_with_llm(user_message)
pipeline = ai_agent.generate_pipeline_from_filters(filters, intent, account_number)

# Validate pipeline manually
import jsonschema
jsonschema.validate(pipeline, PIPELINE_SCHEMA)
```

3. API Timeout Issues

Symptoms: "Request timed out" errors **Causes:**

- Complex queries taking >30 seconds
- High OpenAI API latency
- Database query performance issues

Solutions:

```python
# Increase timeout
async with httpx.AsyncClient(timeout=60.0) as client:
    # API call

# Optimize database queries
db.bank_statements.createIndex({"account_number": 1, "date": -1, "category": 1})

# Add query complexity limits
if len(user_message) > 500:
    return "Please use simpler, shorter queries"
```

4. Facebook Webhook Issues

Symptoms: Messages not received or responses not sent **Causes:**

- Webhook verification failures
- Invalid access tokens

- Rate limiting by Facebook

Solutions:

```bash
# Test webhook endpoint
curl -X GET "https://yourdomain.com/webhook?hub.mode=subscribe&hub.challenge=test&hub.verify_token=your_tok

# Check access token
curl -X GET "https://graph.facebook.com/me?access_token=YOUR_TOKEN"

# Monitor webhook logs
tail -f webhook.log | grep "webhook"
```

Debug Mode

Enable Comprehensive Logging

```python
# Add to ai_agent.py
DEBUG_MODE = os.getenv("DEBUG_MODE", "false").lower() == "true"

if DEBUG_MODE:
    logging.getLogger().setLevel(logging.DEBUG)
    # Log all LLM requests/responses
    # Log all MongoDB queries
    # Log all context operations
```

Test Individual Components

```python
```

```
# Test context resolution
def test_context_resolution():
    agent = BankingAIAgent()
    # Set up test context
    # Test resolution with various queries

# Test pipeline generation
def test_pipeline_generation():
    agent = BankingAIAgent()
    # Test with various filter combinations
    # Validate generated pipelines

# Test intent classification
def test_intent_classification():
    agent = BankingAIAgent()
    # Test with various query types
    # Verify intent accuracy
```

Performance Optimization

Database Optimization

javascript

```
// MongoDB indexes
db.bank_statements.createIndex({"account_number": 1, "date": -1})
db.bank_statements.createIndex({"account_number": 1, "category": 1, "date": -1})
db.bank_statements.createIndex({"account_number": 1, "type": 1, "date": -1})

// Query optimization
db.bank_statements.aggregate([
  {"$match": {"account_number": "1001"}},
  {"$sort": {"date": -1}},
  {"$limit": 1000}  // Add reasonable limits
])
```

Memory Management

python

```python
# Context cleanup
def cleanup_old_contexts():
    current_time = datetime.now()
    contexts_to_remove = []

    for account_number, context in ai_agent.user_contexts.items():
        if context.timestamp and (current_time - context.timestamp).seconds > 3600:
            contexts_to_remove.append(account_number)

    for account_number in contexts_to_remove:
        del ai_agent.user_contexts[account_number]
```

API Optimization

```python
# Connection pooling
httpx_client = httpx.AsyncClient(
    timeout=30.0,
    limits=httpx.Limits(max_keepalive_connections=20, max_connections=100)
)

# Response caching for identical queries
from functools import lru_cache

@lru_cache(maxsize=1000)
def cached_query_processing(query_hash, account_number):
    # Cache results for identical queries
    pass
```

Contributing

Code Style

- Follow PEP 8 for Python code
- Use type hints for function parameters and returns
- Add docstrings for all public methods
- Comprehensive logging for debugging

Testing

- Unit tests for all AI agent methods
- Integration tests for API endpoints
- End-to-end tests for complete conversation flows
- Performance tests for complex queries

Documentation

- Update this documentation for any new features
- Add examples for new use cases
- Document any new configuration options
- Keep troubleshooting section current

Last Updated: July 19, 2024 **Version:** 2.0 **Authors:** Banking AI Team