Knowledge-Powered Q&A and Action Bot (MCP)

MOHAN CHANDRA S S

Github Link - https://github.com/mohanchandrass/QA-MCP-Server

Hackathon Submission Documentation

Overview

This project implements a **Knowledge-Powered Q&A and Action Bot** using the **Model Context Protocol (MCP)**.
The system is designed to:

- Answer user queries using a configurable knowledge base
- Resolve user intent deterministically from configuration
- Allow an LLM to handle the majority of interactions
- Escalate to human agents **only when required**
- Trigger actions (e.g., ticket creation) in a controlled, auditable manner
- Support **industry switching** by swapping configuration files only

The architecture strictly separates **decision logic** from **language generation**, ensuring enterprise safety and predictability.

Architecture Summary

The system consists of two components:

1. **MCP Server (Dockerized)**
   - Exposes knowledge, persona, intents, and actions as MCP resources
   - Resolves intent using deterministic rules
   - Executes actions only when explicitly invoked
2. **MCP Client (LLM-Integrated)**
   - Queries MCP resources
   - Uses an LLM (Gemini) for natural language responses
   - Enforces escalation and sampling policies from configuration
   - Produces full trace logs for observability

```
User
 └─> MCP Client (LLM + Policy Enforcement)
     ├─> MCP Resource: knowledge://search
     ├─> MCP Tool: resolve_intent
     ├─> Decision Logic (from actions.yaml)
     └─> MCP Tool: create_ticket (if required)
         └─> MCP Server
```

Execution Guide

Prerequisites

- Python 3.11+
- Docker
- Gemini API key

Execution Guide (Copy–Paste Ready)

repo structure:

```
qa-mcp/
├── mcp-client/
└── mcp-server/
```

This system consists of:

- **MCP Server** (knowledge + intent + action engine)
- **MCP Client** (LLM-powered conversational interface)

Step 0: Navigate to Project Root

```
cd qa-mcp
```

All subsequent commands assume you start from this directory.

Step 1: Install MCP Client Dependencies (Required)

```
cd mcp-client
pip install -r requirements.txt
```

Why this is required

- Enables Gemini API integration
- Enables MCP client ↔ server communication

Step 2: Set LLM API Key

```
export GEMINI_API_KEY=your_api_key_here
```

This key is used **only by the client**.
The MCP server does **not** require an LLM key.

Step 3: Run the MCP Server

Return to the project root first:

```
cd ..
```

Option A: Run via Docker (Recommended)

Navigate to server directory

```
cd mcp-server
```

Build the Docker image

```
docker build -t qa-mcp-server .
```

Run the container with mounted resources

```
docker run -p 8000:8000 \
  -v $(pwd)/config:/app/config \
  -v $(pwd)/data:/app/data \
  qa-mcp-server
```

Why resource mounting is mandatory

- **/app/config** → persona, intents, actions (industry behavior)
- **/app/data** → knowledge base
- Enables **hot-swappable industry configuration** without rebuilding the image

Server endpoints

MCP Endpoint: ***http://localhost:8000/mcp***

Health Check: ***http://localhost:8000/health***

Option B: Run via Python (Local Development)

```
cd mcp-server
pip install -r requirements.txt
python qa_mcp_server.py
```

Ensure the following directories exist:

mcp-server/config/

mcp-server/data/

Step 4: Run the MCP Client

Open a **new terminal** (or a new tab).

```
cd qa-mcp/mcp-client
python qa_mcp_client.py
```

You can now interact with the system via the terminal.

Runtime Flow (What Happens After Startup)

1. MCP Server loads:
    - Persona configuration
    - Intent taxonomy
    - Action and escalation policies
    - Knowledge base
2. MCP Client:
    - Reads MCP resources (**/knowledge, /persona, /intents, /actions**)
    - Sends user queries
    - Uses Gemini to generate responses
    - Enforces escalation rules deterministically
    - Triggers MCP action tools when required

Configuration-Driven Design

Industry Switching

Industry behavior is controlled entirely via configuration files:

- **persona.yaml** → industry, tone, search behavior
- **intents.yaml** → intent taxonomy and triggers
- **actions.yaml** → escalation, sampling, and action policy

To switch industries (e.g., banking → healthcare), **only the config files change**.
No code changes are required.

Decision Logic (How Escalation Works)

Core Principle

- **The LLM never decides on escalation.**
- All escalation decisions are deterministic and configuration-driven.

Escalation Triggers

Escalation occurs when **any** of the following conditions are met:

1. **Explicit User Request**
    - Phrases like:
        - "talk to an agent"
        - "connect me to a human"
        - "raise a ticket"
2. **High-Severity Intent**
    - Example: billing issues
    - Severity defined in **intents.yaml**
3. **Low Confidence / Fallback**
    - Intent resolution returns `confidence:` **fallback**
4. **Repeated Failure (Sampling Logic)**
    - User repeats issues beyond the configured threshold

All thresholds and rules are defined in **actions.yaml.**

What Happens on Escalation

1. Client detects escalation condition
2. Client explicitly calls MCP tool:

   create_ticket

3. MCP server executes the action
4. Session ends (enterprise-standard behavior)
5. Full trace log is emitted

Role of the LLM

The LLM is used **only** for:

- Natural language generation
- Persona-based tone and formatting
- Explaining known solutions from the knowledge base

The LLM **cannot**:

- Escalate on its own
- Execute actions
- Override policy
- Access internal logic

This ensures safety, predictability, and auditability.

Observability & Trace Logs

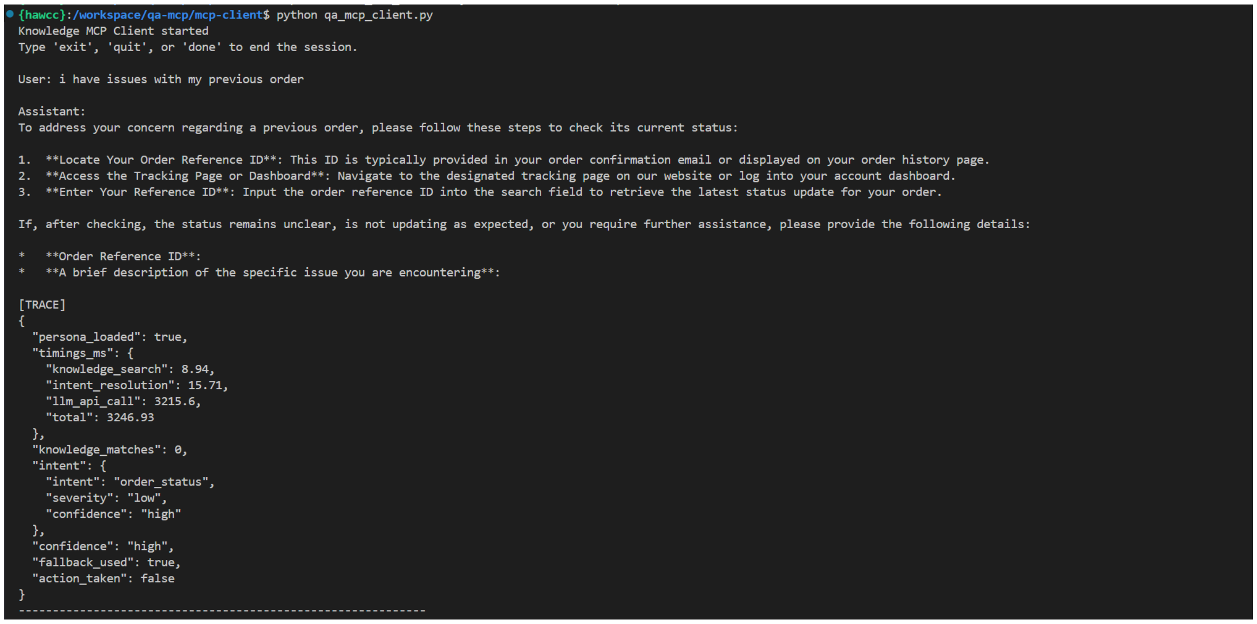
Every interaction produces a structured trace, including:

- Knowledge search time
- Intent resolution time
- LLM API latency
- Action execution time
- Escalation decision
- Confidence level

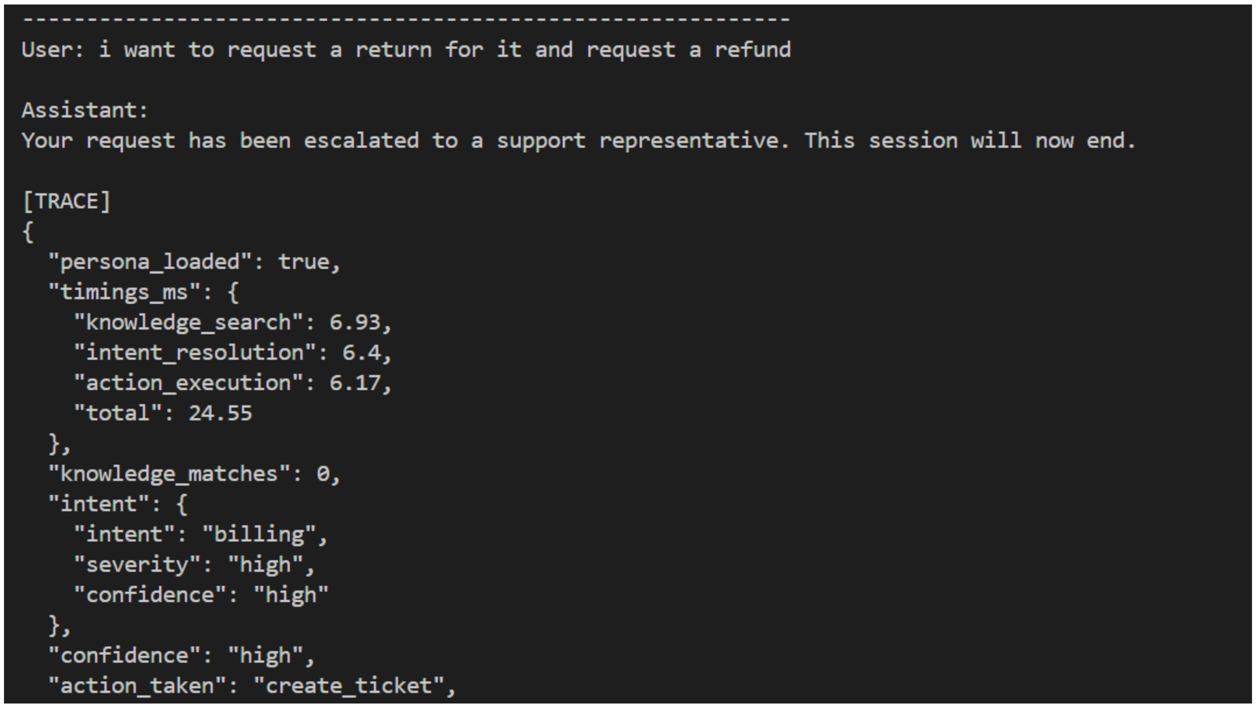
Example:

```
{
 "intent": {
  "intent": "billing",
  "severity": "high",
  "confidence": "high"
 },
 "action_taken": "create_ticket",
 "timings_ms": {
  "knowledge_search": 8.2,
  "intent_resolution": 6.9,
  "llm_api_call": 3200.5,
  "action_execution": 5.1,
  "total": 3225.4
 }
}
```

OUTPUT



```
● {hawcc}:/workspace/qa-mcp/mcp-client$ python qa_mcp_client.py
Knowledge MCP Client started
Type 'exit', 'quit', or 'done' to end the session.

User: i have issues with my previous order

Assistant:
To address your concern regarding a previous order, please follow these steps to check its current status:

1.  **Locate Your Order Reference ID**: This ID is typically provided in your order confirmation email or displayed on your order history page.
2.  **Access the Tracking Page or Dashboard**: Navigate to the designated tracking page on our website or log into your account dashboard.
3.  **Enter Your Reference ID**: Input the order reference ID into the search field to retrieve the latest status update for your order.

If, after checking, the status remains unclear, is not updating as expected, or you require further assistance, please provide the following details:

*   **Order Reference ID**:
*   **A brief description of the specific issue you are encountering**:

[TRACE]
{
  "persona_loaded": true,
  "timings_ms": {
    "knowledge_search": 8.94,
    "intent_resolution": 15.71,
    "llm_api_call": 3215.6,
    "total": 3246.93
  },
  "knowledge_matches": 0,
  "intent": {
    "intent": "order_status",
    "severity": "low",
    "confidence": "high"
  },
  "confidence": "high",
  "fallback_used": true,
  "action_taken": false
}
------------------------------------------------------------
```



```
------------------------------------------------------------
User: i want to request a return for it and request a refund

Assistant:
Your request has been escalated to a support representative. This session will now end.

[TRACE]
{
  "persona_loaded": true,
  "timings_ms": {
    "knowledge_search": 6.93,
    "intent_resolution": 6.4,
    "action_execution": 6.17,
    "total": 24.55
  },
  "knowledge_matches": 0,
  "intent": {
    "intent": "billing",
    "severity": "high",
    "confidence": "high"
  },
  "confidence": "high",
  "action_taken": "create_ticket",
```

Why This Design Works

- **LLM handles most cases** → fast, conversational support
- **Deterministic fallback** → guaranteed escalation when needed
- **Config-only industry switching** → scalable and reusable
- **MCP-native integration** → clean separation of concerns
- **Enterprise-safe** → no hidden logic, no hallucinated actions

Summary

This system demonstrates a **production-grade MCP architecture** where:

- Knowledge, intent, and actions are externalized
- LLMs enhance UX without compromising control
- Escalation is predictable and auditable
- Industry behavior is configurable, not hardcoded

The solution fully satisfies the **Knowledge-Powered Q&A and Action Bot** hackathon requirements.