SL-IT-AI: Intelligent IT Helpdesk Chatbot System

Version: 1.0.0

Date: 16-Jul-25

Author: Sameer Kamani

Company: Systems Limited

Table of Contents

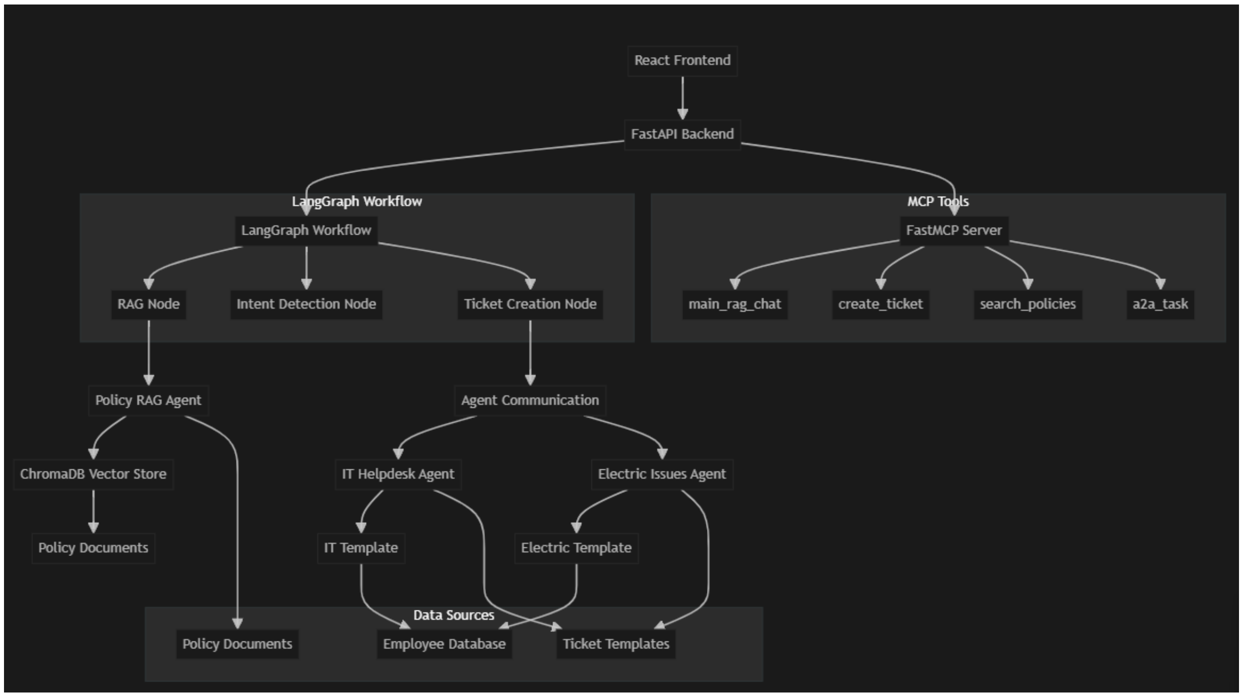
1. Project Overview

SL-IT-AI is an intelligent IT helpdesk chatbot system designed for Systems Limited. It leverages advanced AI (LLM), retrieval-augmented generation (RAG), and modular agent communication to provide:

- Automated answers to IT and policy-related queries
- Step-by-step troubleshooting guidance
- Seamless support ticket creation and routing to specialized agents (IT Helpdesk, Electric, etc.)
- Integration with company policies and employee data for context-aware responses

The system consists of a modern React frontend, a robust FastAPI backend, a modular MCP (Model Context Protocol) tool server, and a custom RAG engine for policy search.

2. System Architecture



Key Components:

- **Frontend:** React-based chat interface with authentication and seamless user experience.
- **Backend:** FastAPI server orchestrating conversation, ticketing, and agent routing using LangGraph workflows and RAG.
- **MCP Tool Server:** Modular tool server exposing core functions (chat, ticketing, policy search) via FastMCP.
- **Policy Index:** Chroma vectorstore with custom embeddings for fast, relevant policy retrieval.
- **Employee Data:** Used for autofill and context in ticket creation.

3. Backend Components

3.1 Configuration & Environment

- `config.py`: Centralizes environment variables, API keys, and file paths.
- Loads `.env` for secrets (OpenAI keys, etc.).
- Defines paths for templates, employee data, and policy index.
- Initializes logging and OpenAI clients.

3.2 API Layer

- `main.py` & `api_routes.py`: FastAPI application exposing endpoints for:
    - Chat (`/main_rag_chat`)
    - Ticket creation (`/create_ticket`)
    - MCP tool proxying (`/api/mcp/proxy`)
    - Health checks and tool listing
    - A2A (agent-to-agent) communication
- Handles CORS, static file serving, and request timeouts.

3.3 Agent Logic & Communication

- `agents.py`: Core business logic for:
    - Classifying issues (IT, Electric, Other) using LLM and MCP tools
    - Extracting user info and problem descriptions
    - Filling ticket fields with LLM, fuzzy matching, and employee data
    - Generating dynamic responses and error messages

    - Routing to specialized agents (A2A)
- `agent_communication.py`: Manages routing to IT Helpdesk or Electric agents, ensuring the right template and workflow are used.

3.4 LangGraph Workflow

- `langgraph_workflow.py`: Models the conversation as a state graph with nodes:
    - RAG Node: Provides solutions and policy context
    - Intent Detection Node: Classifies user intent (QA, create_ticket, confirmation, etc.)
    - Ticket Creation Node: Handles ticket creation, field extraction, and agent routing
- Ensures a robust, extensible workflow for all user interactions.

3.5 Retrieval-Augmented Generation (RAG)

- `policy_rag.py`: Implements a lightweight RAG agent:
    - Uses a deterministic embedding model for privacy and speed
    - Loads and indexes policy documents (text, PDF) using Chroma
    - Provides fast, relevant policy snippets for LLM context
    - Supports adding new documents and re-indexing

3.6 MCP Tools & Server

- `mcp_tools.py`: Exposes backend functions as modular tools via FastMCP:
    - Chat, ticket creation, user info extraction, policy search, A2A communication, etc.
    - Can be run as a standalone server (stdio transport)
    - Enables composable, reusable automation for agent workflows
- `.mcp.json`: Configuration for launching the MCP server and specifying available resources.

3.7 Data Models

- `models.py`: Defines all Pydantic schemas and TypedDicts for:
    - Chat and ticket requests/responses
    - Agent state for LangGraph
    - Event queue for compatibility

3.8 Policy Indexing

- `/policies_index`: Directory containing the Chroma vector index (`chroma.sqlite3` and binaries).
- Used by the RAG agent for fast similarity search.

4. Frontend Application

4.1 User Authentication

- `Login.js` & `firebaseConfig.js`: Implements user authentication using Firebase.
- Ensures only authorized users can access the chatbot.

4.2 Chat Interface

- `Chatbot.js`: Main chat UI component:
    - Manages conversation history, user input, and message display
    - Handles API calls to backend endpoints for chat and ticketing
    - Displays policy search results, troubleshooting steps, and ticket forms
    - Manages session state and error handling

4.3 API Integration

- Communicates with backend via REST API:
    - Sends user messages and receives structured responses (including tickets)
    - Handles ticket creation, status updates, and agent responses

4.4 User Experience & Styling

- `App.css`, `Login.css`, `index.css`: Custom styles for a modern, user-friendly interface.
- Responsive design for desktop and mobile.
- Clear chat bubbles, loading indicators, and error messages.

5. End-to-End Workflow

Example User Journey:

1. User logs in via the frontend (Firebase authentication).
2. User asks a question (e.g., "What are the company policies?"):
   a. Frontend sends message to backend.
   b. Backend uses RAG to retrieve relevant policy snippets and LLM to generate a response.
   c. Response is displayed in the chat.
3. User reports an issue (e.g., "My Wi-Fi isn't working"):
   a. Backend provides troubleshooting steps and asks if the user wants to create a ticket.
4. User requests ticket creation:
   a. Backend analyzes the conversation, extracts problem details, and classifies the issue.
   b. Routes to the appropriate agent (e.g., IT Helpdesk).
   c. Fills ticket fields using LLM, fuzzy matching, and employee data.
   d. Returns ticket details and confirmation to the user.
5. User receives updates and can continue the conversation or create additional tickets.

All steps are logged and traceable for audit and debugging.

6. Deployment & Operations

6.1 Prerequisites

- Python 3.10+
- Node.js (for frontend)
- All dependencies in `backend/requirements.txt`
- Environment variables in `.env` (AzureOpenAI keys, etc.)

6.2 Starting the System

- Use `start_servers.py` to launch both the FastAPI and MCP servers:

    - `python start_servers.py`
- FastAPI server runs on http://localhost:8000
- MCP server runs as a subprocess (stdio transport)
- Frontend connects to backend at http://localhost:8000

6.3 File Structure

- Backend: All Python code, configuration, and policy index
- Frontend: React app in `frontend/`
- Templates: Ticket templates in `Templates/`
- Employee Data: Employee info in `Employee Data/employees.jsonl`
- Policies: Policy documents in `policies/`

6.4 Monitoring & Logs

- All backend actions are logged with timestamps and debug info.
- Logs include workflow transitions, LLM calls, ticket creation, and errors.

7. Extensibility & Customization

- **Adding New Policies:** Place new `.txt` or `.pdf` files in the `policies/` directory and re-index if needed.
- **Adding New Agents:** Implement new agent logic in `agents.py` and update `agent_communication.py`.
- **Customizing Ticket Templates:** Edit or add new templates in the `Templates/` directory.
- **Expanding MCP Tools:** Add new tool functions in `mcp_tools.py` and update `.mcp.json` as needed.
- **Frontend Customization:** Update React components and styles for new features or branding.

8. Appendix: Key Files & Configuration

- `.env`: Environment variables (not committed to source control)
- `requirements.txt`: Python dependencies

- `/policies_index/`: Chroma vector index for RAG
- `Templates/`: Ticket templates (JSONL)
- `Employee Data/employees.jsonl`: Employee autofill data
- `frontend/src/`: Main React app source code

9. API Endpoint Reference

Endpoint	Method	Description	Request Body / Params	Response Example / Notes
/main_rag_chat	POST	Main chat endpoint (RAG, ticketing, etc.)	message, conversation_history, user_info	Chat/ticket response object
/create_ticket	POST	Create a support ticket	user_message, session_id, conversation_history	Ticket object
/api/mcp/proxy	POST	Proxy to MCP tool server	name, arguments	Tool-specific response
/api/mcp/status	GET	MCP server health/status	None	Status JSON
/api/mcp/tools	GET	List available MCP tools	None	List of tools
/electric/a2a/task	POST	Electric agent A2A endpoint	message, conversation_history, context	Electric ticket response
/session/{session_id}	GET	Get session info	session_id	Session details

/clear_session/{session_id}	GET	Clear session data	session_id	Success message

Note: For detailed request/response schemas, see `models.py` in the backend.

10. Security Considerations

- **Authentication:**

User authentication is enforced via Firebase on the frontend. Only authenticated users can access the chatbot.

- **Data Privacy:**

Employee data is stored locally in JSONL format and only used for ticket autofill. No sensitive data is exposed to the frontend or external APIs.

- **API Security:**

CORS is enabled and can be restricted for production. All backend endpoints validate input and handle errors gracefully.

- **Secrets Management:**

API keys and sensitive configuration are stored in `.env` and not committed to source control.

- **Logging:**

All actions are logged for audit and debugging, but logs should be monitored for sensitive information in production.

11. Error Handling & Troubleshooting

- **Common Errors:**
  - API timeouts (requests taking too long)
  - LLM or RAG service unavailability
  - Missing or malformed input data
  - MCP tool server not running
- **Troubleshooting Steps:**
  - Check backend logs for error messages and stack traces.
  - Ensure all required services (FastAPI, MCP server) are running.

    - Verify `.env` configuration and API keys.
    - Use `/api/mcp/status` and `/api/mcp/tools` to check MCP health.
    - For frontend issues, check browser console and network tab.
- **Support:**
    - For persistent issues, contact the development team or refer to inline code comments for debugging tips.

12. Versioning & Maintenance

- **Versioning:**

The project uses semantic versioning (e.g., 1.0.0). Update the version number on the cover page and in documentation with each release.

- **Dependencies:**

All Python dependencies are listed in `requirements.txt`. Use `pip install -r requirements.txt` to update dependencies.

- **Updating Policies or Templates:**

Add new policy documents to the `policies/` directory and re-index if needed. Update ticket templates in the `Templates/` directory as requirements change.

- **Maintaining the RAG Index:**

If new policies are added, ensure the Chroma index is updated or rebuilt.

- **Regular Backups:**

Backup the `policies_index/` and `Employee Data/` directories regularly.

13. Change Log / Revision History

Version	Date	Author	Description
1.0.0	16-07-24	Sameer Kamani	Initial release. Full documentation added.

Conclusion

SL-IT-AI is a robust, extensible, and intelligent IT helpdesk solution that combines the power of LLMs, RAG, and modular agent workflows. It provides a seamless user experience, automates ticketing, and ensures compliance with company policies. The architecture is designed for scalability, maintainability, and easy integration with new tools and data sources.