INCIDENT MANAGEMENT AUTO-RESOLUTION USING AI SELF-HEAL

Introduction

Traditional incident management approaches, although effective, can be time-consuming and resource-intensive. By synergizing the power of Artificial Intelligence (AI) and self-healing mechanisms, organisations can proactively detect, diagnose, and autonomously resolve incidents, thereby minimizing impact and maximizing operational efficiency.

Machine Learning Frameworks - Torch and Transformers

Programming Language - Python

Backend Framework - FastAPI

Frontend Framework - Streamlit

Database Management - SQLlite

CI/CD - Git

Containerisation - Docker

Infrastructure

Backend Infrastructure

1. **Code Changes:**
   You make changes to your application's code, which could include changes to the model, logic, or any other aspect of the application.

2. **Git Action Trigger:**
   Upon pushing your code changes to your Git repository (e.g., GitHub), a Git action that you've defined will be triggered automatically.
3. **Git Action Workflow:**
   Within your Git action workflow, you can define a series of steps to be executed. These steps might include:
   - **Setting Up Environment:** Configuring the necessary environment for your application to run. This might involve specifying the version of Python, installing dependencies, etc.
   - **Building Docker Image:** Creating a Docker image that includes your code, model, and any other dependencies. This ensures that your application runs consistently across different environments.
   - **Testing:** Running automated tests to verify that your changes haven't introduced any regressions or errors.
   - **Deployment:** If all tests pass successfully, you can trigger the deployment of your application. This might involve deploying the Docker image to a server or cloud environment.
4. **Docker Deployment:**
   The Docker image that you built in the previous step is deployed to your server or cloud environment. This ensures that your application runs in a controlled and consistent environment, regardless of the underlying infrastructure.
5. **Model Processing:**
   When a request comes in to your deployed application from frontend, the model is used to process the input and generate the output.
6. **Output:**
   The output generated by your application is sent back to the user or system that made the request.

Frontend Infrastructure

1. **User Interaction:**
   Users interact with the frontend of your application, which is built using Streamlit. They input the issue and click get resolution.

2. **Authentication:**
   When users access the Streamlit app, they go through an authentication process. They provide their credentials (username and password). The app verifies these credentials against an authentication service to determine the user's identity.
3. **User Interaction and Requests:**
   Once authenticated, users continue interacting with the Streamlit UI. They might input data and make requests that require processing by the backend.
4. **Request to Backend:**
   When users trigger an action that requires backend processing (e.g., requesting a prediction from a model), the Streamlit frontend sends an HTTP request to the backend API provided by FASTAPI.
5. **Backend Processing:**
   The FASTAPI backend receives the request, processes it, and communicates with the necessary components, such as the model, to generate the required output. The backend might also apply any business logic, data processing, or other necessary operations.
6. **Model Interaction:**
   If the backend requires data processing by a model, it interacts with the model component. This could involve sending data to the model for inference, receiving predictions, and handling any relevant preprocessing or post-processing.
7. **Response to Frontend:**
   The backend generates a response based on the request processing. This response could be predictions, data, or any other output. The response is sent back to the Streamlit frontend.
8. **Frontend Display:**
   The Streamlit frontend receives the response from the backend and displays it to the user.

Data Generation

ChatGPT prompts were used to generate data pertaining to the already given sample dataset. For each type of issue, multiple examples were generated by covering the different categories starting from the reason, the effects and root cause analysis.

Database

Due to the cost contraints and multiple advantages, SQLlite was used to store the data of issues and resolutions

Model Generation

Sentence Transformers

Sentence Transformer model was used to compute the embeddings. Transformers like BERT are designed for word level embeddings where as Sentence Transformers capture the relative information of a particular word in correspondence to the whole sentence. This makes it more effective for capturing the semantic information, making the tasks like semantic searching more robust and accurate.

Model - all-MiniLM-L6-v2

It is a pre-trained model from the Sentence Transformers library. It is a compact version of BERT, and has been trained on various tasks to offer a well-rounded performance across different NLP applications. The model’s versatility with it’s smaller size makes it suitable for scenarios where the computational resources are limited. When the model is aware of the new issue, the sentence embedding of that issue is created. This embedding is compared with the stored embeddings of the already resolved issues using cosine similarity. The resolution of issue which has the highest similarity to the new issue is reported as the resolution of this issue. There might be cases when the input is irrelevant, or the user gives some random input which does not come under the issue categories. In that case, the output is given as “Resolution not found”

Reason for complexity

The reason for using transformers is the complexity involved in capturing the semantic information of words present in an issue. Using simple models in this use case might produce a lot of redundant results where the embeddings will not capture the actual meaning of the word properly.

Conclusion

Our incident management system seamlessly integrates a frontend developed with Streamlit and a backend powered by FASTAPI. Users authenticate and access the system via the Streamlit frontend, which provides an intuitive and interactive interface. After authentication, users interact with the system to report incidents. The FASTAPI backend processes incident-related requests, facilitates communication with models if needed, and orchestrates data operations. There are many improvisations to this already existing approach which can be implemented in future. Though our system is accurate to an extent, there are many further improvements that can be done. Techniques like Fine-tuning can be done to make the model more robust, and generating data along with frequency and history can be done to improvise on the existing model to make the whole system more user interactive.