FLASH: A Workflow Automation Agent for Diagnosing Recurring Incidents

XUCHAO ZHANG, Microsoft, USA
TANISH MITTAL, Microsoft, India
CHETAN BANSAL, Microsoft, USA
RUJIA WANG, Microsoft, USA
MINGHUA MA, Microsoft, USA
ZHIXIN REN, Microsoft, USA
HAO HUANG, Microsoft, USA
SARAVAN RAJMOHAN, Microsoft, USA

Recurring incidents, typically raised by system monitors, often occur repeatedly, demanding significant human effort for troubleshooting. Automating the diagnosis process for these recurring incidents is crucial for minimizing service downtime, reducing customer impact, and decreasing manual labor. While recent agent approaches based on Large Language Models (LLMs) have demonstrated effectiveness in handling complex tasks requiring multiple logical steps, they still suffer from the reliability issue due to a lack of specific diagnostic knowledge. To enhance diagnostic reliability, we propose an workFLow Automation agent with Status supervision and Hindsight integration (FLASH), which significantly improves diagnostic accuracy by incorporating status supervision to break down the complex instructions into manageable pieces aligned with identified status. Moreover, we generate hindsight using LLMs from past failure experiences, progressively enhancing diagnostic reliability for subsequent incidents. We conduct extensive study over 250 production incidents from CompanyX in five different workflow automation scenarios. The results reveal that our FLASH agent approach outperforms state-of-the-art agent models by an average of 13.2% in terms of accuracy. These compelling results underscore the viability of automating the diagnostic process for recurring incidents.

1 Introduction

Over the past decade, the IT industry has witnessed significant growth and development in cloud services [14, 18]. Despite these advancements, cloud incidents such as unplanned interruptions or performance degradation can severely impact customer satisfaction, leading to revenue loss and a decline in customer trust. Recurring incidents, a prevalent type of issue in cloud services, often originate from system-configured monitors. For instance, these monitors may set up to track the connectivity between services. They continuously check the status of the connection, and if they detect any disruptions, such as interrupted heartbeating for several minutes, an incident is raised to alert the connectivity issue. This allows for timely intervention to restore the connection and minimize the impact on service performance and customer experience. Although this type of incident enables prompt resolution of potential issues, it also elevates the overall number of incidents and raise the likelihood of triggering false alarms. This rise in incidents substantially increase the human effort required for incident diagnosis, thereby straining resources and potentially diverting attention from more critical tasks.

Incident diagnosis, as a pivotal task during incident management lifecycle [4], plays a vital role in the process of identifying, analyzing, and determining the root cause of an incident that disrupts normal service operation and provide the corresponding mitigation solution. Due to the large

Authors' Contact Information: Xuchao Zhang, xuchaozhang@microsoft.com, Microsoft, Redmond, WA, USA; Tanish Mittal, tanishmittal0658@gmail.com, Microsoft, Bengaluru, Karnataka, India; Chetan Bansal, chetanb@microsoft.com, Microsoft, Redmond, WA, USA; Rujia Wang, rujiawang@microsoft.com, Microsoft, Redmond, WA, USA; Minghua Ma, minghuama@microsoft.com, Microsoft, Redmond, WA, USA; Zhixin Ren, zhixingren@microsoft.com, Microsoft, Redmond, WA, USA; Hao Huang, hahua@microsoft.com, Microsoft, Redmond, WA, USA; Saravan Rajmohan, saravan.rajmohan@microsoft.com, Microsoft, Redmond, WA, USA.

amount of recurring incidents, automation of recurring incident diagnosis become a vital problem that can save huge amount of human efforts from tedious and repetative troubleshooting process. The repetitive nature of recurring incidents in cloud services presents unique opportunities for automation, thanks to two distinct features: 1) These incidents are typically well-documented because they occur frequently, allowing for the straightforward acquisition of troubleshooting guides (TSGs) that can be integrated into an LLM-based model; 2) The frequent occurrence of these incidents facilitates the accumulation of historical data from past episodes, which can be leveraged to improve the diagnosis accuracy of automated systems.

Despite recurring incidents providing troubleshooting guides and accumulating historical data, automating them remains challenging due to LLM agents typically lacking domain-specific or diagnosis-related knowledge. Attempting to embed such knowledge into LLM models through detailed instructions is hindered by their limited ability to precisely follow instructions, especially across multiple diagnostic steps. This limitation significantly undermines the overall accuracy of existing LLM models when deployed in complex, multi-step diagnostic processes. For instance, in a task involving five steps, even with an 85% chance of accurately following instructions at each step, the cumulative accuracy after all five steps drops to only 44%.

To achieve reliable incident diagnosis, we propose FLASH, a workflow automation agent specifically designed for diagnosing recurring incidents. The FLASH agent integrates the status supervision and hindsight integartion components to improve the reliability of the automated diagnosis of the recurring incidents via additional status reasoning and hindsight supervision. Figure 1 depicts the overview of the proposed model. Using the available troubleshooting documents, the FLASH agent initiates incident diagnosis by following the steps outlined in these documents and leverages supported tools to dynamically gather diagnostic information. Specifically, we have introduced an additional status reasoning step to assess the current status of the diagnosis. For instance, if the step initialization status is identified, the next actions are determined based on the identified status, focusing on selecting tools related to step preparation and planning. Conversely, if the completion status is recognized, the reasoning focuses on determining the next appropriate step in the troubleshooting document, potentially skipping a step if necessary. Additionally, we incorporate a reflection step to correct any errors detected during the diagnosis process, using hindsight derived from past failure cases. When applying proposed flash agent to the automated diagnosis of recurring incidents, several research question need to be addressed. Specifically, can LLM agents accurately diagnose recurring incidents with an elaborated troubleshooting guide? (RQ1) Can historical incidents be helpful in diagnosing similar recurrent incidents? (RQ2) What are the practical considerations of using LLM agents in real-world recurring incident diagnosis? (RQ3)

To address these questions thoroughly, we carried out an extensive evaluation, analyzing 250 incidents and 52 troubleshooting guide documents from real product environments across multiple cloud services provided by CompanyX, one of the largest cloud providers. In addition to the commonly reported accuracy metrics for these experiments, we also involve the product team to gather manual feedback, including metrics like “Time-to-Mitigate Estimation” and “Overall Score”. Since the original incident owners possess the most expertise about their incidents, their evaluations provide valuable insights into the models’ performance. Our contribution can be summarized as:

- We introduce a novel automated agent model for diagnosing recurring incidents, which delivers significantly better performance than state-of-the-art agent approaches and is deployed in a real-world product environment.
- Our human study, conducted with the actual owners of production incidents, provides compelling evidence of the effectiveness of the proposed approach, demonstrating significant success in automating the handling of recurring incidents.

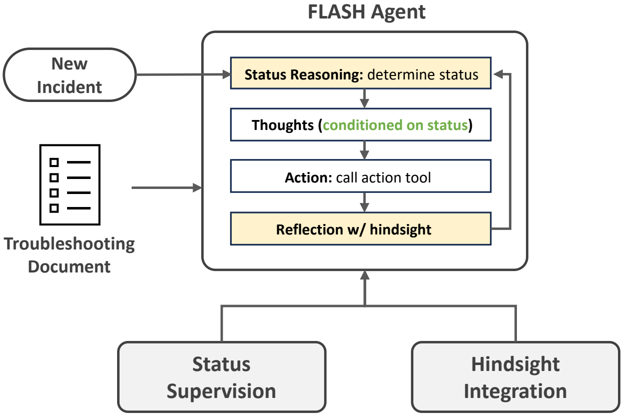

Fig. 1. Overview of FLASH Architecture

- The qualitative study of 52 recurring incident scenarios, using existing troubleshooting guides, offers valuable insights into the challenges of implementing an LLM-based agent for automating recurring incidents.

2 Background

In this section, we start with an introduction to the incident diagnosis task and the latest advancements in LLM models, including LLM agent models. We then proceed with an in-depth discussion of the research questions and the human evaluation conducted in this study.

2.1 Recurring Incident Diagnosis

In large-scale cloud services [5], it is inevitable to encounter production incidents [1] that can significantly impact the customer experience and incur substantial engineering resources for troubleshooting. As one of the most stages in the incident life-cycle, incident diagnosis [3] aims to identify and determine the root cause of incidents. This process is usually complex and demands a significant amount of manual effort, as well as domain knowledge about the services involved. Incidents can originate from two different sources: customer-raised incidents and monitor-raised incidents. Customer-raised incidents are reported by customers when they encounter problems with their system. In contrast, monitor-raised incidents are automatically triggered by monitoring systems that are set up to track system metrics such as memory, CPU usage, and network performance. For example, if a monitor detects an abnormal value, such as a prolonged network interruption, it will automatically generate an incident to notify the team. While this type of incident allows for quicker system verification and helps prevent large-scale failures, it also increases the likelihood of false alarms and requires more human effort for incident diagnosis. Recurring incidents are typically a type of monitor-raised incident that are reported regularly due to specific causes, such as unstable network conditions, unresponsive systems, or long latency tasks. These incidents are not entirely controllable and can occur multiple times a month due to unpredictable situations. However, it is crucial to continuously monitor these incidents, quickly diagnose the underlying causes, and prevent the system from experiencing catastrophic failures.

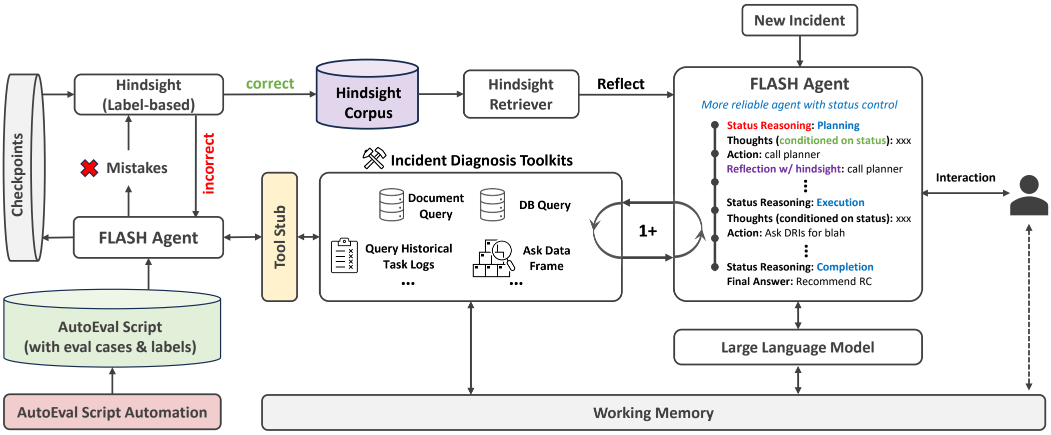
New Incident
Checkpoints
Hindsight (Label-based)
correct
Hindsight Corpus
Hindsight Retriever
Reflect
Mistakes
incorrect
FLASH Agent
Tool Stub
Incident Diagnosis Toolkits
Document Query
DB Query
Query Historical Task Logs
Ask Data Frame
...
1+
AutoEval Script (with eval cases & labels)
AutoEval Script Automation
FLASH Agent
More reliable agent with status control
Status Reasoning: Planning
Thoughts (conditioned on status): xxx
Action: call planner
Reflection w/ hindsight: call planner
Status Reasoning: Execution
Thoughts (conditioned on status): xxx
Action: Ask DRIs for blah
Status Reasoning: Completion
Final Answer: Recommend RC
Interaction
Large Language Model
Working Memory

Fig. 2. Overview of FLASH Architecture

2.2 The Promise of LLM

In recent years, Large Language Models (LLMs) like GPT-4 [17] are revolutionizing the way content is created and utilized, offering innovative tools that enhance creativity, streamline automation processes, and solve complex problems across a wide range of industries. With billions of parameters trained on vast datasets, these LLMs excel at understanding and responding to diverse prompts with remarkable accuracy. Recently, LLM-based agent models [27, 29] have made significant strides, demonstrating their ability to perform complex tasks such as multi-step reasoning, contextual understanding, and even executing intricate instructions in a coherent and contextually appropriate manner [15]. Unlike traditional LLM models, agent models are specifically designed to perform multi-step reasoning. This capability allows them to manage complex tasks that require several steps and logical progression such as knowledge-intensive question answering [25], sequential decision-making problems [23], and multi-step information retrieval [33]. Moreover, they can interact with other systems and take autonomous actions based on the inputs they receive, such as managing workflows, accessing databases, or executing commands. Additionally, with the integration of external knowledge sources, agent models can adapt to specific domains or tasks more efficiently than traditional LLMs.

2.3 Research Questions

RQ1 Can LLM agents accurately diagnose recurring incidents with an elaborated troubleshooting guide?

Previous research has explored the application of agent models to root cause analysis tasks [21]. However, these approaches are significantly limited by a lack of diagnosis-related knowledge, which hinders the agent model's ability to generate a viable troubleshooting plan. In this context, by focusing on the specific features of recurring incidents that include a troubleshooting guide, we investigate the feasibility of using agent models to automate the diagnosis process with a detailed troubleshooting guide. To demonstrate the effectiveness of our proposed approach, we compare its performance against an agent model that lacks a troubleshooting guide, and also against other state-of-the-art models that include troubleshooting plans.

RQ2 Can historical incidents be helpful in diagnosing similar recurrent incidents?

Historical incidents have commonly been utilized in retrieval-augmented models (RAGs) for incident diagnosis tasks, such as providing in-context examples to aid in generating root causes [32]. Recurring incidents, in particular, can gain more concrete diagnosis-related knowledge from the historical data. For instance, errors from similar past incidents are likely to be informative for diagnosing new, similar incidents. The challenge lies in how to efficiently gather this information and apply the knowledge using an automation tool with minimal human effort, as incorrect usage of the information might not only fail to improve accuracy but could also be detrimental. To address this research question, we initially compare the performance of an agent model that either utilizes or does not utilize historical hindsight.

RQ3 What are the practical considerations of using LLM agents in real-world recurring incident diagnosis?

Although troubleshooting guides for recurring incidents and historical data are available to support automated diagnosis, significant challenges persist in translating existing industry documentation into actionable troubleshooting plans for automation models. For instance, the existing documentation for recurring incidents is often ambiguous in outlining specific diagnostic steps. These documents typically lack detailed actionable information and assume that readers possess all the necessary background knowledge to access the required data. To address this research question, we conducted a qualitative analysis of over 50 troubleshooting scenarios from CompanyX. We also collaborated with the document owners to identify issues with the existing documentation and pinpoint the main factors that hinder the automation of incident diagnosis. Based on our findings, we provided recommendations to facilitate the practical implementation of automated diagnosis for recurring incidents.

3 Methodology

We present our FLASH agent that uses the status supervision and hindsight integration to enhance the reliability of recurring incident diagnosis automation. First we provide an overview of our approach in Section 3.1. Then we delve into the details of status supervision in Section 3.2. The diagnostic tools and the human interactions are discussed in Sections 3.3 and 3.4, respectively. Last, the hindsight integration is described in Section 3.5.

3.1 Overview

Figure 2 illustrates the overall architecture of the FLASH framework. The FLASH agent initiates the incident diagnosis using the incident description as input. It then proceeds through the diagnosis process with multiple iterations, alternating between reasoning and action steps. During the reasoning step of the i-th iteration, the agent first decides on the next action $a_i \in \mathcal{A}$ based on the current environment context c_i following the agent policy $\pi(a_i|c_i)$, where $\mathcal{A}$ represents all the possible actions defined by the supported tools. For instance, the agent might initially determine that it needs to generate a troubleshotting plan by calling a "*tsg_planning*" tool. Once the action is determined in the reasoning step, the action step involves executing the action using the selected tool. This sequence of reasoning and action continues until the incident diagnosis is completed. Additionally, we incorporate a status reasoning step to determine the status of the current iteration before moving on to the next reasoning step. This allows us to break down the instructions for the subsequent reasoning step, reducing complexity and thereby enhancing the reliability of diagnosis process. We also introduce a reflection step, using "hindsight" we gathered from previous failed cases to address and correct mistakes made. We utilize the "ToolStub" component to simulate tool outputs when incidents were happening. More details will be discussed later in Section 3.5.

Additionally, we employ a global working memory to record information that can be shared across different diagnostic steps, facilitating end-user feedback and interaction with the agent model.

3.2 Status Supervision

Based on our observasion that the relibility of incident diagnosis agent is highly depended on the instructions and domain knowledge integrated into the agent model. To improve the reliability, we propose a status supervision method to identify the status before the reasoning step. And then try to break down the complexity of instructions conditioned on the status.

Consider an incident diagnosis scenario: At step i, the agent selects an action $a_i \in A$ guided by the agent policy $\pi(a_i|c_i)$, which is based on the current environment context c_i. This context c_i may include observations from the previous step and the historical diagnosis path from all preceding steps. Unlike traditional agents, our FLASH agent model includes an additional step for status reasoning. In this step, it identifies the status $s_i \in \mathcal{S}$ using a pre-defined status detection method f_s based on the current context c_i, where $s_i = f_s(c_i)$ and $\mathcal{S}$ is the pre-defined set of statuses. This process allows the agent to comprehend the status and facilitates the integration and simplication of additional knowledge based on the identified status s_i, thereby enhancing its decision-making capabilities in complex environments more effectively. While the status detection method can vary in format, we employ an Large Language Model (LLM) for this function due to its robust reasoning capabilities.

Once the status s_i is determined, we generate what we call a status-conditioned context, denoted as $\hat{c}_i = \mathcal{G}(c_i|s_i, \mathcal{I})$, where $\mathcal{I}$ represents the set of instructions related to the incident diagnosis. This process involves generating the status-conditioned context $\hat{c}_i$, conditioned on the current status s_i and the relevant instructions or domain knowledge $\mathcal{I}$. For example, if there are specific instructions associated with the status s_i, we can incorporate these instructions into the context and remove the irrelevant ones. This process allows us to break down a large and complex set of instructions into smaller, manageable segments based on the detected status. Consequently, the agent's actions are determined by the updated policy that considers the status-conditioned context: $\pi(a_i|\hat{c}_i, s_i)$. This approach ensures that actions are more effectively tailored to the current circumstances.

Here, we aim to discuss the configuration of the status set $\mathcal{S}$. Diagnosing an incident typically involves generating a diagnostic plan and then following this plan to execute the diagnostic steps accordingly. However, since the diagnostic process is not always sequential, we must dynamically determine the next step based on the current context. Additionally, the plan often requires refinement before we can initiate the next step. Therefore, we use the various stages of the diagnostic process as our predefined statuses, including diagnostic planning, step initialization, execution, and completion, as follows:

- *Diagnosis Planning*: This status is designated for generating the troubleshooting plan. Specifically, when this status is detected, we restrict the instructions and tools to those directly related to diagnosis planning.
- *Step Initialization*: The status indicates when the agent model is preparing to initiate a diagnostic step. During this initialization phase, we refine the diagnosis plan by reviewing the existing plan, incorporating missing details from the troubleshooting document, and ensuring that all necessary information is included.
- *Step Execution*: The execution status marks the phase where a diagnostic step is being carried out. It enables the use of various tools to complete all required actions within the step. A critical aspect of this status is determining whether all actions have been completed by the agent. If completed, the status transitions to completion; if not, the agent must continue executing the remaining actions, and the status remains as execution.

- *Step Completion*: The completion status indicates that the current diagnostic step has been successfully completed. This status also entails determining the next appropriate step in the diagnostic process. Given that incident diagnosis does not proceed in a sequential order, the subsequent step must be dynamically determined based on the current findings and context. If the expected results are achieved during this stage, the process can directly transition to generating the final answer.

In addition to the predefined statuses, users can customize their own statuses based on their domain knowledge for specific usage scenarios. To implement this, users need to provide a corresponding status detection function $f(c_i)$, which detects particular conditions, and a status-conditioned context generation method $\mathcal{G}(c_i|s_i, \mathcal{I})$ to generate the context used in the reasoning step. For instance, if a user identifies a deadlock caused by repeated calls to one or more tools, the detection function is required to identify this deadlock based on the diagnostic history. Subsequently, the user can incorporate specific instructions and tools into the reasoning step's prompt to resolve and prevent the deadlock issue.

3.3 Incident Diagnosis Tools

For FLASH agent, we propose two types of tools for incident diagnosis purpose: system tools and utility tools. System tools are pre-defined tools designed for specific systematic functions, such as *human_query*, which is used for asking questions to the end user. We defined three system tools for our agent as follows:

- *human_query*: This tool is invoked when the agent encounters something unclear or unmanageable and needs to ask the end user for an answer. For example, if a SQL query requires a variable such as the region name, which is unknown to the agent, the agent will prompt the end user for this information.
- *dummy_action*: The tool is designed to bypass the action step when the reasoning step has already completed all necessary tasks, eliminating the need for further action. For example, if a diagnostic step simply involves explaining certain concepts without requiring any specific actions, this tool can be called to move on to the next step.
- *llm_reasoning*: In certain scenarios, an action requires execution through an LLM call. For instance, when the task is to extract the region name from an incident title using a specific example, the agent can utilize this tool to initiate the LLM call.

Utility Tools are functional tools that carry out practical tasks, such as querying databases or logs. We have introduced two pre-defined tools for generating the diagnosis plan, as follows:

- *diagnosis_planning*: This tool is designed to generate an overarching diagnosis plan based on the troubleshooting document. The steps in the document often consist of complex and detailed procedures, which are impractical to fully incorporate into the high-level plan. Therefore, this tool creates a simplified plan, giving both the agent and the end-user a clear outline of the troubleshooting strategy without delving into specific execution details. In the implementation phase, the *plan_refining* tool will later be used to provide more detailed steps, which will be explained further.
- *plan_refining*: This tool is employed to incorporate all necessary details into the execution plan to make it actionable. It is designed to be used right before initiating the execution of a plan. The tool queries details from the troubleshooting document according to the context of the current step, ensuring that the execution plan is both detailed and precise.

In addition to the pre-defined tools for diagnosis planning, we have also introduced other tools that facilitate access to system diagnosis information, enhancing our support for various incident diagnosis scenarios. These tools are also designed to be easily extendable by users, allowing for customization and expansion to address additional diagnosis needs. For example, users can implement a "db_query" tool to query a database or a "log_query" tool to access log files.

3.4 Human Interaction

We support two types of human interaction. First, through the *human_query* tool, the agent can ask the end user for information that it cannot find on its own. The second scenario involves pausing the diagnosis at each step to request user feedback. This approach allows users to confirm satisfaction with each step's output and promptly identify and correct any mistakes made by the agent. To facilitate user feedback, we present details of both the reasoning and action steps in a conversational format within a chatbot. Users can easily type their feedback into the chatbot's input box. Moreover, to simplify user actions when approving the agent's output, we provide a "continue" button that allows users to advance to the next step without typing. Additionally, we offer a silence mode, which automatically proceeds with the diagnosis without seeking user feedback at each step, minimizing interruptions. However, users still have the option to use a "stop" button to interrupt the agent and provide feedback if they notice any errors. The collected human feedback can be integrated with the hindsight integration component to improve the accuracy of diagnoses for future incidents.

3.5 Hindsight Integration

In this section, we introduce a hindsight integration component designed to leverage past failure experiences to correct the agent's mistakes and enhance the accuracy of the agent model. Specifically, hindsight can be automatically generated from past failed cases and integrated into the reflection step to improve the reliability of the agent's diagnosis. To automate the collection of hindsight, we first present an automatic evaluation framework, called AutoEval, which automates the evaluation of diagnostic incidents using step-by-step labels. We will then explain how hindsight is automatically collected through the AutoEval framework, followed by a discussion on how hindsight is retrieved and incorporated into the reflection step to correct the agent's errors.

3.5.1 Automated Evaluation. The AutoEval framework aims to automatically perform incident diagnosis and identify failure cases using predefined step-wise labels. To achieve this, we start by gathering similar historical incidents, using the incident descriptions as inputs and the expected step-by-step outputs as labels. The AutoEval framework will then attempt to diagnose the historical incident and verify whether the agent's output for each step matches the prepared label. If it does, we consider the agent's behavior to be correct and proceed with the diagnosis. If not, we recognize this as an error made by the agent and attempt to generate hindsight from it.

Since the outputs for various diagnostic steps typically vary in format, such as appearing as a table when a database query is run or as a numeric ID when information is fetched from a log file, we opt to use a textual description to succinctly capture the key content of the expected results, thereby simplifying the label. This allows us to employ a Large Language Model such as GPT-4 to compare the model output with the label and determine whether the expected results are included in the model output. Figure 3 illustrates an example of both the model output and a step label. Although the model outputs a table that includes two fields with both ID and cluster name in a JSON format, the LLM determines the correctness of the output based on whether the four clusters appear in the model output, regardless of the format of the output.

3.5.2 ToolStub. We are encountering a challenge when utilizing historical incidents in automatic evaluation, as the system conditions during incident evaluation differ from those when the incident originally occurred. This discrepancy prevents us from relying on the current system to accurately replicate the results from tools as they interacted at the time of the incident. To address this issue,

Label: A table containing these four clusters: BY1PrdApp36, BY1PrdApp35, IAD01PrdApp59, IAD20PrdApp66.

Model Output: [“id”: 1, “cluster”: “BY1PrdApp36”, “id”: 2, “cluster”: “BY1PrdApp35”, “id”: 3, “cluster”: “IAD01PrdApp59”, “id”: 4, “cluster”: “IAD20PrdApp66”]

Fig. 3. An Example of Step Label and Model Output

```
“status”: “<3:execution>”
“tool_name”: “db_query”,
“inputs: {
        “cluster”: “clusterA”,
        “db”: “databaseA”,
        “query”: “<sql_query>”
},
“output”: “[dcount(serviceId): 0]”
```

Fig. 4. An Example of ToolStub for a database query.

we propose the implementation of a “ToolStub” component. This component is specifically designed to simulate the outputs of tools as they would have appeared during the original incidents. ToolStub defines the inputs for a tool invocation and provides the expected outputs directly, without the need to run the actual tool. We simply need to prepare the expected output for each tool invocation, and ToolStub will simulate these outputs. Figure 4 illustrates an example of a ToolStub entry that simulates the output for a database query. If the agent invokes the *db_query* tool with the exact same inputs, including the cluster, database, and query, the ToolStub will directly return the output “[dcount(serviceId): 0]” which can be parsed to a data frame without actually running the query. Note that we use a placeholder for the SQL query in the example, but it is necessary to replace it with the exact query. We now support exact string matching for identifying the corresponding tool stub. To enhance the efficiency of matching within the ToolStub, we have integrated this feature with predefined step statuses. This integration eliminates the need to iterate through all ToolStub entries to find a match, allowing us to focus only on tools associated with the same status.

3.5.3 Hindsight Generation and Validation. Using the AutoEval framework, we automate the diagnosis of historical incidents by collecting data on past failures through a comparison of model outputs and step-wise labels. If a mismatch occurs, it signals an error in the current step. Our objective in this step is to automatically generate hindsight that can help correct the error in future incidents. To achieve this, we gather all relevant context information, including historical diagnosis steps, and pairs of incorrect and expected outputs. We then use the prompt in Figure 5 to request the LLM to automatically generate hindsight for the current error. Specifically, we provide the LLM with historical diagnostic logs and the expected result, asking it to generate hindsight for identifying the root cause of the issue and then suggest the corresponding solution to resolve it.

While the LLM demonstrates promising capabilities in language understanding and reasoning, we still cannot guarantee that the generated hindsight will effectively resolve errors. The most direct way to validate the usefulness of these hindsight is by retrying the failed diagnosis step using the generated hindsight and verifying their effectiveness. To facilitate this validation, we have implemented a checkpoint function that enables the agent to roll back to the state of the previous step and integrate the hindsight into the reflection step. This process helps determine whether the hindsight is effective in correcting the error and achieving the expected outcome. If successful, the hindsight is stored in a hindsight corpus; if unsuccessful, the process of generating hindsight is repeated until it passes the validation or reaches the maximum number of trials.

You are a helpful assisstant helping to generate hindsight based on the past actions.
- Historical diagnostic logs: {{historical_diagnosis_log}}
- The previous steps encountered problems and cannot generate the intended results. Here is the expected result: {{expected_result}}

Imagine you're in an advisory role, with the responsibility of offering suggestions based on historical diagnostic logs. Your task is to pinpoint the root cause of the problem that's leading to unexpected outcomes, as detailed in the "Thoughts:" section.
Then, you're required to provide a fix in the "Solution:" section. The expected result provided is just for your reference, you cannot directly use it as the suggestion. Instead, you need to analyze the historical diagnostic log to understand the underlying issue and formulate an appropriate solution.

Fig. 5. Prompt of Hindsight Generation

Thoughts: The assistant was supposed to run the Kusto query for each cluster to check if they have live traffic. However, it seems like the assistant only ran the query for one cluster. This is likely the reason why the assistant was not able to generate the expected results.
Solution: Execute the Kusto query for all clusters collected in the previous step.

Fig. 6. Example of Hindsight

You are a helpful assisstant helping to generate hindsight based on the past actions.
- Historical diagnostic logs: {{historical_diagnosis_log}}
- Retrieved hindsight: {{hindsight_section}}

Imagine you're in an advisory role, with the responsibility of offering suggestions based on historical diagnostic logs. Your task is to analyze whether there are any problem from the previous diagnosis steps. Please provide the following sections. Analysis: provide your detailed analysis here. Need Reflection: provide "Yes" if you think the hindsight or reflection is helpful to improve the answer. Otherwise provide "No".
Solution: provide a fix based on the analysis on all the provided information in the hindsight.

Fig. 7. Prompt of Reflection

To store the generated hindsight, we first use the text-embedding-ada-002 model [2] to create an embedding vector for each hindsight. Next, we build a retrieval index using the FAISS library [10], which facilitates efficient similarity searches. Figure 6 illustrates a hindsight generated by our automated hindsight generation framework. This hindsight attempts to resolve an issue where we query the status of all clusters retrieved from the previous step, but only some clusters are executed while others are missing in the execution.

3.5.4 Hindsight-based Reflection. Once the hindsight is generated and stored in the hindsight corpus, we aim to utilize it during the diagnosis of new incidents in the reflection step. To reduce the overload during this phase, reflection is conducted only when the agent reaches the step completion status. At this point, we use the current context as input to generate an embedding using the text-embedding-ada model and retrieve the top-3 most relevant hindsight instances. Then, we leverage an LLM to determine whether the extracted hindsight is applicable to the current context, as illustrated in Figure 7. Specifically, we present the LLM with both the log and the retrieved hindsight, asking it to analyze the situation and determine whether the reflection step is needed with a "Yes" or "No" answer. If the response is "Yes", we then request a fix solution based on the analysis. If the answer is "No", we skip the reflection step and continue the diagnosis. Otherwise, the extracted hindsight is integrated into the reflection step to attempt to fix the detected issue. Note that the reflection step is typically costly to conduct and may lead to incorrect outcomes if misapplied. Therefore, we've divided the reflection into two distinct steps. The first step, as previously shown, is to determine whether reflection is necessary. Following this, we integrate the

Table 1. Data Statistics for Incident Diagnosis

Scenario	# Groups	# Incidents
Setting-Drift-Alert	6	60
TIP Session	5	50
CAPA-WrongMapping	5	50
NameResolver-PubSubQueueLength	5	50
NSM-To-RNM-Connection	4	40

hindsight and the proposed solution to proceed with the reflection step. This approach may reduce the latency associated with reflection and minimize the negative impacts of incorrect reflection.

4 Experiment

In this section, We empirically verify the performance of the proposed method against other methods on recurring incident diagnosis task.

4.1 Experiment Setup

4.1.1 Dataset and Labels. For the answering the research questions RQ1 and RQ2, We evaluated our model on real incident diagnosis data, covering five different troubleshooting scenarios at CompanyX. Each scenario typically shares the same troubleshooting guide for a specific type of recurring incident. Additionally, for each scenario, we broke down the incidents into different groups based on various troubleshooting paths and their corresponding diagnosis results. Table 1 provides statistics on the selected scenarios, including the number of groups and incidents. Within the same group, incidents may share the same labels. These labels are described in natural language format. For example, if troubleshooting reveals that the reported incident is a false alarm, the label might be "*It's a false alarm.*" For research question RQ3, we conducted a thorough study of 52 troubleshooting scenarios across multiple products and teams at CompanyX. For each scenario, we used the corresponding troubleshooting document to conclude the practical considerations of automating incident diagnosis using LLM agents.

4.1.2 Baseline Methods. We select the following baseline methods for comparison: (i) *Chain-of -Thoughts* (CoT) [26]: We use CoT to enhance the reasoning abilities of vanilla LLMs. This approach encourages the model to break down the input problem into smaller, more manageable parts, effectively thinking through the problem step by step. In our work, we utilize the GPT-4 model as the backbone LLM to achieve incident diagnosis through step-by-step reasoning. (ii) *Retrieval-Augmented Generation (RAG)* [13]: For the RAG baseline, we instructed the model to retrieve the most relevant troubleshooting document and integrate it into the GPT-4 model to generate the diagnostic output. (iii) *In-Context Learning (ICL)* [32]: The in-context learning model utilizes historical incidents as the retrieval corpus to identify the most similar incidents. These similar incidents are then used as in-context examples for the large language model (LLM) to generate the diagnostic result. (iv) *ReAct* [29]: The ReAct agent model interleaves reasoning and tool usage steps, combining principles from reasoning-based approaches like Chain of Thought (CoT) with tool usage models such as Toolformer [22]. This approach is particularly well-suited for the incident diagnosis task, which requires both reasoning and tool invocation. (v) *TaskWeaver* [19]: TaskWeaver is a code-first agent framework that seamlessly plans and executes data analytics tasks by interpreting user requests through code snippets and efficiently coordinating various plugins as functions to perform these tasks in a stateful manner.

Table 2. Experiment Results on the Diagnosis Data

	SettingDrift	TIPSession	CAPA	NameResolver	NSM2RNM	Average
CoT [26]	4.0	0.0	30.0	0.0	0.0	6.8
RAG [13]	14.0	80.0	30.0	30.0	31.3	37.1
ICL [32]	63.0	76.0	35.0	30.0	30.0	46.8
ReAct [29]	52.0	62.0	22.0	40.0	32.5	41.7
TaskWeaver [19]	80.0	71.0	26.0	84.0	42.5	60.7
FLASH	**87.0**	**86.0**	**48.0**	**87.0**	**61.3**	**73.9**

4.1.3 Evaluation Metrics. We choose both quantitative metrics and human evaluation for the evaluating our model. For the quantative metrics, we use the accuarcy compared to the final ground truth labels as our metrics. For the human evaluation, we use the following metrics for querying the On-call engineers (OCEs) who is responsible for troubleshooting the incident in product environment. (i) TSG Retrieval Accuracy: We assess the accuracy of the retrieved TSG by comparing it to the ground truth TSG, in order to evaluate our agent's ability to identify the correct TSG document. (ii) Troubleshooting Plan Correctness: OCEs are asked to assess whether the troubleshooting plan generated was appropriate for resolving the incident. (iii) Diagnosis Accuracy: We asked the OCEs to assess whether the automated diagnosis result is accurate and corresponds to the actual problem. (iv) Time-to-Mitigation (TTM) Estimation: We asked the OCEs to estimate the time-to-mitigation. (v) Overall Rating: We asked the OCEs to provide the overall rating from 1 to 5, where 5 represents the highest score and 1 is the lowest score.

4.2 Evaluation on RQ1

To address RQ1, we conducted incident diagnosis experiments using our proposed FLASH model across five real-world scenarios. Table 2 presents the evaluation results in comparison with five state-of-the-art (SOTA) baseline methods. Our findings indicate that the FLASH model outperforms all other methods in all five troubleshooting scenarios, exceeding their accuracy by more than 13.2% on average. Notably, non-agent models, such as CoT, retrieval-augmented models, and in-context learning models, exhibited suboptimal performance, with accuracy rates below 50%. This is due to their inability to perform step-by-step diagnoses and call external tools to extract dynamic information, which limits their capability to follow workflows effectively. In contrast, agent models, such as the TaskWeaver agent approach, achieve significantly better performance, outperforming ICL models by 13.9%. However, ReAct [29], despite being an agent model, does not perform as well as other agent models, primarily due to the high frequency of mistakes made during the intermediate steps, which leads to incorrect final answers. Our FLASH agent shows more than 27.1% performance gain over any non-agent models and outperforms both TaskWeaver and ReAct by 13.2% and 32.2%, respectively. To answer the research question, we find that in certain scenarios, such as SettingDrift and NameResolver, nearly 90% accuracy is achieved. This indicates that LLM agents can accurately diagnose recurring incidents based on a detailed troubleshooting guide. However, in CAPA scenarios, performance drops to around 50% because these scenarios require specific external tools that the agent model cannot fully interpret or utilize.

We also conducted a human evaluation using 15 incidents from two troubleshooting scenarios: *Setting Drift Alert* and *NSM-To-RNM connection*. We asked the corresponding OCEs to provide feedback based on the output of the FLASH agent. As shown in Table 3, the results indicate 100% accuracy in TSG retrieval and troubleshooting plan generation. For the overall accuracy, the system successfully delivered accurate diagnoses in 11 out of 15 incidents. The improvement in TTM is calculated based on a comparison between the average diagnosis time of 5.3 minutes and the

Table 3. OCE feedback on the automated incident diagnosis

	OCE Feedback
TSG Accuracy	15/15 = 100.0%
Plan Correctness	15/15 = 100.0%
Diagnosis Accuracy	11/15 = 73.3%
TTM Estimation	5.3 mins
Overall Score	4.3/5

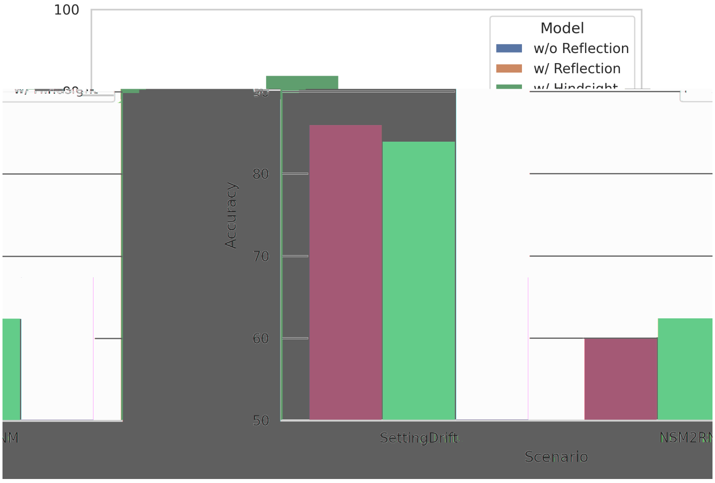

Fig. 8. Performance on Hindsight Integration

average mitigation time of 90 minutes. Lastly, the overall rating was 4.3 out of 5, based on the 15 incidents.

4.3 Evaluation on RQ2

To address RQ2, we applied the hindsight approach to two scenarios: *Setting Drift Alert* and *NSM-To-RNM connection*. We used the first five incidents for each scenario as past incidents for collecting hindsight and the remaining five incidents for testing purposes. Figure 8 shows the results for three settings of the FLASH model: without reflection step, with vanilla reflection, and with hindsight-integrated reflection. Based on the results, we observe that integrating hindsight can significantly improve performance. Specifically, the hindsight-integrated reflection achieves a 6% and 7.5% performance improvement compared to the FLASH model without reflection step for *Setting Drift Alert* and *NSM-To-RNM connection*. This indicates that the agent model struggles to resolve issues independently without extra hindsight information. However, when provided with step-wise labels, the LLM can analyze why the model cannot generate the expected output and suggest appropriate fixes. In contrast, vanilla reflection without hindsight integration performs similarly to the FLASH model without reflection. Specifically, the performance in the *SettingDrift* scenario is 2% worse with reflection than without it. This suggests that generating reflection without the benefit of hindsight can actually harm performance, even making it worse than having no reflection at all. However, a similar conclusion cannot be drawn for the *NSM2RNM* scenario. We observe that even

Table 4. Issues preventing from TSG Automation

Issues	Description
AuxiliaryTSG	The TSG is not a main entry of troubleshooting and it includes part of troubleshooting steps that be referred by other TSGs.
Ambiguous Action	The action steps are explained but some of them may not clear enough for agent to follow
Unavailable Tools	The troublshooting step requires some tools that are not available for agent to call and the tool function is not clear for implementation
Incomplete Action	Action steps are not completely provided
Incomplete Document	Dummy document with few troubleshooting information
Subsidiary TSG	One TSG refers to another TSG for troubleshooting, which cannot be directly supported by existing agent due to the connection between the TSGs are not clearly specified.

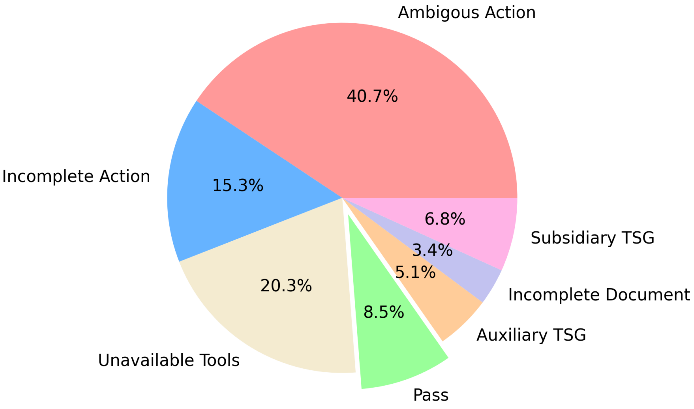

Fig. 9. Statistics of TSG Issues

without hindsight, the reflection step still leads to a performance improvement of around 2.5%. These results suggest that reflection without hindsight does not guarantee consistent improvement across different scenarios. However, when integrated with hindsight, we see a clear performance boost in both scenarios.

4.4 Evaluation on RQ3

To address the research question on the practical considerations of using LLM agents for real-world recurring incident diagnosis with troubleshooting documents, we investigated 52 real-world troubleshooting documents from cloud service incidents at CompanyX. We categorized the main issues preventing the direct application of TSG (Troubleshooting Guide) in diagnosis automation into six categories, summarized in Table 4. Two of these issues are caused by the hierarchical structure of the TSG document design. The *AuxiliaryTSG* issue arises when a TSG document is not the main entry point for troubleshooting, making it infeasible for the agent to start diagnosis based on intermediate steps. In contrast, the *SubsidiaryTSG* issue occurs when a TSG refers to another TSG for troubleshooting but lacks clear instructions on how to use the other TSG for continued troubleshooting. We also identified two issues arising from incomplete information at both the document level and the actionable action level: *Incomplete Action* and *Incomplete Document*. Both issues provide insufficient information for the agent to follow effectively. For instance, an *Incomplete Action* issue in some TSGs may state "extract the status of the listed clusters" without providing actionable steps on how to extract the cluster status. Meanwhile, an *Incomplete Document* issue refers to troubleshooting documents that provide incomplete information without any clear actions or background introduction. Besides the issues with incomplete information, we also encountered the *Ambiguous Action* issue. This occurs when the actions mentioned in the document are unclear and require clarification from engineers. For instance, a troubleshooting guide might provide an SQL query for checking the status of different clusters but omit essential details such as the database name or region name, making the action ambiguous. The last but not least issue is the *Unavailable Tool* issue. This occurs when a troubleshooting step requires the use of tools that are not available for the agent to automate. While this issue can often be resolved by asking a human to perform the action and provide the result back to the agent, it hinders further automation of the diagnosis process. The constant need to interrupt the process with questions to a human is not ideal for incident diagnosis automation.

Based on the detected issue categories, we manually classified each TSG into at least one of these categories, or categorized it as "Pass" if the TSG can be directly used for automation with no or minor changes. Figure 9 shows the percentage of each issue in our study. We found that the most common issue is *Ambiguous Action*, covering around 40% of cases, and *Unavailable Tools*, covering 20% of cases. Moreover, around 8.5% of TSGs fall into the "Pass" category and can be directly used. Combining these three categories, we found that around 70% of cases can be automated with minor TSG revisions or human involvement. This shows great potential for realizing automated diagnosis of recurring incidents.

4.5 Case Study

We present a case study showcasing incident diagnosis based on the FLASH agent model using a real-world setting-drift-alert incident. The diagnosis involves five main steps, as listed in Figure 10. This section will outline the key actions and results for each step. Initially, the agent model generates an overall plan for incident diagnosis by invoking the "diagnosis_planning" tool, using the incident title as input. This tool retrieves the relevant TSG from a large TSG corpus and produces a high-level plan that outlines all the necessary steps. Next, the FLASH agent transitions the process from planning to the initialization of the first step. For the initialization stage, the agent calls the "plan_refining" tool to reconstruct the diagnosis plan for the current step, ensuring all required details are included. This step ensures that the diagnosis plan is comprehensive and well-prepared. Once the plan is prepared, the agent moves the current status to execution and invokes the "incident_query" tool to extract the setting drift name from the incident title. In our case, the setting name "XYZSpace" should be extracted from the incident title "XYZSpace is drifted". After this, the status transitions to completion, and the agent determines that the next step is step 1. In step 1, the agent needs to identify clusters with drifted settings by executing a database query. Since the query template is provided by the TSG, the agent must automatically substitute the setting name in the query with the one extracted in the previous step. It will use the "db_query" tool to retrieve the clusters and proceed to the next step. In our case, five clusters are returned. In Step 2, the agent needs to filter the cluster table based on the specified four regions. It will use the "dataframe_query" tool, which automatically generates and executes a Python code snippet to filter

Incident Description: [SettingDrift] XYZSpace is drifted
————————

Troubleshooting Steps in TSG

Step 0: Find the Setting Name From the Incident Title
The drifted setting name is listed in the incident title, such as "EnableLiveSetting" in the incident "Incident 4100000001 : [SettingDrift] EnableLiveSetting is drifted".

Step 1: Find Clusters with Drifted Setting
After receiving this alert, the first step is to find out which clusters have the incorrect (or different) setting value relative to the gold setting value. This can be achieved by the following query (note that the driftedSettingName must be replaced with the actual setting name): Database Query (Omitted)

Step 2: Filter out Stage and Canary Regions
The following regions are stage and canary which typically do not have customer impact if settings are drifted, and sometimes those clusters in the regions are used for testing. So we can filter them out from the result in Step 1: A table listed four region names (omitted)

Step 3: Check if a Cluster has Live Traffic
Goal setting values are applied during the cluster buildout. Before a cluster is released, the settings are expected to be drifted. The following query is to count the number of tenants: Database Query (Omitted). At this final step, we should have a list of clusters with the drifted settings.

Mitigation: *If the number of cluster is 0, the incident is a false alarm. The DRI is expected to paste the TSG output in the incident discussion following the investigation steps, and mark the incident as false alarm. If the number of clusters is equal to or less than 5, the DRI can run the "SetServiceSetting" OaaS job to mitigate the incident. If the number of clusters is greater than 5, the DRI should consult with their manager or the feature owner to investigate why the setting in many clusters are drifted.*

Fig. 10. An Example of Recurring Incident TSG

out the clusters listed in the table provided in the TSG. In our case, five clusters were returned from the previous step, and two of them, which appeared in the filtering table, were successfully excluded. In the next step, the agent checks the live traffic of the clusters by counting the number of tenants identified in the previous step and retaining only the clusters with active traffic. Based on the number of clusters with live traffic, the agent can then recommend various mitigation plans following the instructions in the mitigation step.

5 Related Work

5.1 Automated Incident Diagnosis

The rise of large-scale cloud services has inevitably led to incidents that detrimentally impact customer experience and necessitate allocating substantial engineering resources. Recent research has explored the application of Large Language Models (LLMs) [1, 9, 32] to streamline this process, leveraging their advanced capabilities to enhance incident resolution. One such study by *Zhang et al.* [32] investigates the utilization of LLMs for automated root cause analysis of cloud service incidents. This work uses LLM to predict the Root Cause of an incident given the description, title and contextually related examples with the solution. It also shows that when in-context examples are given to a GPT-4, it works at par with a finetuned GPT-3 model in terms of understanding and predicting the root cause.

More recently, LLM agents have been performing better than just LLM in solving complex problems. As these agents provide LLMs a proper framework to interact with memory along with functionalities like retrieval mechanisms, integration with external tools etc. Similar to this, *Roy et al* [21] explored the usage of the ReAct agent for the task of predicting the Root cause for incidents. Sometimes, Addressing these incidents effectively can also involve intricate diagnostic steps to identify the root cause, followed by a series of actions to mitigate the issue—both of which are time-consuming and cost inefficient. LLM-based agent frameworks have shown promising results when it comes to solving problems involving multiple steps of logical deductions. There are multiple LLM agents like ReAct agent [29], TaskWeaver[19] etc. Taskweaver is a code-first agent framework which interprets user requests through code snippets and efficiently coordinates different plugins in the form of functions to execute complex tasks. By utilizing a structured framework like TaskWeaver, LLMs can efficiently navigate through multiple steps, leading to quicker and more accurate resolution in multistep issues.

To our knowledge, there has been no work which explores the usage of LLM agents for recurring incident diagnosis. This is the first work which explores the Taskweaver for the automated incident diagnosis and also creates a new LLM agent named Flash which is working even better than the existing Taskweaver.

5.2 Workflow Automation

Workflow automation has long been considered as a promising solution for labor-intensive and repetitive tasks [12, 20]. While automation can address some complex issues through simple, heuristic, policy-based functions [16], these policies are typically developed in isolated and controlled environments. In contrast, the human mind is complex, learning from a diverse range of experiences. As a result, policy-based automation falls short of replicating human-level decision-making and planning [24].

With significant advancements in large language models (LLMs), these models have demonstrated near-human intelligence in problem-solving and decision-making. As a result, several studies are now utilizing LLMs as central controllers to automate workflows, enabling human-level decision-making within these processes [31]. For instance, *Gebreab et al.* [7] introduced an LLM-based multi-agent framework designed to automate administrative tasks in clinical settings, such as parsing instructions, breaking down tasks, and executing sequences of actions. Similarly, *Ye et al.* [31] proposed an LLM-driven agent to automate robotic process automation (RPA). RAP [11] leverages a dynamic retrieval mechanism to use past experiences relevant to the current context as knowledge. AutoGuide [6] augments pre-trained LLMs with implicit knowledge from offline experiences through state-aware guidelines.

Recently, several studies [6, 8, 11, 28, 34] have explored the use of LLMs to automate workflows by integrating workflow knowledge to minimize hallucinations, adapt to new scenarios, and improve decision-making. For instance, KnowAgent [34] employs an action knowledge base and a self-learning strategy to guide action paths during planning. Meanwhile, SmartFlow [8] leverages computer vision and natural language processing to interpret visible elements on graphical user interfaces and transform them into textual representations. However, most existing methods aim to incorporate domain knowledge to develop a more effective execution plan. However, they cannot be directly applied to our recurring incident diagnosis scenario, which already includes step-by-step guidance but lacks the ability to execute the plan reliably.

6 Discussion and Future Work

In the discussion section, we address the limitations and future directions of our approach. One limitation is that its effectiveness is highly dependent on the instruction-following capability of large language models (LLMs). For this reason, we chose the state-of-the-art LLM, GPT-4, based on our observation that most predecessor models, such as GPT-3.5 [30], struggle with instruction-following tasks. Additionally, to enable hindsight integration, we need to prepare step-by-step evaluation cases, which typically require extra human effort. This design may hinder the scalability of the hindsight integration method for handling a large number of recurring scenarios.

There are several validity threats that require consideration. Firstly, successful automated incident diagnosis relies on high-quality troubleshooting documents to guide the agent model through multi-step diagnosis, as off-the-shelf LLMs lack the domain-specific knowledge required for particular incident scenarios. If we encounter a low-quality troubleshooting document, our current design lacks the ability to accurately follow it for effective diagnosis automation. Secondly, the small sample size in our human subject study presents a challenge in achieving statistical significance across all evaluation metrics. Scaling up the study is difficult and may vary in feasibility depending on the nature of the research.

For future work, we aim to automatically generate hindsight using collected successful cases, eliminating the need for manual labeling. By comparing the current diagnosis process with historical success cases, we can identify differences and uncover potential issues. This approach would significantly reduce the effort required to collect hindsight offline and improve the accuracy of recurring incident diagnosis as more incidents are processed over time. Additionally, we plan to develop an automatic TSG refining tool to assist end users in revising troubleshooting guides (TSGs) for easier onboarding into incident diagnosis automation. This tool will help identify existing issues in the TSGs and offer suggestions to the users on how to address these problems.

7 Conclusion

In this paper, we present the effectiveness of utilizing cutting-edge LLM-based agent model for diagnosing recurring incidents. We propose a workflow automation agent with status supervision and hindsight integration to decomposite the complex instructions and integrate past failure experience for improving the reliability of workflow execution. Through extensive experiments on 5 real-world incident diagnosis scenarios from cloud services, we demonstrate that our FLASH approach outperforms the existing workflow automation model by an average of 13.2% across all the scenarios. Human evaluation from the incident owners also incident promising diagnosis accuracy and TTM reduction. The qualitative analysis on 52 real-world troubleshooting scenarios also pinpoint the existing issue preventing from automated diagnosis and showcases that the feasibility of extending the solution to more scenarios.

8 Data Availability

To facilitate the reproduction of our proposed approach, we will make the source code publicly available after the paper is published. However, since the data we used comes from CompanyX and includes real-world incident data, which may contain customer information, we are unable to release the full dataset due to privacy concerns. Instead, we will provide a detailed guide in our code repository on how to create a similar dataset step-by-step, along with some sample data for reference.

References

[1] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending root-cause and mitigation steps for cloud incidents using large language models. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), 1737–1749.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[3] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688.

[4] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, et al. 2020. Towards intelligent incident management: why we need it and how we make it. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1487–1497.

[5] Yuan-Shun Dai, Bo Yang, Jack Dongarra, and Gewei Zhang. 2009. Cloud service reliability: Modeling and analysis. In *15th IEEE Pacific Rim International Symposium on Dependable Computing*. Citeseer, 1–17.

[6] Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. 2024. AutoGuide: Automated Generation and Selection of State-Aware Guidelines for Large Language Model Agents. *ArXiv* abs/2403.08978 (2024). https://api.semanticscholar.org/CorpusID:268385171

[7] Senay Gebreab, Khaled Salah, Raja Jayaraman, Muhammad Habib ur Rehman, and Samer Ellaham. 2024. LLM-Based Framework for Administrative Task Automation in Healthcare. *Conference: 2024 12th International Symposium on Digital Forensics and Security (ISDFS)* (04 2024), 1–7. https://doi.org/10.1109/ISDFS60797.2024.10527275

[8] Arushi Jain, Shubham Paliwal, Monika Sharma, Lovekesh Vig, and Gautam M. Shroff. 2024. SmartFlow: Robotic Process Automation using LLMs. *ArXiv* abs/2405.12842 (2024). https://api.semanticscholar.org/CorpusID:269929773

[9] Pengxiang Jin, Shenglin Zhang, Minghua Ma, Haozhe Li, Yu Kang, Liqun Li, Yudong Liu, Bo Qiao, Chaoyun Zhang, Pu Zhao, Shilin He, Federica Sarro, Yingnong Dang, S. Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. Assess and Summarize: Improve Outage Understanding with Large Language Models. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023). https://api.semanticscholar.org/CorpusID:258959521

[10] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[11] Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. 2024. RAP: Retrieval-Augmented Planning with Contextual Memory for Multimodal LLM Agents. *ArXiv* abs/2402.03610 (2024). https://api.semanticscholar.org/CorpusID:267500377

[12] Toru Kobayashi, Kenichi Arai, Tetsuo Imai, Shigeaki Tanimoto, Hiroyuki Sato, and Atsushi Kanai. 2019. Communication Robot for Elderly Based on Robotic Process Automation. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)* 2 (2019), 251–256. https://api.semanticscholar.org/CorpusID:199014976

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[14] Zheng Li, He Zhang, Liam O'Brien, Rainbow Cai, and Shayne Flint. 2013. On evaluating commercial cloud services: A systematic review. *Journal of Systems and Software* 86, 9 (2013), 2371–2393.

[15] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983* (2023).

[16] Robert Müller, Ulrike Greiner, and Erhard Rahm. 2004. AGENTWORK: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.* 51 (2004), 223–256. https://api.semanticscholar.org/CorpusID:2512160

[17] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[18] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. 2009. Cloud computing: An overview. In *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*. Springer, 626–631.

[19] Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. 2023. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541* (2023).

[20] Milla Ratia, Jussi Myllärniemi, and Nina Helander. 2018. Robotic Process Automation - Creating Value by Digitalizing Work in the Private Healthcare? *Proceedings of the 22nd International Academic Mindtrek Conference* (2018). https://api.semanticscholar.org/CorpusID:53037738

[21] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring llm-based agents for root cause analysis. *arXiv preprint arXiv:2403.04123* (2024).

[22] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2024).

[23] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2024).

[24] Rehan Syed, Suriadi Suriadi, Michael Adams, Wasana Bandara, Sander J. J. Leemans, Chun Ouyang, Arthur H. M. ter Hofstede, Inge van de Weerd, Moe Thandar Wynn, and Hajo Alexander Reijers. 2020. Robotic Process Automation: Contemporary themes and challenges. *Comput. Ind.* 115 (2020), 103162. https://api.semanticscholar.org/CorpusID:211061438

[25] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509* (2022).

[26] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[27] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864* (2023).

[28] Rui Xiao, Wen-Cheng Ma, Ke Wang, Yuchuan Wu, Junbo Zhao, Haobo Wang, Fei Huang, and Yongbin Li. 2024. FlowBench: Revisiting and Benchmarking Workflow-Guided Planning for LLM-based Agents. *ArXiv* abs/2406.14884 (2024). https://api.semanticscholar.org/CorpusID:270688713

[29] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[30] Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, et al. 2023. A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *arXiv preprint arXiv:2303.10420* (2023).

[31] Yining Ye, Xin Cong, Shizuo Tian, Jian Cao, Hao Wang, Yujia Qin, Ya-Ting Lu, Heyang Yu, Huadong Wang, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2023. ProAgent: From Robotic Process Automation to Agentic Process Automation. *ArXiv* abs/2311.10751 (2023). https://api.semanticscholar.org/CorpusID:265295561

[32] Xuchao Zhang, Supriyo Ghosh, Chetan Bansal, Rujia Wang, Minghua Ma, Yu Kang, and Saravan Rajmohan. 2024. Automated Root Causing of Cloud Incidents using In-Context Learning with GPT-4. *arXiv preprint arXiv:2401.13810* (2024).

[33] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854* (2023).

[34] Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Ningyu Zhang, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, and Huajun Chen. 2024. KnowAgent: Knowledge-Augmented Planning for LLM-Based Agents. *ArXiv* abs/2403.03101 (2024). https://api.semanticscholar.org/CorpusID:268248897