LLexus: an AI agent system for incident management

Pedro Las-Casas, Alok Gautum Kumbhare, Rodrigo Fonseca, Sharad Agarwal
Microsoft

ABSTRACT

When operating a software service on a cloud, the complexity of keeping multiple distributed components responsive is a significant challenge for engineering teams. Engineers frequently rely on Troubleshooting Guides (TSGs) to navigate how to mitigate performance or outage incidents. However, the effectiveness of TSGs is often hindered by their length, implicit reliance on tribal knowledge, and the variable quality of their content. This paper introduces LLexus, an agent-based AI system to automate the execution of TSGs.

LLexus employs Large Language Model (LLM) agents to transform TSGs into precise, executable plans during a planning phase. Those plans are then executed when an incident occurs. LLexus primarily comprises of an interactive planner and an executor. The planner aids engineers in refining TSGs into detailed plans represented as flowcharts, delineating tasks and decision points. The executor then autonomously carries out these plans, requiring human assistance only for tasks that require physical actions such as replacing equipment. Through a series of use cases, we demonstrate the feasibility and preliminary effectiveness of LLexus, showcasing its potential to streamline the incident management process.

Our key findings highlight LLexus's ability to improve the readability of TSGs. The system's use of LLMs in the planning phase ensures a reduction in both LLM-related costs and the impact of potential errors in execution.

1 INTRODUCTION

Operating large cloud systems is an extremely complex task. Such systems have hundreds of components, built and operated at different layers, evolving independently by different teams and even companies, and operate in multiple, changing environments. Despite communal best practices, and a continuous effort to incorporate lessons from failures into systems and processes, there is a constant stream of incidents that have to be addressed by teams of on-call engineers.

To reduce the time to mitigation (TTM) of incidents, it is common practice for on-call engineers to rely on troubleshooting guides (TSGs), which are documents that detail steps to identify the causes and address the symptoms of the problem.[1] At Microsoft, for example, a recent study by Shetty et al. [27] documented over 50,000 TSGs in use by more than 60,000 engineers. The study also found that some TSGs are used in hundreds of incidents, and that there is a strong correlation between incidents for which there is a TSG and the TTM of those incidents.

TSGs are by their nature quite prescriptive, having the structure of a flowchart that intersperses instructions, the use of tools, examples, and decision points. They can also be cross-referenced, with some TSGs referring to other TSGs as sub-steps. Automating TSGs has been a long goal of many research and commercial projects [27, 29, 30]. Some proposals try to extract programs directly from TSGs, while others advocate writing TSGs as code (*e.g.,* notebooks [27].) Writing TSGs directly as code could potentially work, but raises the bar for writing, testing, and maintaining TSGs significantly, to the point that understaffed teams of on-call engineers might abandon the approach after all.

Automating TSGs also has significant problems, because TSGs are written in natural language, and assume tribal knowledge among the engineers that does not need to be explicitly written. They are also frequently poorly written. Shetty et al. [27] found that there was significant variation in quality among the more than 4,000 TSGs they examined in detail, with problems including incomplete information, broken links, and incorrect or missing information. We found similar problems in our own analysis of the complete set of 54 TSGs used by a particular team at Microsoft (§ 2), which can severely limit their effectiveness in helping on-call engineers mitigate incidents.

The recent advent of Large Language Models (LLMs) offers a potential direction to tackle these problems, as they have been very successful in natural language tasks [4], and, to some extent, in simple reasoning and planning [32]. In this paper we explore the use of LLMs to help engineers improve existing TSGs and craft new ones, and take initial steps in using LLMs to automate their execution.

An initial naïve approach to automate troubleshooting is to simply prompt an LLM to solve a particular incident using a TSG. In our exploration, however, this did not work reliably for a few different reasons. First, TSG quality posed a significant barrier. Issues included imprecise language, too much assumed knowledge (including ill-specified tools), inconsistent or missing inputs and outputs to different steps, and outdated information. In summary, even a technically capable person, but not trained in the specifics of the service in question, would probably fail in carrying out the instructions in the TSG.

Second, LLMs have variable outputs and are prone to hallucinations. A team would be hard-pressed to trust them to automatically attempt troubleshooting. A co-pilot [30], or a chat-based approach may mitigate this concern, with the LLM proposing investigative and remediation steps individually to an engineer and seeking their approval on each.

[1]Different organizations have different names for essentially equivalent documents, such as run books, playbooks and MOPs.

However, TTM remains limited by human attention. Lastly, even if all of these problems are solved, there is another aspect to be taken into account, which is the cost of LLM calls, which can increase significantly with increase in number of incidents.

In this paper we describe a different approach, in a system we call LLexus. LLexus aims to automate the execution of TSGs by using LLM agents to produce executable plans from a source TSG. Instead of having the LLM read and try to execute the TSG at the time of an incident, we use the LLM in a planning phase where, in an iterative fashion, the LLM agent assists a human to improve the TSG. The goal of this phase is to have enough information in the TSG such that it can be converted into a precise, executable plan. This plan, represented as a flowchart, has nodes representing tasks (e.g., "obtain the IP address of the failed network interface"), and edges representing execution flow and data flow between tasks. Execution of this plan does not require an LLM, as tasks are represented in sequences with simple branching instructions. The tasks themselves can still be complex, and can be a combination of existing tools (*e.g.,* shell scripts), non-planning LLM tasks (such as extracting an identifier from a log), or even a human action, such as replacing a network interface.

This approach has a few advantages. First, it results in improved TSGs for both human use and automatic execution. Second, by front-loading the use of LLMs, with a human in the loop, it both reduces LLM costs and mitigates the impact of hallucinations and non-deterministic execution.

In the remainder of the paper we describe LLexus's approach and components in detail. The interactive planner is a multi-step, LLM-based tool engineers use to both improve a TSG and produce an executable plan. The executor then executes the plan with little to no supervision. We demonstrate the feasibility and preliminary effectiveness of LLexus through three use cases from real TSGs.

2 BACKGROUND: TSGS AND INCIDENT MANAGEMENT

In most SaaS (Software-as-a-Service) engineering organizations, when an outage or performance regression occurs in a production system, an *incident* is generated and registered in an incident management system.

There are essentially two types of incidents: human-generated incidents, which are reported by an engineer or customer, and machine-generated incidents, which are created by automatic monitors that detect one or more metrics violating specified thresholds.

Incidents are classified by severity, either automatically by rules in machine-generated incidents, or manually by the human generating an incident. The number of severity classes varies, but is typically a few different classes. Machine-generated incidents will typically include a lot of relevant information and the metric(s) violating acceptable thresholds. There is typically a direct correlation between a type of machine-generated incident and the relevant TSG, such as the same name of an issue being used in both the incident title and the TSG title (e.g., network interface down). Human-generated incidents tend to be more unique, including descriptions of specific scenarios and steps to reproduce the problem, and an engineer may have an additional step of having to find appropriate TSGs to help with the mitigation. Prior work [12] has explored mapping incidents to TSGs. In our work we rely on existing mapping in machine-generated incidents, and focus on the execution of these TSGs.

A TSG (troubleshooting guide) is a human-readable document that is followed by an engineer to investigate and resolve an incident. It will typically contain multiple steps, some of which may require executing commands (such as retrieving logs), and some of which may depend on the result of a previous step (such as following one or the other path of investigation based on results of a step). It is usually written when repeated occurrences of a problem are observed in a running system.

The goal of a TSG is to help the engineer reduce MTTM (mean time to mitigate). As one set of examples, TSGs for the Microsoft Azure Service Fabric are publicly available [21], and prior work [27] describes TSGs in more detail. There are other documents that are similar in nature, such as MOPs (management operation), run books, and playbooks [29, 30]. LLexus is agnostic to whether the input document is a TSG, MOP, run book, or playbook, and hence for brevity we simply use the term TSG to refer to any such document.

A TSG may be followed by certain engineers, such as an SRE (site reliability engineer), DRI (designated response individual), or on-call engineer. For brevity, we will simply refer to them as engineers. The first priority of the engineer is to mitigate the incident, restoring the systems to operating conditions. Finding the root cause of the incident is usually a separate process, that can take longer than mitigation, and can result in deeper actions to correct the problem, such as bug fixes and deployment changes. While an incident is being resolved, engineers will annotate the incident log with notes on what diagnosis and resolutions steps were followed and what the outcomes of those steps where. Such notes aid post-incident analysis, or even transfer responsibility from one engineer to another for long-running incidents.

LLexus is being deployed for a Microsoft hybrid-cloud SaaS (software-as-a-service) product. There are numerous reasons why the engineering team for this SaaS product considers LLexus to be useful, including:

(1) Their TSGs are lengthy, requiring significant time for an engineer to mitigate incidents.
(2) Their TSGs undergo frequent changes, hence would require too much engineering effort to write code to automate.
(3) A high frequency of incidents is resulting in high MTTM, stress for engineers, and increased spend on on-call engineers.

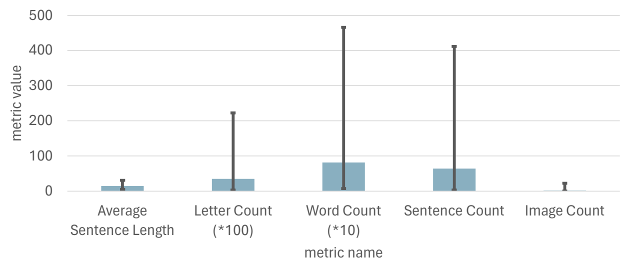


Figure 1: Length of 54 TSGs. Bar shows median value, and error bars show 5th and 95th percentiles.

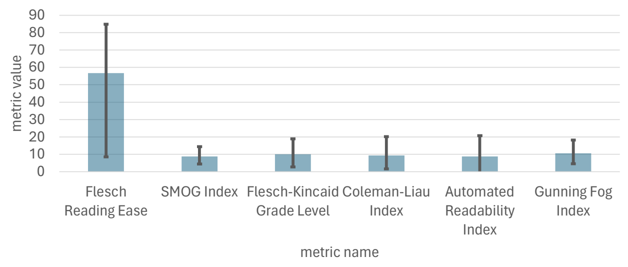


Figure 2: Readability of 54 TSGs, as defined by common readability metrics. Bar shows median value, and error bars show 5th and 95th percentiles.

2.1 Examining TSGs

We characterized the 54 TSGs that the engineering team relies on. Our observations are aligned with previous studies [27]. Figures 1, 2, and 3 shows our results. Figure 1 shows that the median length of a TSG is 815 words, with the largest is at over 5500 words. When a live service has an outage with impacted customers, reading through a highly technical document of 5000 words is time consuming and stressful.

Complicating the problem further is that these TSGs are highly technical documents that are not easy to read. Figure 2 shows the distribution of these TSGs across a variety of well known readability metrics. The graph shows that these TSGs are far from casual reading, some requiring as much as 20 years of education to understand as measured by the Flesh-Kincaid Grade Level metric. Hence, it takes longer for an engineer to read and understand a TSG, and requires a more sophisticated language model to understand.

Figure 3 shows the frequency of edits to these TSGs as measured by their Git commit history. The median number of updates is 8, but some have been updated over 60 times. At the median, TSGs are updated at a frequency of 19 days, but some are updated almost constantly as the product improves and its behavior changes, or more reasons behind a persistent problem are uncovered and documented. In a stable product that is not undergoing any code change, TSGs may not see any changes, and hence it may be worth the engineering effort to automate TSGs by writing code to replicate what an on-call engineer would do. However, in a young SaaS product undergoing rapid innovation, updating a human-readable TSG multiple times a week is a lot easier than writing new or updated code every week for it.

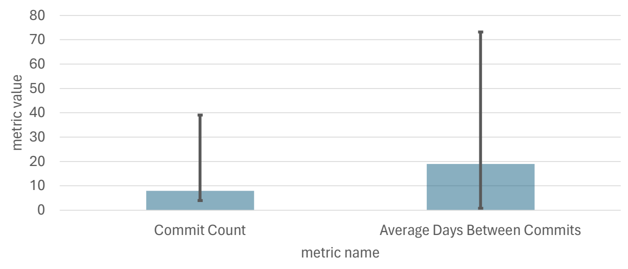


Figure 3: Commit history of 54 TSGs. Bar shows median value, and error bars show 5th and 95th percentiles.

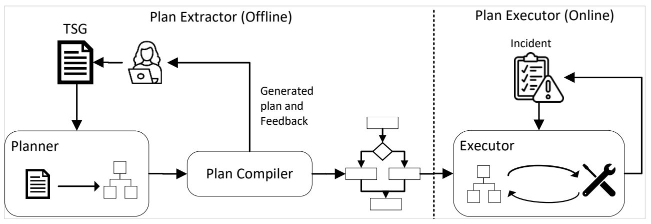


Figure 4: Overview of LLexus.

3 OVERVIEW OF LLEXUS

Figure 4 presents a high-level overview of LLexus. There are two main components: an offline component called Plan Extractor (or planner) and an online component called Plan Executor (or executor). The Planner is an LLM-based AI agent that analyzes a TSG and produces an executable plan. This process is triggered when a TSG is created or modified. It requires as input the TSG and a set of tools available to execute the plan.

The Executor then executes the plan produced by the Planner against an incident. It is activated everytime a new incident occurs. It first fetches the plan relevant to the incident, then retrieves all the required information from the incident and follows the steps described in the plan, taking the required actions, executing the defined tools and makes decisions until the investigation or mitigation of the incident is complete. It annotates the incident log with actions taken and decisions made, which allows for post-incident auditing.

In addition LLexus cosists of a Plan Compiler and a Plan Validator that ensures that the generated plan is complete and executable.

We made the following design choices in LLexus:

- Execution plans are generated at the time of TSG creation or update, not when resolving an incident. This design choice supports two goals – (a) reduce LLM usage cost by generating a plan for a TSG once, instead of every time a relevant incident occurs; and (b) reduce MTTM by relying on a pre-generated plan when addressing an incident.

- Plans are iteratively generated. The AI agent first provides a high level plan, and subsequently fills in the details of each step in the plan, such as what tool to use and/or what inference to make (see § 4). This design choice allows for a more detailed plan while limiting LLM hallucinations.
- Engineers are expected to audit a plan when it is generated. The comprehensiveness and accuracy of a plan are important for the correct mitigation of an incident. Auditing such a plan the moment a TSG is authored allows an engineer to correct inadvertent mistakes or omissions in their TSG. This also limits the scope of LLM hallucinations.
- The TSG is the source of truth. If there are issues with the plan, the engineer edits the TSG and reruns the planner, rather than editing the plan. This process can happen multiple times, and results in consistently improved TSGs for both human and LLEXUS consumption.
- Plan execution relies on existing, specific tools that the engineering team has. Typically, there will be several tools, such as log retrieval or reboot a host. Without such specific tools, LLEXUS would have to rely on generic tools such as SSH, which would give the system unrestricted access to production systems. Relying on these existing, specific tools supports two goals – (a) reduce LLM usage cost by not requiring an LLM to manipulate SSH sessions to perform such actions; and (b) reduce the scope for hallucinations by relying on such deterministic tools. Any actions for which tools do not exist could be performed by an LLM, and typically this would be analysis of a tool output that an engineer would otherwise do.

We will use the following example to detail the functioning of the Planner and Executor and present the characteristics of an executable plan. It is a simplified version of a TSG that can be used to troubleshoot a web server with slow response time. Here, we assume the existence of four tools: *retrieve application metrics*, *identify impacted server*, *retrieve resource usage* and *scale up/down server*. Note that this is a toy example used only for didactic purposes to present the concepts defined in LLEXUS, hence not intended to be a complete TSG.

TSG: Investigating and Mitigating Latency Issues in a Web Server

Users are experiencing slow response times when accessing the web server.

(1) Identify the web server
    - Determine the specific web server experiencing latency issues in the description of the incident.
(2) Fetch latency metrics for that web server
    - Retrieve CPU usage in the server
(3) Check if the CPU usage is above 60%. If not, and issue still persists escalate to check other bugs/issues.
(4) If above 60%, scale up the number of servers.
(5) Wait for an hour and go back to step 2.

Available tools:

- Application metrics tool: Retrieve end-to-end latency
- Identify impacted server tool: Extract server details from app metrics
- Resource metrics tool: Retrieve CPU usage
- Admin tool: Scale up/down server

3.1 Planner

As described earlier, the quality of TSGs varies significantly and hence even with the best LLM agent, it may not be possible to extract an executable plan. To improve quality, the Planner is run in rounds with a "human in the loop". Within each round, the planner uses multi-step plan generation (described in detail in §4), which defines the necessary steps for the plan and the details required for execution of each step. After each round, an engineer has the ability to update the TSG based on the generated plan and the feedback from the planner. This process helps not only to produce a comprehensive executable plan but also to improve the quality of the TSG for human consumption.

Figure 5 shows the plan generated from the example TSG.

3.1.1 Plan. The plan is represented as a flowchart (inspired from BPMN [9]). It a directed graph where each node represents a step and the solid edges represent the flow of execution. It also consists of data flow edges that represent the flow of data between steps (dashed edges). It consists of three different types of steps: action, conditions, and events.

- **Action steps** involve the execution of a tool and are always followed by only one next step. In Figure 5, the rectangular nodes describe the action steps. As can be seen, each node has an associated tool and receives the required parameters (*e.g.,* incident ID, incident details).
- **Condition steps** require a decision to be made to determine the investigation path to follow. This step can have two or more next steps, each with an expression

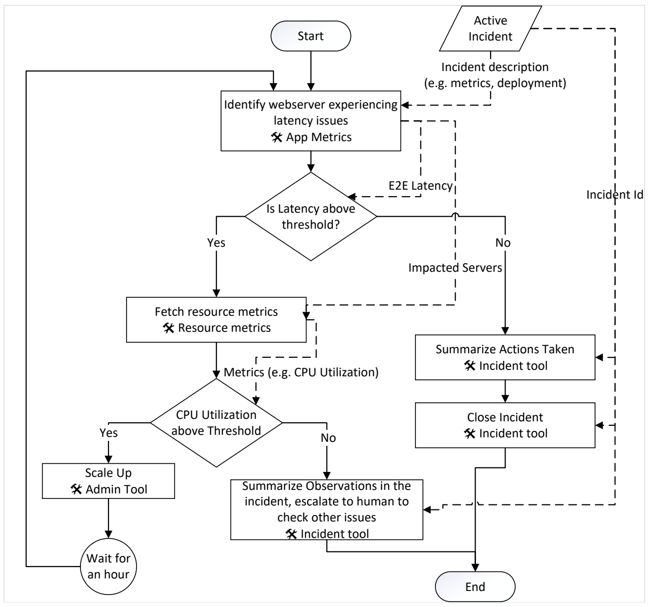

Start
Active Incident
Incident description (e.g. metrics, deployment)
Identify webserver experiencing latency issues
App Metrics
E2E Latency
Incident Id
Is Latency above threshold?
Yes
No
Impacted Servers
Fetch resource metrics
Resource metrics
Summarize Actions Taken
Incident tool
Metrics (e.g. CPU Utilization)
CPU Utilization above Threshold
Close Incident
Incident tool
Yes
No
Scale Up
Admin Tool
Summarize Observations in the incident, escalate to human to check other issues
Incident tool
Wait for an hour
End

Figure 5: Web server latency issue investigation plan.

that needs to be evaluated for plan execution to proceed along this path. They are represented as the diamond shaped nodes in Figure 5. In this example, the node has two possible next steps, based on the condition to be evaluated during execution (CPU Usage > 60 or CPU Usage ≤ 60).

- **Event steps** can be of two different types: *external actions* and *timer*. *External actions* are the ones in which the step will wait for a manual action or the receipt of a message before the plan is resumed, while *timer* events pauses for certain time before it continues the execution. The example plan (circle node) shows a timer *timer* event. This step makes the execution wait for one hour before resuming.

All steps must have at least one next step, with the exception of the final step (in this case it is *Close Incident*). Note that cycles in the plan are allowed. In the example above, we have a cycle in which after scaling up the server, we wait for an hour and then go back to a previous step to re-fetch the latency data and evaluate whether CPU usage has returned to an acceptable level.

3.1.2 *Tools.*

When extracting the details for each step, the planner defines the tool that will be used for its execution as well as the inputs and outputs of that step. This is essential for the executing the plan. The plan supports two types of tools: (i) script tools and (ii) semantic tools.

(i) Script tools refer to conventional programs that can be executed without the need for a language model. These tools encompass a wide range of functionalities, including but not limited to Python scripts, PowerShell scripts, Azure commands, and Log queries. They can be executed directly by the system without the need for additional interpretation. Script tools offer flexibility and efficiency in automating tasks, making them indispensable components in various domains where precise execution of predefined actions is required. Script tools are further categorized into two types: (a) *Retrieval tools* that are used to fetch data from external sources, and (b) *Action tools* that are used to perform actions on the system.

(ii) Semantic tools relies on the understanding and generation capabilities large language models. These tools encompass a variety of tasks, from natural language processing to complex reasoning and inference. For instance, they can analyze the output from script tools and extract relevant information from logs or help make routing decisions. Semantic tools are key for handling tasks that involve ambiguity, context sensitivity, or require a deeper understanding of language semantics.

In the previous example, *Resource metrics*, *Admin* and *Incident* are script tools, while *Identify Web Server* is a semantic tool that will receive the incident description and metrics, and use an LLM to extract the information about the server that is presenting latency issues.

3.1.3 *Plan Compiler.*

Plan compiler is responsible to validate that the generated plan is complete and executable. The compiler analyzes the plan and ensures that the plan follows a set of predefined rules that are required for the execution. For example, some of the rules are (i) ensure that all action steps have a defined tool to execute the action; (ii) ensure that all steps in the plan are reachable; (iii) validate each expression provided in the condition nodes and (iv) make sure that the input required for execution of each step can be retrieved, either from a previous step, the incident or has a predetermined value. While these rules are essential to ensure that the plan is executable, they also help provide feedback to the engineer to improve the TSG as these same issues will faced by the engineer following the TSG, even without automation. Hence the Plan Compiler plays a crucial role in improving the quality of the TSGs.

3.2 Executor

The Executor is the online component of LLexus. This component is triggered every time a new incident arrives. The first step is to retrieve the plan related to the incident. Then, the executor follows each step described in the plan. The plan aims to extract the most deterministic set of steps possible to investigate and mitigate the incident. This removes much of the responsibility of the Executor in terms of decision making when executing the plan. In the end, all it does is to blindly follow each step defined by the plan.

The executor handles all the infrastructure and authentication aspects required. As mentioned, we support tools like Powershell scripts, Kusto queries, Python scripts, among others. That requires different resources, which is all handled by the executor.

A key aspect of the executor is the ability to run semantic tools for steps that require reasoning or understanding of the context. These are tasks that would have otherwise been completed by an engineer and require basic understanding (e.g. understanding exceptions in logs, or interpreting the results of a query). Given the complexity of these tasks, the executor uses a large language model to perform these tasks. This is an important aspect of the executor, as it allows the system to handle a wide range of tasks that would otherwise require human intervention. The use of LLMs is limited within a given step, and not across steps. This is to ensure that the executor remains deterministic and does not introduce non-determinism in the execution.

Further, it is important to note that some incident mitigations require manual intervention (external events) or take long time to mitigate. Hence it is critical for the executor to be stateful, and be able to pause the execution and resume it later. We achieved this through use of Durable Functions [17].

3.2.1 Deterministic execution of a plan. Previous works [23] present approaches to investigate an incident and determine its root-cause by using a ReAct [35] agent that reasons about the incident and which tools to use during the investigation. While being an interesting and useful approach, this comes with downsides. The nondeterministic execution might lead to unexpected paths during the investigation, which leads to incorrect answers and significantly increase the cost of execution. Since it requires reasoning before each action to be taken, including the context, which increases as the investigation goes on, the costs needed might end up reaching a prohibitive level when considering large and complex services with large number of incidents.

In this work, we focus on extracting a deterministic execution plan, that will largely reduce the cost of executing it. In that case, the majority of the cost will come from extracting the plan from the TSG, since it requires multiple steps and iterations to achieve an optimal and deterministic plan. However, the cost during the execution phase is minimal, since the actions to take are previously know. While great for optimizing cost, it is important to note that this approach can only be applied to known issues that have a well-defined set of steps to be taken. Unexpected issues might still benefit from approaches like the one proposed in [23].

4 MULTI-STEP PLAN GENERATION

A key part of LLexus is the iterative plan generation process. Figure 6 presents the architecture for this phase. The plan generation benefits from the understanding and reasoning abilities of large language models. It is composed by four LLM-based components which we describe in this section, including a validation component that is triggered after the completion of each of the other steps.

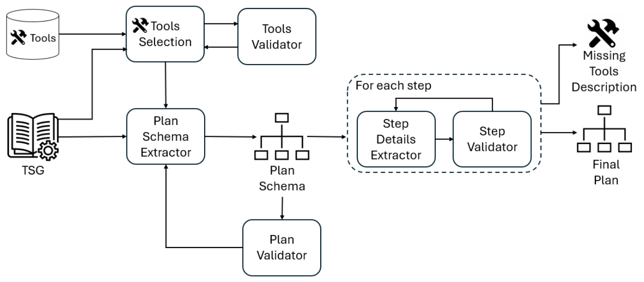


Figure 6: Planner architecture

4.1 Tool Selection

Our experiments have shown that providing a set (or a superset) of tools to the plan schema extractor step (see next section) significantly improves the quality of the generated plans. Hence, the first step in the plan generation process is tool selection.

As the name suggests, this component is used to select the tools that are required during the execution of the plan. It analyzes the given TSG and a full set of available tools to determine which tools are relevant for the task at hand. The component is guided by a prompt that provides the LLM with the necessary context to understand the TSG and the tools. Here we leverage advance prompting techniques such as chain-of-thought [34] to guide the LLM in selecting the tools.

The prompt aims to guide the LLM's comprehension of the TSG and each associated tool. Based on this semantic understanding, the prompt maps tools that are relevant to the tasks proposed in the TSG. The result is a JSON object parsed by the Planner and stored for downstream use.

Given that the LLM semantically evaluates the tools to determine their relevance to the TSG, it is crucial to provide detailed descriptions of each tool. Therefore, we expect the description for each tool to include the following details. In most cases, this can be obtained from the tool's documentation.

- Descriptive Name
- Detailed description of tool and what is its purpose
- Input Parameters
  - Parameter name
  - Detailed description of the parameter
  - If parameter is required
- Outputs
  - Output name
  - Detailed description of the output

Following tool selection, a validation step ensures the selected tools are appropriate. This "validation" or "double-checking your work" is key to avoid issues due to hallucinations from the large language models, as the LLMs have been shown to be more accurate with validation and self-improvement [11, 28] (arguably, this is still an open question [31] and may not generalize, but we have found it to be effective in this case as the TSGs are fairly well-structured by design). The validation steps checks the selected tools and confirms the completeness and correctness of the tool list, returning a flag indicating the validity of the response from the tool selector component. If the response is invalid, the reasons for the failure and potential fixes are provided. The component is then re-executed with this additional context, continuing until a valid response is generated or the execution limit is reached. The feedback is then provided to the TSG writer for further review and corrections to the TSG.

4.2 Plan Schema Extractor

After selecting the tools for executing the TSG, the next step involves creating the plan. Initially, we construct the high-level plan schema, outlining the plan's structure with nodes and their relationships. Detailed steps, including tool mapping and inputs, are defined in subsequent stages.

This process is iterative, leveraging the LLM to generate and validate the plan until a comprehensive version is achieved. In the initial part of the prompt, we provide the LLM with knowledge of the plan's structure and characteristics to ensure the generated response aligns with our specifications for an executable plan. This knowledge serves as context for the prompt. In the second part, we specify instructions for extracting the plan from the TSG. These instructions guide the LLM to understand how the TSG addresses the incident and extract self-contained steps covering all possible paths for investigation and mitigation.

This initial version of the plan includes the description of each step, the type of step (Action, Condition or Event), and the subsequent steps. For conditional steps, each condition expressions are also defined for deciding the execution after evaluating the condition.

Following the plan schema generation, a validation stet ensures that the specifications of the plan are correctly followed. For example, action nodes should have only one next step, and condition steps should have a valid expression for each next step. Additionally, it ensures that the plan is complete, following all the possible paths described in the TSG, and that each step is self-sufficient and can be executed.

Similar to the previous process, the validation and update is repeated until it gets a valid response or no further updates can be made from available information, in which case a set of feedback is shared with the TSG writer to make improvements and retry. It is worth noticing that for each re-execution, the previous plan, the reasons why it was rejected, and possible fixes are provided as context to the plan schema extractor.

4.3 Step Details Extractor

Now that we have the high-level plan schema, the next step is to extract the details for each step within the plan. These details include specifying which tool to use for executing the step, identifying the inputs required for that step and their sources (whether from a previous step, the incident, or pre-defined values from the TSG), as well as defining the outputs of the step.

The process of extracting these details involves providing the TSG, the list of available tools for executing the step, the high-level description of the step extracted by the previous component, and the path of the plan up to the current step. Within this provided plan, all steps already have their details such as tools, inputs, and outputs.

With this information provided, we guide the LLM to understand the requirements of each step, evaluate all available tools, and select the most relevant one. In cases where a relevant tool cannot be retrieved from the provided list, the LLM is instructed to propose a new tool for use in that step. It can suggest both script-based tools and semantic tools. When proposing a new tool, the LLM describes the tool's purpose, provides the list of inputs and outputs, and for semantic tools, it already provides the required prompt.

The details generated for each step are also subject to validation. During validation, we verify the relevance of the selected tool for the step and, most importantly, ensure that the inputs can be sourced from previous steps, the incident, or have pre-defined values. This validation is critical for ensuring plan is executable deterministically.

4.4 Validation

As mentioned, the validation component is crucial and utilized in all three of the other components of the Planner. Its purpose is to ensure that responses generated by each component align with the expected goals, as described previously. To achieve this, we have developed a general approach of self-reflection and correction [28] that instructs the LLM on how to validate a given response, against a set of goals. This component takes as input the response being validated and the specific set of goals for that validation, as well as any additional context or hints. For example to validate the plan generated by plan schema extractor, the goals would be to ensure that all steps are self-contained and that the plan is complete. It takes as input the original tsg, the goals of the planner, the response generated by the plan schema extractor, and the output of the plan compiler (which is a set of predefined rules as described earlier) as the context that guides the validation and self-improvement.

5 IMPLEMENTATION

LLexus is built on the Azure stack, with the core service deployed using Azure Functions and implemented using Python, providing REST APIs as entry points for both the Planner and Executor modules.

The **Planner** module comprises three Azure Functions, forming an asynchronous process for plan generation. This asynchronous design ensures scalability, allowing the generation process to scale without being limited by the size or number of steps in the TSG. These functions handle initiating the generation process, executing the plan generation, and retrieving the final plan. Orchestration between these functions is managed using Azure Queue and Azure CosmosDB.

During plan generation, Semantic Kernel [20] serves as the framework integrating models with the underlying code needed to process the TSGs. For the LLM component, LLexus uses the GPT-4-Turbo model accessed via the Azure OpenAI API. Each component of the Planner makes one request to the LLM and processes it accordingly. The total number of LLM requests for a Planner execution is directly related to the number of steps in the plan.

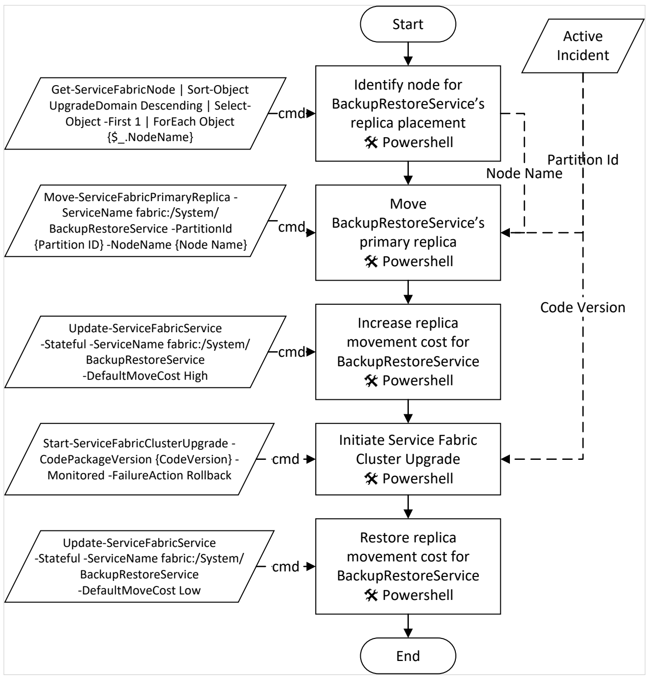
The final output of the **Planner** is a JSON object that defines each step, their relationships, and all the necessary details for executing each step. **Executor** uses this object along with the incident as input to execute the plan.

Executor is implemented using Azure Durable Functions [17], allowing for stateful execution, event handling and pause/resume, which are essential for maintaining the state of each step, especially to account for cases where the incident resolution depends on external events. One of the key goal of the executor is to seamlessly integrate automated and manual steps and continue execution without human intervention. Further, this state can be utilized in subsequent steps. Each step in the plan triggers a function that processes by collecting all the parameters required for the tool from a state store, invoking the appropriate tool (such as Kusto queries [18], Powershell scripts [19], Azure commands [16], among others.) and storing the results back into the state store for subsequent steps. For semantic tools (like understanding logs or deriving certain parameters from previous steps outputs), it leverages Semantic Kernel to execute LLMs with specified prompts specific to those tools.

6 EVALUATION

LLexus is still in early phases of development and deployment, as as such we do not yet have a comprehensive evaluation. Ideally, we would demonstrate effective reduction in MTTM and on-call engineer hours, and a substantial improvement in the quality of a large body of TSGs. These are our longer-term evaluation goals.

In this section, instead, we demonstrate the effectiveness and practical application of LLexus through three distinct case studies. The first two case studies involve the generation of execution plans for public troubleshooting guides from Azure Service Fabric. The final example presents a more intricate scenario from an internal Microsoft service. These case studies offer insights into how LLexus can streamline both the process of iterating and improving the troubleshooting guides and the process to automate the execution of these guides. Additionally, we provide a cost analysis to evaluate the efficiency of the proposed approach of generating executable plans using LLexus.


Start
Active Incident
Identify node for BackupRestoreService's replica placement
Powershell
cmd
Node Name
Partition Id
Move BackupRestoreService's primary replica
Powershell
cmd
Code Version
Increase replica movement cost for BackupRestoreService
Powershell
cmd
Initiate Service Fabric Cluster Upgrade
Powershell
cmd
Restore replica movement cost for BackupRestoreService
Powershell
cmd
End

```
Get-ServiceFabricNode | Sort-Object UpgradeDomain Descending | Select-Object -First 1 | ForEach Object {$_.NodeName}
```

```
Move-ServiceFabricPrimaryReplica -ServiceName fabric:/System/BackupRestoreService -PartitionId {Partition ID} -NodeName {Node Name}
```

```
Update-ServiceFabricService -Stateful -ServiceName fabric:/System/BackupRestoreService -DefaultMoveCost High
```

```
Start-ServiceFabricClusterUpgrade -CodePackageVersion {CodeVersion} -Monitored -FailureAction Rollback
```

```
Update-ServiceFabricService -Stateful -ServiceName fabric:/System/BackupRestoreService -DefaultMoveCost Low
```

Figure 7: Generated plan for Case study: Fabric Upgrade fails.

6.1 Case study: Sevice Fabric - Fabric runtime upgrade fails due to BackupRestoreService

For our first case study, we used the troubleshooting guide from Azure Service Fabric, which offers guidance on addressing runtime upgrade failures in fabric clusters when BackupRestoreService is enabled[2]. This troubleshooting guide outlines a series of steps involving Powershell commands that users can follow to resolve upgrade issues. In summary, it describes the process of identifying a node to relocate BackupRestoreService's replica so that it is the last node to be upgraded, moving the service to that node, increasing the cost of replica movement to prevent further movement, restarting the upgrade, and then restoring the replica movement cost. Despite its simplicity, this example serves to evaluate the quality of the generated plans.

The process of generating this plan begins by inputting the troubleshooting guide and specifying *Powershell* as the tool. The resulting plan is depicted in Figure 7. This plan was generated in a single pass, without the need of any user intervention to modify or enhance the TSG. This is attributed to the well-written nature of the TSG, with clear and well-defined steps.

In the original TSG ([22]), specific Powershell commands are provided for step 2 (*Move BackupRestoreService's primary replica*), step 3 (*Increase replica movement cost for BackupRestoreService*), and step 5 (*Restore replica movement cost for BackupRestoreService*). The extraction process for these steps

[2]https://github.com/Azure/Service-Fabric-Troubleshooting-Guides/blob/master/Known_Issues/Fabric6.4Upgradefails.md

correctly identified these commands and incorporated when executing each step. However, steps 1 and 4 do not explicitly mention any commands in the TSG. In such cases, the Step Details Extractor proposed the usage of Powershell commands to achieve the desired actions outlined in the steps. This is possible because the large language model used (GPT-4) possesses knowledge of Azure Service Fabric and Powershell commands, allowing it to recommend appropriate commands. This process can be further enhanced by using a retrieval augmented generation [15] or fine-tunig process to adapt to a specific domain. In instances where no clear action or tool is available or inferred, LLexus may prompt the user to make changes to the TSG to ensure clarity and provide all the required information for execution.

6.2 Case study: Service Fabric - Periodic backups stop for configured backup policies

In the second case study, we again examine another public troubleshooting guide from Azure Service Fabric [3]. This guide addresses issues encountered when periodic backup ceases to function after a runtime upgrade to specific versions. Unlike the previous case study, this troubleshooting guide presents some distinctions that allow us to showcase additional capabilities of the executable plan.

While the preceding case study depicted a linear plan where each step follows sequentially without requiring decisions, this example introduces the concept of potential divergent outcomes based on the results of certain steps. Here, we find not only *conditional* steps but also an *event* step. This particular one serves the function of waiting for 2 minutes for the actions taken to fix the backup to take effect before closing the incident.

Another distinction between the two case studies is that, in the initial case, we required only one iteration to generate the plan. However, in this case, an extra iteration was necessary. After human-in-the-loop validated the generated plan, we updated the TSG to cover missing scenarios, specifically adding instructions on what to do when the investigation was no longer needed (e.g., cluster version different than expected or no issue identified) to avoid wasted dev cycles on false-positives.

During the initial plan generation, the LLM assumed the issue persisted and included a step to escalate the problem, though this was not expected in the specific TSG. To address this, we updated the TSG to clarify this point. We changed the TSG by adding a few lines to the text explicitly mentioning what to do after identifying that cluster version was different than expected or no issue was identified. Consequently, the planner removed the *"Escalate issue"* node (highlighted in red), and added the *"Close the investigation"* node (highlighted in green). However, a side-effect of regenerating the plan was the reordering of some steps because it was not clear in the original TSG which should preceed, as seen in Figure 9b (highlighted in blue). In this case, this reordering is not consequential, hence the TSG writer may accept the plan as is. However, it is key to note that as the troubleshooting steps in the TSG lacked a clear distinction on the order in which these two steps should be executed, and due the non-deterministic nature of large language models, unrelated changes in the TSG can lead to alterations in the generated plan. This highlights the importance of providing clear and well-defined steps in the TSG to ensure the generated plan is accurate and efficient.

Structure of the plan: After observing structural changes in the TSG following the updates, we decided to investigate the determinism of plan structure for a given TSG. To achieve this, we generated 10 plans for the updated TSG. The goal was to compare potential variations in the plans due to the non-deterministic nature of LLMs. Upon evaluating the results, we found that all 10 generated plans exhibited the same structure (as seen in Figure 9b). Although the wording of each step's description might differ between plans, the fundamental concept of each step remained consistent across all plans.

We observe, as expected, that the better written and clearer the TSG, the better the generated plan will be. In particular, ambiguous situations or those that can be inferred based on context may end up affecting the plan obtained by the system. This is beneficial not only for the planner, but also for human consumption. The goal of LLexus is to provide a framework for improving TSGs and achieving better versions for both humans and automation.

6.3 Case study: Unexpected Behavior in Network Link

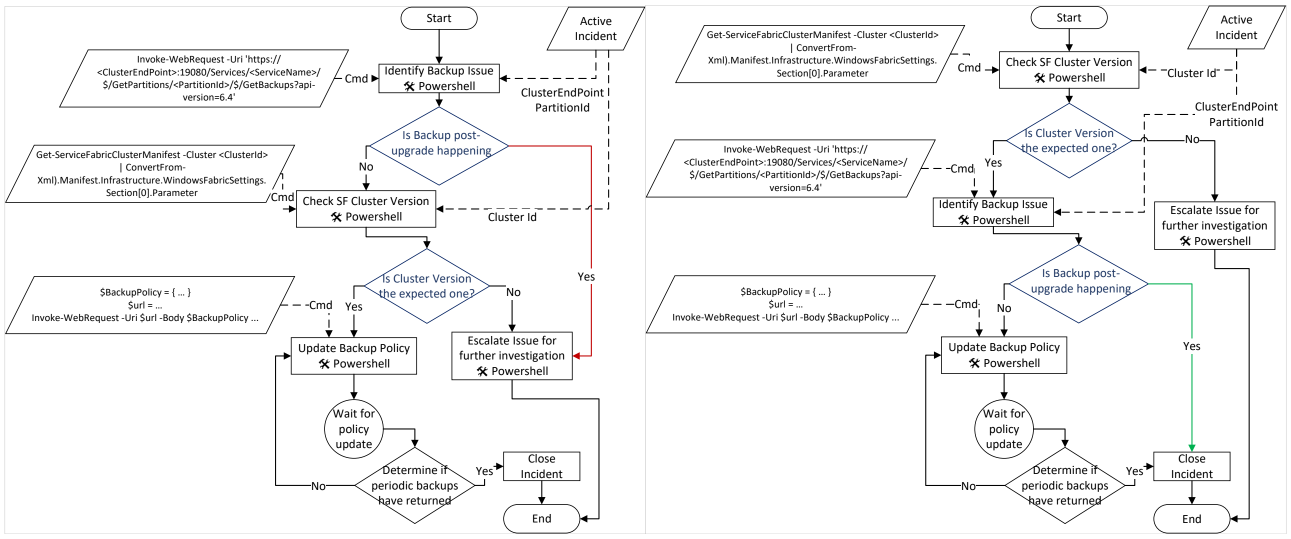
The final case study refers to a troubleshooting guide used for investigating issues within a specific service in Microsoft Azure. Since this is an internal TSG, the details cannot be disclosed.

In essence, this TSG is employed to diagnose unexpected behavior occurring in network links within that specific service. During the investigation, engineers first check if the involved resources are under maintenance, as this may be the cause of the behavior. If not, they proceed by examining metrics and logs to determine the affected resource and escalate the issue to the physical network team for mitigation.

It is worth noting that this particular service is relatively new and continuously evolving. Consequently, its TSGs are also a work in progress and can vary significantly in quality depending on the evolution of the related system components.

As we started our investigation of this scenario, we found that this specific TSG exhibited low quality. The writing quality was poor, and the descriptions and ordering of the steps were unclear to someone without prior knowledge.

[3]https://github.com/Azure/Service-Fabric-Troubleshooting-Guides/blob/master/Known_Issues/BRS-stops-taking-backup-after-upgrading-to-latest-runtime.md


Start
Active Incident
Cmd
Identify Backup Issue 🛠 Powershell
ClusterEndPoint PartitionId
Is Backup post-upgrade happening
No
Yes
Check SF Cluster Version 🛠 Powershell
Cluster Id
Is Cluster Version the expected one?
Update Backup Policy 🛠 Powershell
Escalate Issue for further investigation 🛠 Powershell
Wait for policy update
Determine if periodic backups have returned
Close Incident
End

```
Invoke-WebRequest -Uri 'https://<ClusterEndPoint>:19080/Services/<ServiceName>/$/GetPartitions/<PartitionId>/$/GetBackups?api-version=6.4'
```

```
Get-ServiceFabricClusterManifest -Cluster <ClusterId> | ConvertFrom-Xml).Manifest.Infrastructure.WindowsFabricSettings.Section[0].Parameter
```

```
$BackupPolicy = { ... }
$url = ...
Invoke-WebRequest -Uri $url -Body $BackupPolicy ...
```

(a) Initial plan **(b) Final plan**

Figure 8: Case Study: Backup failure.

Essentially, it required prior familiarity with the service to comprehend and follow it.

Predictably, the plan generated from this initial version was incomplete and lacked major paths. Essentially, the LLexus Planner could only identify the maintenance checks, completely overlooking the crucial aspect of the investigation when the resources were not under maintenance. It only had 12 steps composing it. Based on this, the tool returned the incomplete plan for evaluation by the responsible user. As mentioned in Section 3, this is an iterative process in which users will use LLexus feedback to update and improve the TSG.

After multiple rounds, we significantly improved the TSG, with each step clearly described and having a clear execution order. The result was a complete plan produced by LLexus. The new plan covers all possible paths and each step is clear, self-contained, and contains all the necessary information for execution. The final plan contains 21 steps, 9 more than the initial version. Figure 9 shows a representation of the incomplete plan and the final complete version.

In summary, the process of enhancing the TSG and generating the plan underscored the importance of having a clear and well-written TSG. It also demonstrated the utility of LLexus's iterative process in assisting engineers in improving troubleshooting guides. By providing guidance and identifying possible improvements, LLexus proves to be a valuable tool for refining troubleshooting procedures.

6.4 Cost Analysis

As discussed in Section 3, one of our key design considerations was to generate a plan for the TSG to reduce the overall cost when addressing each incident. While this approach significantly lowers the expenses associated with investigating and mitigating incidents, there are incurred costs during the plan generation process. We adopted an iterative methodology for generating the plan, which involves validating each step along the way. While this iterative approach enhances the quality of the generated plan, it also incurs corresponding cost during the offline phase.

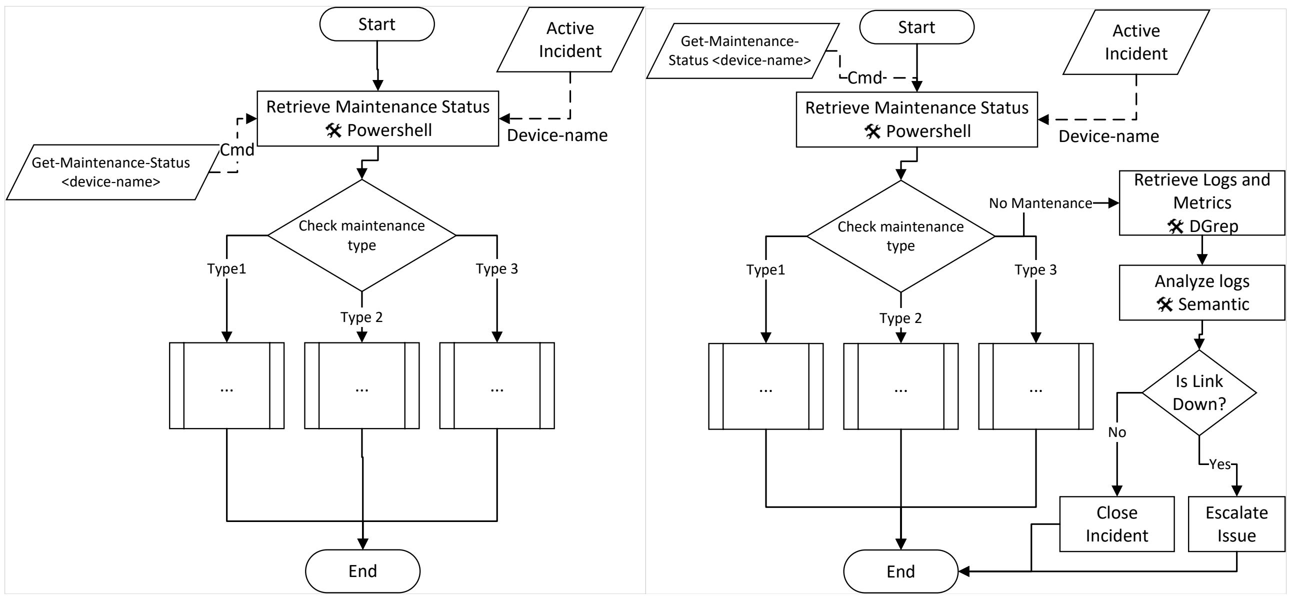
Table 1: TSG: Upgrade Failure

Component		Avg. Input Tokens	Avg. Compl. Tokens	# Exec.	Cost
Tool Selector		1,184	166	1	$0.02
Plan Schema		2,273	390	2	$0.07
Step Details		3,705	435	5	$0.25
Validator		1,520	568	8	$0.26
Total Cost		-	-	-	**$0.60**

To better understand the costs associated with generating each plan and how each component of the LLexus Planner contributes to the overall cost, we conducted an analysis on the costs of generating the public TSGs discussed earlier. We extracted the number of input tokens and completion tokens required for the execution of each component. For our experiments, we utilized the *GPT-4-turbo* model provided by Azure OpenAI. Current prices [4] stand at $0.01 per 1,000 input tokens and $0.03 per 1,000 completion tokens.

Table 1 illustrates the token usage for each component when generating the plan for the Case Study: Fabric upgrade fails. The presented numbers represent the average obtained for the number of executions of each component. Input tokens correspond to the prompt of each component, consisting of a static part that provides instructions to the model for that step, and a dynamic part composed of the TSG, the

[4]Checked on May 5, 2024



(a) Initial plan **(b) Final plan**

Figure 9: Case Study: Unexpected Behavior in Network Link

Table 2: TSG: Backup Stopped

Component		Avg. Input Tokens	Avg. Compl. Tokens	# Exec.	Cost
Tool Selector		1,388	182	1	$0.02
Plan Schema		2,477	904	1	$0.03
Step Details		4,423	461	16	$0.93
Validator		2,238	599	18	$0.73
Total Cost		-	-	-	**$1.71**

plan, and selected tools. Completion tokens refer to what is generated by each component and are significantly smaller than the input sizes. As shown in the table, the total cost for generating the plan was $0.60, with the majority of the cost attributed to extracting the details of each step and validating the responses of each component, accounting for approximately 85% of the total cost.

The second case study (Unexpected Behavior in Network Link) presented in Table 2 incurred total costs almost three times higher. This is due to the higher number of steps in this plan (10 steps compared to 5 in the other case study) and more frequent validation failures. While in the previous plan generation, only the Plan Schema component failed once, in this case, there were six failures during step details extraction. These failures were caused by missing information in step descriptions (such as name or description) or incorrect identification of the source of expected input. Consequently, these steps were considered invalid and had to be re-executed to obtain a valid response.

Through the evaluation of these two case studies, it becomes apparent that the cost of generating a plan is directly influenced by the number of steps required to troubleshoot the incident and the frequency of invalid component executions. Although the number of steps required cannot be easily altered, the validation process can be significantly improved based on the quality of the TSG. Clear and well-described steps in the TSG can directly impact the planner's ability to extract a valid response in the first attempt for each component.

However, compared to a fully online approach that analyses and generates a plan at the time of the incident [23, 35], the cost of generating a plan offline and handling incidents using a deterministic execution, as proposed in LLexus, is significantly lower, especially as the number of incidents increase. Figure 10 illustrates the cost comparison between the two approaches for a single TSG, and illustrates that LLexus is more cost-effective as the number of incidents increases and breaks even after very small number of incidents. This is in addition to other advantages deterministic execution provides, such as faster execution and reduced risk of errors. Here, we assume a cost of $1.15 to process a TSG based on the average cost of the two case studies. We assume similar cost for the online model, per incident processed since it needs to process the TSG each time. Further, both LLexus and the online approach use LLMs to process natural language results, such as logs. We assume $500KB$ of such data being processed incident, and is common across both approaches. The cost of processing $500KB$ of data (50K tokens) using previously stated LLM model is about $0.5.

In conclusion, our analysis highlights that while generating executable plans from TSGs incurs costs, these costs are reasonable considering the significant reduction in expenses during incident execution. As demonstrated through

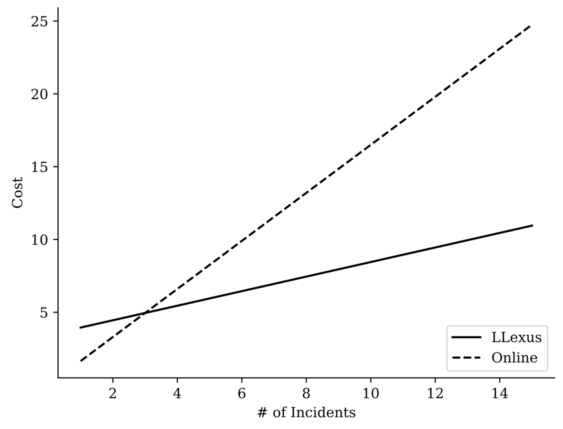


Figure 10: Comparing cost in USD for LLexus and an online-only approach for incident handling.

our case studies, the cost of generating a plan may increase depending on the number of steps required, as well as the frequency of validation failures. However, these costs are outweighed by the benefits, as the approach results in significant reduction in costs for executing the plan. Therefore, despite the initial investment in plan generation, the overall cost-effectiveness of our approach makes it a valuable tool for incident management.

6.5 Learnings

Through the exploration of the three case studies, several key learnings and insights were gained regarding the generation of executable plans using LLexus and troubleshooting guide improvement:

(1) **The Importance of Clear and Well-Written TSGs**: The case studies emphasized the critical role of having clear and well-written troubleshooting guides (TSGs). Plans generated by LLexus were directly influenced by the quality of the TSGs. In cases where the TSGs were poorly written or ambiguous, the generated plans were less effective and required more iterations to improve.

(2) **Iterative Process for TSG Improvement:** LLexus's iterative process proved to be invaluable in improving TSGs. By generating initial plans and receiving feedback, engineers could identify areas for improvement in the TSGs, such as clarifying steps, adding missing scenarios, or adjusting the order of operations. This iterative approach allowed for continuous refinement and enhancement of the troubleshooting process.

(3) **Adaptability to Diverse Scenarios:** The case studies highlighted LLexus's adaptability to diverse troubleshooting scenarios. Whether dealing with straightforward linear plans or more complex situations with conditional steps and event triggers, LLexus was able to generate executable plans that addressed the specific issues outlined in the TSGs.

Overall, these learnings emphasize the importance of clear communication in troubleshooting guides, the value of iterative improvement processes, and the adaptability and reliability of LLexus in generating effective plans for diverse troubleshooting scenarios.

7 RELATED WORK

Effective incident management is critical for maintaining reliable cloud operations. Traditionally, managing an incident involves triaging the issues [5, 6, 8], diagnosing the problems [3], and then mitigating the incident [12, 33]. Our work contributes to both the diagnosis and mitigation aspects of the incident lifecycle. Automated execution of a plan can improve the investigation and help in the resolution of the problem.

Previous research has explored mining structured knowledge from various incident-related artifacts, including incident reports [14, 25, 26], root cause documentation [24], and troubleshooting guides [12, 27]. Of particular relevance to our work is the framework proposed by Shetty et al. [27], which combines machine learning and program synthesis to automate the creation of executable workflows extracted from incident troubleshooting guides [27].

The advent of LLMs has significantly empowered new approaches to incident management [1, 2, 7, 13, 23]. Previously, reasoning about the various artifacts produced in incident management was challenging due to its human-driven nature. However, recent advances in LLMs have greatly improved this capability. For instance, Roy et al. [23]. explore LLM-based agents for incident root-cause analysis, proposing a ReAct agent that interacts with external tools to identify the root cause of an incident. Xpert [13] is another framework that automates KQL (Kusto Query Language) recommendation during incident investigation by leveraging historical incidents.

More closely related to our work, An et al. [2] utilize TSGs and historical incidents as a knowledge base to feed a Copilot that assists engineers during incident investigations. Their approach involves building an interactive tool that relies on human input during the investigation. Similarly, Transposit [30] also provides a chat-based interactive CoPilot for incident management. In contrast, our approach interacts with the engineer during the generation of an executable plan (and improvement of the TSG), but once the incident is created, the plan is automatically executed without human intervention. We believe our approach reduces human burden during incident management and reduces MTTM.

8 CONCLUSION

On the problem of mitigating live site incidents for SaaS products, a variety of approaches have been proposed by prior work. We believe we are taking a unique approach in LLexus, in leveraging an LLM-based AI agent to generate a plan from a TSG or equivalent document. Subsequently, the plan can be automatically executed against an incident, with or without additional use of an LLM-based agent depending on whether all the plans needs can be satisfied by existing tools.

In this paper, we focus on the AI agent for plan generation. It is that stage where two critical design challenges exist – mitigating LLM hallucinations and the cost of LLM execution. We have plan execution also working in LLexus, but more engineering effort is needed to generalize execution to a variety of environments (on-prem and cloud, customer deployments and engineering team deployments, access with multiple levels of authentication, etc.).

There are multiple avenues of future work that we are considering, and we encourage the research community to investigate. Recently, SLMs (Small Language Models) have been trained on curated input [10] and have been shown to run faster, incur less cost, and yet be effective at basic tasks. Perhaps an SLM could be a drop in replacement for the LLM in LLexus. The current focus of LLexus is to address machine-generated incidents, where there is a direct correlation between the incident title or details and which TSG should be followed. With customer-generated incidents, this is less clear and a learning approach may be fruitful in mining prior customer-generated incidents and resolution notes to identify which TSG is relevant. For LLexus to succeed, we still need engineers to maintain human-readable TSGs. Perhaps TSGs can be learned from prior incidents that have been resolved. Eventually, perhaps LLexus can be used earlier in the lifecycle of an incident, such as when a metric is trending towards bad but has not crossed a critical threshold, some mitigation steps could be automatically applied to avert an outage.

REFERENCES

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

[2] Kaikai An, Fangkai Yang, Liqun Li, Zhixing Ren, Hao Huang, Lu Wang, Pu Zhao, Yu Kang, Hua Ding, Qingwei Lin, Saravan Rajmohan, and Qi Zhang. Nissist: An incident mitigation copilot based on troubleshooting guides. *arXiv preprint arXiv:2402.17531*, 2024.

[3] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. Decaf: Diagnosing and triaging performance issues in large-scale cloud services. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 201–210, 2020.

[4] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[5] Junjie Chen, Xiaoting He, Qingwei Lin, Yong Xu, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. An empirical investigation of incident triage for online service systems. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 111–120, 2019.

[6] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. Continuous incident triage for large-scale online service systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 364–375, 2019.

[7] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, XueChao Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo GHOSH, Xuchao Zhang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Automatic root cause analysis via large language models for cloud incidents. In *EuroSys'24*, pages 674–688, April 2024.

[8] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee, David Maltz, et al. Scouts: Improving the diagnosis process through domain-customized incident routing. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 253–269, 2020.

[9] Object Management Group. Business process model and notation specification. https://www.omg.org/spec/BPMN, January 2014. Last accessed May 1st, 2024.

[10] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

[11] R. Howey, D. Long, and M. Fox. Val: automatic plan validation, continuous effects and mixed initiative planning using pddl. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301, 2004.

[12] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 14101420, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. Xpert: Empowering incident management with query recommendations via large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[14] Shinji Kikuchi. Prediction of workloads in incident management based on incident ticket updating history. In *Proceedings of the 8th International Conference on Utility and Cloud Computing*, UCC '15, page 333340. IEEE Press, 2015.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[16] Microsoft. Azure Resource Manager. https://docs.microsoft.com/en-us/azure/azure-resource-manager/, 2024.

[17] Microsoft. Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview, 2024.

[18] Microsoft. Kusto Query Language. https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/, 2024.

[19] Microsoft. PowerShell. https://docs.microsoft.com/en-us/powershell/, 2024.

[20] Microsoft. Semantic Kernel. https://github.com/microsoft/semantic-kernel, 2024.

[21] Microsoft. Service Fabric Troubleshooting Guides. https://github.com/Azure/Service-Fabric-Troubleshooting-Guides/tree/master, 2024.

[22] Microsoft. Service Fabric Troubleshooting Guides / 6.4 Upgrade fails for 6.3 Clusters with fabric:/System/BackupRestoreService enabled. https://github.com/Azure/Service-Fabric-Troubleshooting-Guides/blob/master/Known_Issues/Fabric%206.4%20Upgrade%20fails.md, 2024.

[23] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In *Proceedings of the Symposium on the Foundations of Software Engineering*, FSE 202, New York, NY, USA, 2024. Association for Computing Machinery.

[24] Amrita Saha and Steven CH Hoi. Mining root cause knowledge from cloud service incident investigations for aiops. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 197–206, 2022.

[25] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, and Nachiappan Nagappan. Softner: Mining knowledge graphs from cloud incidents. *Empirical Software Engineering*, 27(4):93, 2022.

[26] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, Nachiappan Nagappan, and Thomas Zimmermann. Neural knowledge extraction from cloud service incidents. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 218–227. IEEE, 2021.

[27] Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula, Arjun Radhakrishna, and Anurag Gupta. Autotsg: learning and synthesis for incident troubleshooting. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1477–1488, 2022.

[28] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

[29] Shoreline.io. Runbooks vs playbooks: Understanding their distinct roles. https://www.shoreline.io/blog/runbooks-vs-playbooks, 2024.

[30] Transposit. Transposit: Ai-powered incident management. https://www.transposit.com/, 2024. Accessed on May 5, 2024.

[31] Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. Can large language models really improve by self-critiquing their own plans? *arXiv preprint arXiv:2310.08118*, 2023.

[32] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models - a critical investigation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 75993–76005. Curran Associates, Inc., 2023.

[33] Weijing Wang, Junjie Chen, Lin Yang, Hongyu Zhang, and Zan Wang. Understanding and predicting incident mitigation time. *Information and Software Technology*, 155:107119, 2023.

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022.

[35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.