École Polytechnique Fédérale de Lausanne

Enhancing Log Analytics with Generative AI

by Edin Guso

Master Thesis

Approved by the Examining Committee:

Prof. Dr. Boi Faltings
Master Project Supervisor

Ketevani Zaridze
In-Company Supervisor

Dr. Virginia Bordignon
External Expert

EPFL Innovation Park
Building C
CH-1015 Lausanne

August 23, 2024

Simplicity is prerequisite for reliability.
— Edsger W. Dijkstra

Dedicated to my parents, without whom I would not be where I am today.

Acknowledgments

First and foremost, I would like to thank my academic supervisor, Prof. Dr. Boi Faltings, for accepting me as his student. I am also thankful to my company supervisor, Ketevani Zaridze, for her support and mentorship. Her practical insights and guidance were influental in aligning my academic work with real-world applications.

A sincere thank you to my colleagues and mentors, Marco and Virginia. Your mentorship extended far beyond this project, guiding me through both professional challenges and personal decisions. Your wisdom and support have had a great impact on my journey, and I am grateful for everything.

To my friends, Hatice, Orhan, and Su, thank you for being a second family to me. Na'Ima, thank you for the laughs we shared and the time we've spent together; your friendship means a lot to me. A special thanks to Mustafa for teaching me what selflessness really is; your kindness and generosity have been a source of inspiration.

Lastly, I want to express my deepest love and gratitude to my family. To my girlfriend Güneş, for making me a better person every day. To my majka Hidajeta, for giving me the adventures of my childhood, which continue to inspire me. And to my mom, Meliha, and dad, Jasmin, for teaching me what unconditional love is. Your support and belief in me have been the foundation of all my achievements.

Lausanne, August 23, 2024 Edin Guso

Abstract

This thesis explores the integration of Generative AI (GenAI) models, specifically Large Language Models (LLMs), into the field of log analytics to address the growing challenges posed by the increasing volume and complexity of log data in modern IT systems. Traditional log analysis methods, which often rely on manual inspection or rule-based approaches, struggle to keep up with the scale and variability of logs in large-scale environments. To address these limitations, this work introduces the ModuGPT framework, a modular system designed to simplify the development of LLM-powered applications for log analytics. The framework introduces Prompt Elements, a structured catalog of reusable components that streamline prompt engineering and improve the consistency and effectiveness of LLM interactions. Additionally, three specialized tools are developed within the framework: Insight-to-Text, which generates natural language explanations and recommendations for log insights; Text-to-ES, which provides a natural language interface for querying Elasticsearch; and SmartSearch, which enhances traditional search functionality by combining semantic and keyword-based methods. Evaluation results demonstrate significant improvements over baseline methods, with the Insight-to-Text tool outperforming the control in 95% of cases, Text-to-ES achieving perfect accuracy in translating natural language to Elasticsearch queries, and SmartSearch showing enhanced performance in both syntactic and semantic components of queries. The results show that integrating GenAI into log analytics workflows is important, offering a powerful new approach to understanding and managing large-scale log data and aligning with the evolving demands of AI-driven IT operations.

Contents

Chapter 1

Introduction

In modern IT systems, log data plays a crucial role in maintaining and troubleshooting applications, networks, and infrastructure. Logs, which are records of events that occur within these systems, are essential for diagnosing issues, understanding system behavior, and ensuring security. The growing complexity and scale of software applications have amplified the importance of effective log analytics. As systems become more complex and the volume of generated logs grows, the need for advanced tools and methods to efficiently manage log data becomes increasingly critical.

Despite their usefulness, logs can be challenging to analyze due to their large volume and the redundancy of information they contain. Modern systems can generate about 30-50 gigabytes (around 120-200 million lines) of logs per hour [1], making it difficult to process and extract valuable insights from them. Unscalable log analysis, inadequate log-processing tools, incorrect log classification, and the variety of log formats further complicate the process [2]. In traditional standalone systems, developers often manually inspect logs or create rules based on domain knowledge to detect anomalies, using keyword searches or regular expression matching. However, this manual approach is insufficient for large-scale systems, where the complexity, volume, and fault-tolerant mechanisms make it challenging to identify key information and accurately detect anomalies [3]. The following log examples, taken from NetScaler ADC (Application Delivery Controller), illustrate the complexity of log data generated in a typical high-volume system:

```
ICADHT dht-free: Core 1: freeing entry for ACB947BEB2CEFDB2313E1779A01D51
ICADHT dht-free: Core 0: freeing entry for 7E9FA07DAB54AA063F56B14A3393E7
ICADHT dht-free: Core 1: freeing entry for 0223D4DA12830B4C2EFCD4A8DDDA3D
ICADHT dht-free: Core 0: freeing entry for 9D84F28DFE3EB4141A9BF19786378A
ICADHT dht-free: Core 2: freeing entry for E2418358F10DBEB5733F1AC10D966B
```

To address the challenge of high log volumes and the need for efficient log analytics, various algorithms have been developed to extract high-level *log patterns* from log data. These algorithms aim to reduce the number of data points from millions or billions to hundreds or thousands and simplify the analysis process. Notable examples include IPLoM [4], which uses an iterative partitioning approach to hierarchically cluster log messages; SHISO [5], which builds a structured tree to detect log patterns in a streaming fashion without prior knowledge; Spell [6], which applies a longest common subsequence based method to parse system event logs; and Drain [7], which utilizes a fixed-depth parse tree with specially designed rules for online log parsing. Among these, Drain excels in terms of efficiency and accuracy. Drain+ [8], strengthens robustness by addressing challenges like separator variety and log message length variability, making it even more reliable for log analysis. Even though large language model-based approaches such as LILAC [9] offer better accuracy, Drain remains a popular choice due to its predictability and simplicity. The following is an example of a log pattern generated from the previously shown logs, demonstrating how these algorithms abstract raw log data into more manageable patterns:

```
ICADHT dht-free: Core <*> freeing entry for <*>
```

While extracting log patterns reduces the data volume and helps in the initial analysis, it is only the first step in the log analytics process. Even with the reduced volume, the number of log patterns can still be overwhelming for manual inspection. The next challenge is determining the significance of each log pattern. Some log patterns correspond to routine informational logs, while others correspond to logs critical for diagnosing issues. This is where *anomaly detection* becomes important. Anomaly detection helps identify critical log patterns that require immediate attention, concentrating the data analysis on the most relevant information. State-of-the-art log anomaly detection models often use deep learning techniques. Examples include DeepLog [10], LogAnomaly [11], and LogRobust [12]. Among deep learning approaches, some of the most effective ones utilize Bidirectional Encoder Representations from Transformers (BERT) [13]. BERT-based models, such as LogBERT [14], LAnoBERT [15], and LogFit [16], have been shown to outperform other approaches in log anomaly detection, making them a popular choice for the task.

1.1 Logmind Platform

In the SaaS market for log analytics, *Logmind* offers an AI-powered log analytics platform designed for IT operations professionals. The platform streamlines log data management and provides insights. It addresses the difficulties associated with the large volumes of log data generated by modern IT systems. Logmind employs advanced log pattern extraction and anomaly detection techniques to simplify the analysis process and highlight critical information.

Logmind uses an advanced version of the Drain algorithm, incorporating enhancements similar to those in Drain+ and additional proprietary upgrades. These refinements reinforce the robustness and accuracy of log pattern extraction, significantly improving the quality of the log patterns. Furthermore, Logmind employs a proprietary BERT-based model for anomaly detection, developed and trained in-house. This model identifies critical log patterns, referred to as *insights* in the context of Logmind, which are essential for diagnosing issues and maintaining system stability. Logmind's processing pipeline is illustrated in Figure 1.1.

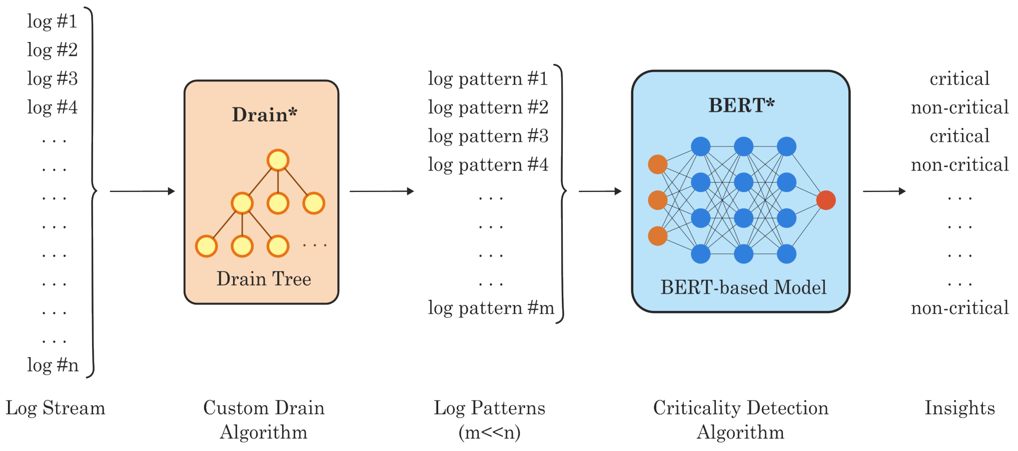

Figure 1.1: Logmind pipeline. The pipeline processes the raw log stream through advanced pattern extraction and anomaly detection to generate insights.

Despite these advancements, there is still potential for enhancement. One important challenge is that the insights presented to users are essentially groups of logs, known as log patterns, that have been identified as critical. These insights still resemble the original log data in both complexity and technicality. As a result, interpreting these log patterns can be difficult without domain-specific knowledge and expertise. This issue is particularly pronounced for non-technical users, but it also affects technical users who lack expertise in the specific domain from which the logs originate.

Understanding these insights is often not enough to identify the underlying causes of issues. Users typically need to investigate further by using the insights to query for individual logs that provide more detailed information about the problem. This deeper investigation requires a log search engine that operates on domain-specific query languages such as SQL or Elasticsearch. Such a search engine is necessary because it allows users to filter and locate specific logs within large datasets. However, proficiency in these query languages is crucial to perform effective searches, as users need to construct precise queries to retrieve relevant logs efficiently. Consequently, this requirement adds another layer of complexity, making it challenging for users who are not familiar with these specific query languages.

These challenges show the need for more advanced, user-friendly tools that can bridge the gap between insights and actionable solutions. Making log patterns easier to understand and respond to is essential for improving IT efficiency. Additionally, enabling simpler and more powerful querying can reduce the time required to diagnose and resolve problems. AI-assisted solutions that use natural language processing can address these challenges effectively, making log analytics more accessible and intuitive.

1.2 Large Language Models

Artificial Intelligence (AI) involves creating systems capable of performing tasks that typically require human intelligence by using algorithms. AI can be categorized into four approaches: thinking humanly, which involves modeling human cognitive processes; thinking rationally, which uses logical systems to mimic human reasoning; acting humanly, exemplified by replicating human behavior as in the Turing Test; and acting rationally, where agents aim to achieve optimal outcomes based on their perceptions of the environment. This classification covers the diverse spectrum of AI research and applications [17]. Within AI, *Generative AI* (GenAI) refers to systems that can create new content, such as text, images, or audio, often producing outputs that are indistinguishable from those created by humans.

Machine Learning (ML) is a branch of AI that enables systems to learn from data and improve their performance over time without being explicitly programmed [18]. A popular approach to implementing ML is through artificial neural networks (ANNs), which are computational models inspired by the human brain's interconnected neurons [19]. These networks consist of layers of interconnected nodes that work together to process input data, enabling the model to recognize patterns and make predictions. Deep learning, a subset of ML, utilizes neural networks with many layers, which is where the term 'deep' comes from, to analyze various forms of data. It excels in tasks such as computer vision, autonomous systems, natural language processing, and more, by automatically discovering intricate patterns in high-dimensional data [20]. *Natural Language Processing* (NLP) focuses on the interaction between computers and human languages, enabling machines to understand, interpret, and manipulate natural language data. NLP applications include sentiment analysis, machine translation, text summarization, and question answering [21].

Language Models (LMs) are NLP systems that can learn the structure and patterns of human language [22]. One common objective for LMs in GenAI is to predict the next token in a sequence based on the preceding tokens, where a token is typically a word or a few characters. This ability is fundamental to understanding and generating text. Early models, such as n-gram models, use statistical methods to predict the likelihood of word sequences [23]. However, these models struggle when faced with rare or unseen words and lack the ability to capture long-range dependencies in text [22]. The development of Recurrent Neural Networks (RNNs) [24] and Long Short-Term Memory (LSTM) networks [25] improved the ability to model sequences and context in text data, but they still faced challenges in capturing complex, long-range patterns and relationships.

The introduction of the Transformer architecture [26] marked a major advancement in the field of NLP. Transformers utilize self-attention mechanisms to process input data in parallel, which increases the efficiency of language modeling and allows for the capture of long-range dependencies. This architecture has revolutionized NLP, leading to the development of *Large Language Models* (LLMs) [22], which are at the forefront of GenAI. LLMs, such as GPT-3 [27], InstructGPT [28], and GPT-4 [29], are trained on large amounts of text data and are capable of understanding, generating, and manipulating human language in a way that closely mimics human communication, making them perform very well in various NLP tasks.

Reinforcement Learning from Human Feedback (RLHF) [30] has further expanded LLM capabilities by aligning AI with human preferences. This method creates a reward model based on human feedback, which is then used to fine-tunes LLMs through reinforcement learning enabling them to generate responses that better meet human expectations. This approach is particularly important for adapting foundational LLMs to specialized applications, such as conversational agents and virtual assistants.

1.2.1 Prompt Engineering

In conversational agents, users interact with the model by providing *prompts*, which are sets of natural language instructions designed to customize, amplify, or refine the model's capabilities and outputs [31]. Prompts aim to provide context for the LLM, guide its response, and establish rules or constraints the model should follow. The process of crafting these prompts to achieve specific outcomes, known as *prompt engineering* (or prompting), is crucial for maximizing the effectiveness of LLMs. Well-crafted prompts can greatly influence the quality of LLM responses, making them more accurate and contextually relevant.

Prompts for *chat models*, which are designed for interactive conversations, typically consist of three main types of messages: *system messages*, *user messages*, and *assistant messages*. System messages set the role, guidelines, or constraints for the model, influencing how it behaves. User messages contain the questions or inputs that the user wants the model to address. Assistant messages are the responses generated by the model, based on the context provided by the system and user messages. *Text completion models*, in contrast, are designed to continue or complete a given piece of text. They do not have distinct message types like chat models. However, it is possible to create a similar structure by artificially adding system and user message tags to the input text.

One of the biggest distinctions in prompt engineering is between *zero-shot* and *few-shot* prompting. In zero-shot prompting, the model generates responses without being shown any examples, relying solely on the given prompt (system message), user input (user message), and internal knowledge. This approach is useful for tasks where labeled data is unavailable. Few-shot prompting, on the other hand, involves providing a small number of demonstrations to the model to guide its responses. Demonstrations are pairs of inputs and desired outputs that help the model understand the task and generate more accurate responses [27]. These demonstrations are passed as user-assistant message pairs to chat models, and as input-output pairs to text completion models. This method can significantly improve the model's performance on specific tasks.

However, even with well-crafted prompts that utilize few-shot prompting, models often struggle with complex tasks that require multi-step reasoning. In such cases, guiding the model through the task can lead to better performance. *Prompt chaining* is a prompting technique that breaks down a complex task into a series of simpler subtasks. Each subtask is handled separately, with its own prompt and call to the LLM. The output from one subtask is then used as part of the prompt for the next subtask, creating a step-by-step process that guides the model through the overall task. A similar approach is seen in the least-to-most prompting [32] strategy, which also decomposes complex tasks into series of sequential steps. This method has proven highly effective, outperforming traditional prompting techniques on challenging benchmarks.

1.3 Semantic Search

Searching and interpreting human-generated content, such as text, is a difficult yet important task. One effective way to increase search relevance is through *semantic search*, a technique that uses the meaning of text to deliver more relevant results. Unlike traditional keyword-based search, which only relies on matching specific words, semantic search recognizes related concepts. For example, if someone searches for "network failure", a traditional keyword search would only return results with those exact words, while a semantic search could return results mentioning "connectivity issues" or "packet loss", recognizing the connection between these ideas.

To perform semantic search, *embedding models* play a crucial role in modern NLP systems. These models convert a piece of text—whether it's a word, a sentence, or even an entire document—into a vector representation in a high-dimensional space, typically ranging from 50 to over 1,000 dimensions [33]. The resulting embedding vectors, known as *embeddings*, capture the semantic meaning of the text, enabling text with similar meanings to have similar vector representations [34].

Efficiently storing and searching thorugh embeddings is a complex task. Vector databases, or *vectorstores*, solve this by providing specialized storage and querying capabilities for embeddings. They enable efficient similarity searches by comparing query embeddings against stored document embeddings to find the closest matches. This capability is useful for applications like semantic search, where quickly retrieving semantically similar documents is essential.

1.4 GenAI in Log Analytics

Existing log analytics tools can extract patterns and detect insights from log data, but users often struggle to understand the insights, especially without deep domain knowledge. GenAI models, such as LLMs, can address this challenge by enhancing insights with natural language explanations and recommended actions. These explanations and recommendations make the insights more accessible and actionable for a wider range of users.

Another challenge in log analytics is the need for effective search within large and complex log datasets. Traditional search methods, which rely on domain-specific query languages such as SQL or Elasticsearch, require users to have both technical expertise and detailed knowledge of the logs they are searching. This can make log analysis difficult and time-consuming. LLMs, with their advanced natural language processing capabilities, can simplify this process by enabling users to search logs using simple, intuitive language instead of complex queries.

Additionally, integrating LLMs with semantic search capabilities can significantly improve log search by understanding the context and intent behind user queries. This combination allows for the retrieval of logs that are not only relevant based on keywords but also aligned with the broader meaning of the user's query. By capturing both the syntax and the underlying context, this approach makes it easier and more efficient for users to find the specific information they need, enhancing the overall effectiveness of log investigations.

1.4.1 Related Work

Despite the promising capabilities of LLMs in various domains, their application in log analytics remains an underexplored area. Current GenAI solutions have not yet fully addressed the specific needs of this domain, such as generating natural language explanations for log insights, providing natural language search interfaces for log data, and developing semantic search capabilities tailored to the unique challenges of log analytics. Additionally, the integration of keyword-based and semantic search results is another topic that lacks exploration. Furthermore, the field of prompt engineering, which is crucial for maximizing the effectiveness of LLMs, has not been extensively studied. This gap exists both in the log analytics domain and more broadly in general LLM applications.

An essential requirement for log analytics platforms like Logmind, which utilizes *Elasticsearch* (ES) for storing and querying logs, is the ability to translate natural language queries into Elasticsearch queries (*text-to-ES*). However, the literature lacks comprehensive tools or frameworks specifically designed to address this need. On the other hand, substantial advancements have been made in translating natural language queries into SQL queries (*text-to-SQL*) using LLMs. Understanding the progress and challenges in text-to-SQL can provide valuable insights for developing similar tools for ES.

Recent research has explored various methods to utilize LLMs for text-to-SQL tasks, focusing on improving performance on complex queries and ensuring robustness across different database schemas [35]. For instance, methods utilizing zero-shot prompting which leverage the inherent capabilities of LLMs, have shown potential in generating SQL queries [36–40]. One of the studies attempts to use GPT-3 for zero-shot text-to-SQL, demonstrating LLMs can generate SQL queries without task-specific fine-tuning, making it a promising tool for scenarios where labeled datasets are unavailable [36]. However, despite this potential, zero-shot prompting faces significant challenges. It particularly struggles with complex queries that require deeper understanding and reasoning. The results can be inconsistent and heavily influenced by the design of prompts. Additionally, without structured prompt templates and external knowledge, zero-shot models often fail to generate accurate SQL queries in more complicated cases [35].

In addition to zero-shot prompting, few-shot prompting has emerged as a powerful technique for enhancing text-to-SQL performance. By providing a small set of example queries, few-shot prompting helps the model generalize across different database schemas and handle query complexities more effectively [39–42]. One innovative approach involves strategically selecting demonstrations from both out-of-domain examples and synthetically generated in-domain examples to create a hybrid set of prompts. This method leverages the strengths of diverse data sources, enhancing the model's ability to perform in cross-domain text-to-SQL tasks [41]. While this approach has shown increased execution accuracy, few-shot models still face challenges with complex queries, especially when the examples provided do not fully capture the necessary context or schema. The effectiveness of these models is highly dependent on the quality of the selected examples, and they can struggle with queries where the examples do not align well with the user's question or the database schema [35].

Other advanced methods have further pushed the boundaries of text-to-SQL performance. SQL-PaLM [43] combines few-shot prompting with instruction fine-tuning, introducing techniques such as consistency decoding with execution-based error filtering, and expanded training data coverage during instruction fine-tuning. DIN-SQL [44], on the other hand, decomposes the text-to-SQL task into smaller sub-tasks, utilizing in-context learning and self-correction to iteratively refine solutions. Both approaches have shown notable success on benchmarks like Spider [45] and BIRD [39], especially in handling complex SQL queries.

However, the application of both SQL-PaLM and DIN-SQL to log databases presents unique challenges due to the inherent complexity of log data and the need for sophisticated queries. Transitioning these advanced text-to-SQL techniques to text-to-ES tasks introduces additional difficulties. SQL is far more commonly used, meaning LLMs are exposed to it more frequently during training, which strengthens their ability to generate SQL queries. In contrast, models are less familiar with ES queries, decreasing their effectiveness. Furthermore, the JSON format of ES queries requires managing multiple levels of abstraction due to its hierarchical structure and nested elements, and often necessitates more tokens to express the same query compared to SQL. These factors make handling ES queries more challenging for LLMs than the flatter, more linear structure of SQL queries. These combined challenges highlight the need for specialized techniques and models tailored to the text-to-ES task.

Moreover, a notable gap exists in tools that generate natural language explanations and recommendations for log data. Such tools are essential for effectively interpreting and acting on log insights. The development of these tools could improve the usability of log analytics platforms by simplifying the interpretation of log data and facilitating quicker and more accurate responses to system issues.

Another area that holds great potential for log analytics is semantic search, which is extensively studied in various other domains [46–49], has seen limited adoption in this field. By focusing on the meaning and context of queries rather than strict keyword matching, semantic search can improve the relevance and accuracy of log searches. Integrating it with traditional keyword-based methods could provide a more comprehensive search experience.

On top of the existing gaps in the utilization of LLMs for log analytics, there is a broader lack of research on prompt engineering. Recent efforts have made significant progress, such as the development of a structured catalog of prompt patterns designed to enhance prompt engineering for ChatGPT [31], and the introduction of guiding principles for crafting effective prompts across multiple LLMs [50]. These contributions provide valuable guidelines that improve the accuracy of LLM outputs by offering structured approaches to prompt design. However, while they effectively guide prompt creation, they lack a modular framework for creating prompts. The current literature does not address the need for a practical framework that enables the systematic combination and adaptation of prompts, leaving a critical gap in the field. Addressing this gap is crucial for enhancing the adaptability and performance of LLMs in diverse and complex tasks, including those in log analytics.

These limitations highlight the urgent need for more advanced AI-driven solutions that can provide deeper insights and enable more intuitive interactions with log data. Additionally, the development of a comprehensive and modular framework for prompt engineering is critical to fully unlocking the potential of LLMs across diverse applications. By addressing these challenges, there is considerable potential to advance the field of log analytics, making it more accessible, effective, and responsive to the demands of real-world scenarios.

This thesis aims to contribute to the field of log analytics by integrating Generative AI models with traditional log analysis techniques, introducing innovative tools that enhance the accessibility, interpretability, and actionable value of log data.

1.5 Core Contributions

This thesis introduces several key contributions designed to enhance the effectiveness of log analytics by leveraging Generative AI:

- *ModuGPT* is a modular framework designed to simplify the development of applications that interact with LLMs. It provides a flexible and extensible architecture, allowing developers to build LLM-powered applications efficiently. The repository can be accessed at this link. It is important to note that the proprietary prompts and tools developed for Logmind are redacted from the public repository. However, the tools demonstrated in this thesis, as well as the core framework, are available for public use.
- *Prompt elements* form a structured catalogue within the ModuGPT framework, consisting of reusable and composable prompt components designed to make LLM interactions more flexible and effective.
- *Insight-to-Text* enriches insights with natural language explanations and actionable recommendations. By simplifying the interpretation of insights, Insight-to-Text ensures a clearer and quicker understanding of the log data.
- *Text-to-ES* provides a natural language interface for querying log data, allowing queries to be performed without the need for proficiency in domain-specific query languages. This tool facilitates effective log analysis through straightforward, intuitive querying.
- *SmartSearch* builds on traditional search functionality by integrating semantic search with keyword-based methods. This combination enables more accurate and contextually relevant search results, making log data retrieval more efficient.

Each of these tools has undergone extensive evaluation to validate their effectiveness. The evaluation process included rigorous testing against diverse datasets, simulating the complex and large-scale environments typical of modern IT operations. The proposed tools consistently demonstrated strong performance.

These contributions address the increasing complexity and scale of log data in modern IT environments. By integrating LLMs, this work improves the contextualization and usability of log analysis tools, aligning with the evolving demands of AI-driven IT operations.

1.6 Thesis Outline

The remainder of this thesis is structured as follows:

- **Chapter 2** outlines the key design decisions for the ModuGPT framework, prompt elements, and the three tools: Insight-to-Text, Text-to-ES, and SmartSearch. It explains the rationale behind these choices and how they enhance log analytics with GenAI.
- **Chapter 3** details the implementation of ModuGPT and the associated tools. It discusses the technical challenges, technical decisions, and the steps taken to integrate these GenAI tools into log analytics workflows.
- **Chapter 4** presents the evaluation of the proposed tools, including an analysis of their performance. It also covers the LLM evaluation methods used, highlighting the improvements in log analytics achieved through these innovations.
- **Chapter 5** summarizes the key findings, discusses the impact of the research, and suggests directions for future work, such as further fine-tuning LLMs and exploring new features like topic generation for log patterns.

Chapter 2

Design

In this chapter, the design choices made during the development of the ModuGPT framework and its associated tools are discussed. It begins with a brief overview of the ModuGPT terminology, followed by a detailed description of the *ModuGPT pipeline*. Next, prompt elements are defined and discussed. The chapter then examines the design of the Insight-to-Text, Text-to-ES, and SmartSearch tools, focusing on their key features and design decisions. The implementation details of these tools are discussed in Chapter 3.

2.1 Terminology

Understanding the specific terms used in the ModuGPT framework is important for grasping its architecture and functionality. These terms define the key parts and ideas that make up the framework, helping to clarify the design and operation of ModuGPT throughout this document.

- *DriverAgent* is the core component of ModuGPT, and is responsible for managing the ModuGPT pipeline. DriverAgent handles all aspects of a single LLM call, from receiving user input to producing the final output. A DriverAgent consists of multiple specialized Modules.
- *Module* is a specialized component within DriverAgent. Each module performs a specific role in the pipeline, such as input validation, prompt generation, or output formatting.
- *CustomAgent* is an agent capable of performing a variety of operations beyond a single LLM call. A CustomAgent can chain multiple DriverAgents sequentially, execute them in parallel, or perform tasks not requiring LLMs, such as conducting semantic searches.

- *Tool* is a blueprint for constructing agents. Tools define the modules and overall structure of an agent. They serve as templates that dictate how agents are built and operate within the ModuGPT framework.
- *Prompt elements* are reusable and composable components used in the prompt generation process. These elements are combined to create prompts that effectively guide the LLM in generating the desired output.

2.2 ModuGPT Pipeline

Deploying LLMs in a production environment demands more than just prompt design and calling the model; it requires a robust and well-structured workflow to handle data validation and manipulation, automated prompt generation, cache operations, and error management. The DriverAgent coordinates these tasks in what is known as the ModuGPT Pipeline (Figure 2.1), ensuring that interactions with LLMs are reliable, efficient, and scalable.

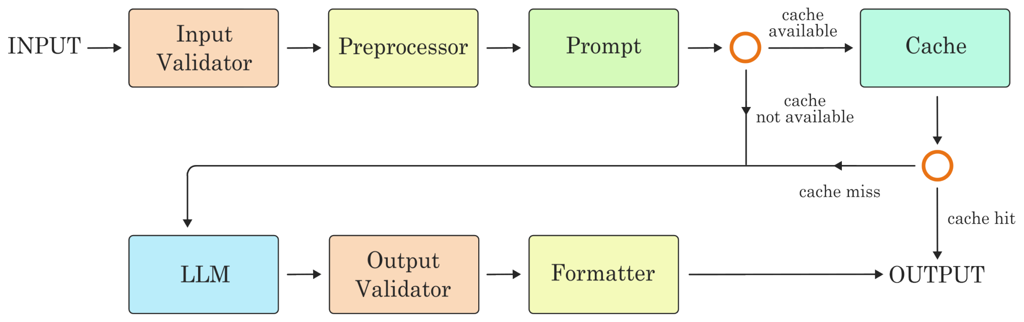

Figure 2.1: The ModuGPT pipeline. The pipeline consists of several modules that handle input validation, preprocessing, prompt generation, model interaction, output validation, and formatting. This modular design ensures that interactions with LLMs are reliable, efficient, and scalable.

The pipeline begins with the *Input Validator* module, which guarantees that input data is correctly formatted and adheres to the constraints of the model. This module checks for missing fields, incorrect data types, and other tool-specific requirements. If the input is invalid, the pipeline returns a specific error message to the user, preventing the model from processing flawed data, which could lead to unreliable outcomes.

Following the validation step, the *Preprocessor* module adjusts and enriches the input data to meet the tool's requirements. This may involve adding domain-specific knowledge, converting data into an API-friendly format, or performing other necessary transformations. This module makes sure that the data passed to subsequent modules is properly formatted and contains all required information.

Next, the Prompt Generator (or simply *Prompt*) module is responsible for creating the prompt that will be sent to the model. Acting as a template engine, it replaces placeholders in the prompt template with relevant input data. In addition, it may also perform operations such as querying databases to fetch examples for few-shot prompting or retrieving tool-specific information. Prompts consist of various prompt elements, integrated to form system, user, and assistant messages that guide the model in generating the desired output.

An optional component of the pipeline, the *Cache* module, stores the results of previous interactions with the model. This module can reuse previously generated outputs, thereby speeding up response times and reducing operational costs by minimizing redundant model calls. It is particularly beneficial in scenarios where same prompts are repeatedly submitted to the model.

At the core of the pipeline is the *LLM* module, which handles the interaction with the model to generate the output. This module sends the prompt to the model, receives the generated text, and passes it to the next stage for further processing. It also handles errors that may occur during the interaction with the model, keeping the pipeline robust and reliable. It abstracts the complexities of interacting with various model types—whether local, remote API, or cloud-based—allowing the rest of the pipeline to remain focused on the application logic.

Once the model generates the output, the *Output Validator* module guarantees that the output meets the tool's requirements. This module checks that the output is in the expected format and verifies that the model did not refuse the prompt, a concept that will be discussed later alongside the prompt elements. If the output is invalid, the pipeline returns an error message to the user, so that only reliable and consistent results are provided.

Finally, the *Formatter* module processes the raw output, converting it into a structured format. This format is designed to be easily understandable and actionable for the user, as well as compatible with other tools if needed. This module ensures that the final output is presented in a user-friendly manner.

In summary, the ModuGPT Pipeline integrates several key modules that collaborate to provide reliable, efficient, and scalable interactions with LLMs. By modularizing each stage—from input validation to output formatting—this pipeline simplifies the process of building LLM-powered applications.

2.3 Prompt Elements

Designing prompts is not trivial; it requires an understanding of how LLMs interpret and process natural language inputs. Learning prompt engineering is similar to learning a new language—a complex language where nuances of phrasing, structure, and context significantly impact the model's output. It also requires knowing how to structure the prompt effectively, such as determining which parts to place where and how to utilize system, user, and assistant messages. Even the specific placement of a relevant piece of information within the prompt can affect the model's ability to utilize that information. For instance, it was observed that LLMs are considerably more effective at retrieving information from the beginning and the end of a prompt compared to the middle [51]. The non-intuitive nature of prompt construction often demands extensive trial and error, which can be time-consuming and prone to errors. Due to the lack of standardization, this trial-and-error process typically starts from scratch with each task, making it inefficient and increasing the likelihood of mistakes, especially when dealing with complex tasks.

To address these challenges, prompt elements were developed specifically for the ModuGPT framework. These modular building blocks streamline prompt creation, each serving a unique purpose and encapsulating a distinct part of the prompt. These elements can be combined in various ways to form complete prompts that effectively guide LLMs. By modularizing prompt construction, prompt elements make the process more intuitive, enabling users to focus on the task rather than the intricacies of prompt design. The modular nature of prompt elements also facilitates reuse and customization, with each element being self-contained and designed with a clear purpose, making it adaptable for different tasks or domains. This approach represents a novel method for enhancing the flexibility and efficiency of LLM interactions.

2.3.1 Types of Prompt Elements

The prompt elements available in the ModuGPT framework are as follows:

- *Persona.* This element generates a system message which defines the role or character (`persona`) the model should assume, tailoring the model's language and tone. A relevant persona can considerably increase the quality and relevance of responses. Optionally, it can include a specialization (`specialization`) and even the description of the field (`field_desc`), in case the field is not well-known.

```
System Message

You are {persona}. You are specialized in {specialization}.
{field_desc}.
```

- *Input Description.* This element generates a system message specifying the content of the data (`input_desc`) that the model will receive. Optionally, it can include further clarifications (`clarification`) about the input data regarding its format or structure. This element helps the model interpret the input correctly.

```
System Message

You will be given {input_desc}. {clarification}.
```

- *Task Description.* This element generates a system message that defines the specific task (`task_desc`) the LLM is expected to perform. It sets the context and guides the model's response by providing a clear and concise description of the task. This element makes sure that the model understands the user's intent and generates relevant output.

```
System Message

Your task is to {task_desc}.
```

- *User Profile.* This element generates a system message incorporating information about the user (`user_profile`). This information helps the model tailor its responses to the individual's needs, enhancing the user experience.

```
System Message

You are talking to {user_profile}.
```

- *Template.* This element generates a system message that describes an output format (`template`) that the model must follow. This element is useful for ensuring the model's output is structured in a particular way so that specific information can be extracted or processed easily.

```
System Message

Your output must imperatively be formatted as follows: {template}.
```

- *Refusal.* This element generates a system message instructing the model on how to respond (`error_keyword`) when information is insufficient or when the given task (`task_summary`) cannot be completed. It defines the model's behavior in such scenarios, making it possible to detect when the model is unable to complete the task. The `error_keyword` defined in the refusal element is later used in the output validation step to detect when the model has refused the prompt and generate an appropriate error message.

```
System Message

If you cannot {task_summary}, then you must output the following:
{error_keyword}.
```

- *Rules.* This element generates a system message which sets guidelines or constraints (`rules`) that the model must follow, resulting in consistent and accurate outputs. Triple hash signs are used to delimit the rules, making it clear where the rules begin and end. These rules usually include specific requirements that are not explained in other elements.

```
System Message

You must follow the rules which are delimited by triple hash signs. ###
{rules[0]}
{rules[1]}
...
###
```

- *Schema.* This element generates a system message presenting the schema of the database (`schema`) on which the query will run. Triple backticks are used to delimit the schema, helping the model understand which part of the prompt contains the schema information. This element also handles retrieving the relevant schema information from the database, abstracting the complexity of database interactions from the developer. This element is useful when the model needs to generate responses that interact with a database, as it provides the necessary context needed to generate accurate responses.

~~~
System Message

The query will run on a database whose schema is delimited by triple
backticks. ```
{schema}
```
~~~

- *Examples.* This element provides sample inputs and expected outputs as user-assistant message pairs (`examples`) to guide the model's response. This element is used to perform few-shot prompting. It handles retrieving the examples from the database and selecting the most relevant ones for the task, abstracting the complexity of example retrieval and selection from the developer. This element is useful when the model struggles to generate satisfactory responses using zero-shot prompting.

```
User Message
{examples[0].input}
```

```
Assistant Message
{examples[0].output}
```

```
User Message
{examples[1].input}
```

```
Assistant Message
{examples[1].output}
```

```
User Message
...
```

- *Default.* This element generates a system message with no specific instructions or formatting. It is fully customizable (`prompt`) and is used when none of the other prompt elements are suitable for the task. This element provides a flexible option for creating prompts that do not fit the predefined elements.

```
System Message
{prompt}
```

2.3.2 Combining Prompt Elements

Effectively combining prompt elements is essential for maximizing the performance of the LLM. Empirical evidence indicates that using multiple system messages within a prompt degrades LLM performance, leading to less accurate outputs. This is an expected behavior, as LLMs are trained with a single system message and multiple user-assistant message pairs during the RLHF fine-tuning process. To avoid degrading the model's effectiveness, prompt elements must be 'compressed' into a single system message.

An efficient order for combining these elements is as follows: Persona, Input Description, Task Description, User Profile, Template, Refusal, Rules, Schema, and Examples. This sequence has been empirically shown to yield strong results. The Persona element comes first, setting the role for the model and ensuring consistent tone and style. Input Description and Task Description follow, providing context about the data and the specific task. Next, the User Profile element personalizes the response, followed by Template, Refusal, and Rules, which define output structure and behavioral constraints. The Schema element, crucial for tasks involving data queries, is included to give the model the necessary structural context. Finally, the Examples element is positioned last, naturally following the system message with user-assistant message pairs that guide the model. It is important to note that while this specific ordering of prompt elements has been shown to perform well, it is not a strict rule, and the order can be adjusted based on the task requirements.

An example prompt combining multiple prompt elements (Persona, Input Description, Task Description, Refusal, and Examples) is shown below. The example replaced the pertinent variables with the user-chosen values, and it uses the one-shot prompting strategy, which is essentially few-shot prompting where the number of examples is one.

```
System Message

You are a network engineer. You are specialized in log analytics. You
will be given a log message. Your task is to extract the host name from
the log message. If you cannot extract the host name, then you must
output the following: <FAILED>.
```

```
User Message

2024-02-26 08:00:00, host=server1, message=connection established
```

```
Assistant Message

server1
```

```
User Message

2024-08-23 12:00:00, host=server2, message=connection terminated
```

The framework is designed to be flexible, allowing for the integration of new prompt elements as needed. This adaptability makes certain that ModuGPT remains robust and scalable for a variety of applications, including future features that will be discussed in Chapter 5. By compressing and strategically ordering these elements, the ModuGPT framework optimizes LLM interactions.

2.4 Insight-to-Text

The insight-to-text tool is a DriverAgent designed to generate natural language explanations and actionable recommendations for insights. A central design choice for insight-to-text was the decision to use zero-shot prompting for the task. LLMs, particularly those trained on diverse datasets, possess significant inherent knowledge about various log formats and system behaviors. This enables them to generate insightful and accurate explanations without needing extensive examples or fine-tuning. The prompt used for insight-to-text includes several key elements: Persona, Input Description, Task Description, User Profile, Template, and Rules. This combination aids the model in understanding the context, the nature of the input, and the specific task it needs to perform. The complete prompt is available in Section A.1 for reference. The insight-to-text tool is designed to hide underlying complexities, offering developers a user-friendly interface that abstracts away the internal processes, as shown in Figure 2.2.

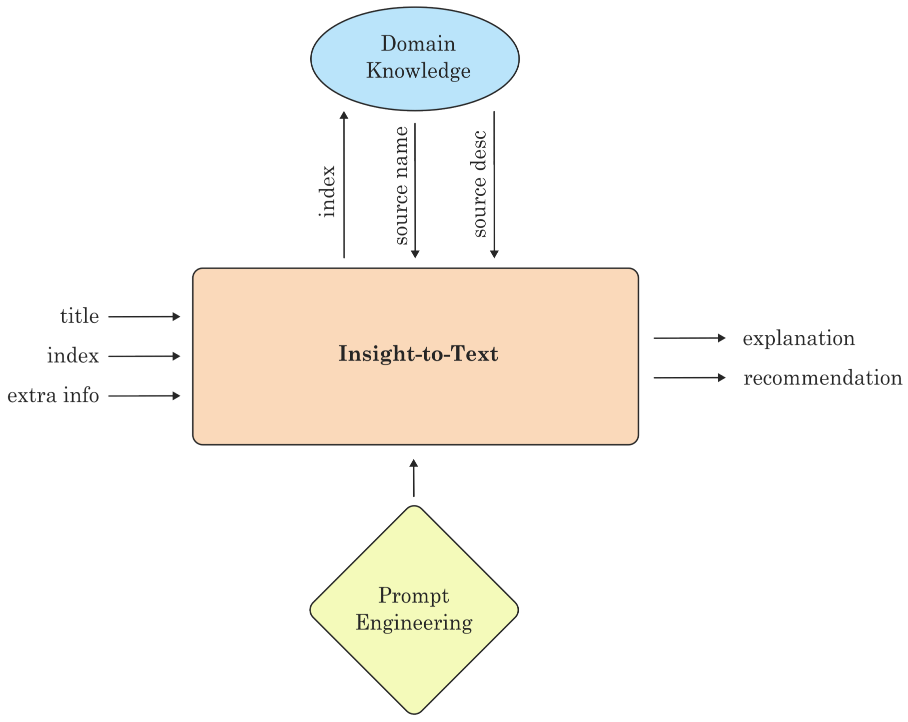

Figure 2.2: Insight-to-Text tool. The tool receives as input the insight title (which is the log pattern text), index name, additional information, and uses prompt engineering together with domain knowledge to generate an explanation and a recommendation for the given insight.

2.4.1 Optimizing Input Processing and Output Generation

The design of the input interface and preprocessing system for the insight-to-text tool prioritizes an intuitive and straightforward experience for developers. They can easily input insight details into the tool, including the log pattern text (referred to as the `title`), the source of the logs (`index`), and any additional relevant information (`extra_info`). This approach allows developers to submit insights into the tool with minimal effort. The input validation process guarantees that the input is correctly formatted and contains all the necessary information.

Once the input is submitted and validated, the Preprocessor module prepares the data in a format optimal for the LLM to process. The module begins by combining the `title` and any `extra_info` into a cohesive user message. The `index` is then used to retrieve the name and description of the log source from the domain-specific knowledge base, enriching the input with relevant context. This translation is particularly important because index names in Logmind often do not stick to common naming conventions and may include customer-specific information, making it difficult for the model to interpret them correctly without additional context. This preprocessing step ensures that the model has all the necessary information to generate accurate and relevant explanations. The resulting input is then passed to the Prompt module, which constructs the prompt for the model.

A significant challenge during the development of Insight-to-Text was the model's tendency to generate explanations that often began with phrases such as `This log indicates that...` or similar constructions. Although these phrases were technically correct, they made the output feel less natural and more robotic. User feedback indicated a strong preference for direct and concise explanations. To address this, the Formatter module was specifically designed to remove such introductory phrases, resulting in a more natural and fluid output. Despite this removal, the remaining part of the sentence generated by the model remained valid and coherent, maintaining the integrity of the explanation while improving readability.

To maintain consistency in applying this approach, the Template prompt element was utilized to guide the model in generating outputs that began with `This log indicates that...`. This deliberate prompting allowed for reliable formatting and served as a part of the output validation process, where the presence of this specific phrase is checked every time to maintain consistency. In addition to this refinement, the design of the output format itself was carefully considered. Instead of opting for a JSON format, which some models struggle with and which can increase token usage—leading to higher costs and longer response times—a simpler and more flexible format was chosen. The selected format uses delimiters (`<|EXP|>` and `<|REC|>`) that facilitate easy extraction of the explanation and recommendation. This approach also gives the model more freedom in generating outputs, which in turn boosts performance. To keep the model's responses neatly contained, a stopword (`<|END|>`) is employed to signal the end of the output, so that the LLM's responses terminate at a sensible point.

2.4.2 Enhancing Formatting Consistency

Despite extensive prompt engineering, the model occasionally replicated variable tokens (`<*>`) from the insight titles when generating explanations. Ideally, these tokens should be referenced abstractly, as the user interface represents them with visual elements that display the distribution of values for those tokens. Since no prompt strategy could entirely prevent the model from using these tokens, the logit bias parameter was adjusted to reduce the likelihood of such tokens being generated. Logit bias allows for the artificial adjustment of specific tokens' probabilities, effectively preventing the appearance of unwanted tokens and helping the output align more closely with the desired format.

Example: Following is an example execution of the insight-to-text tool, where the input is a log pattern text, the index name, and additional information. The output includes an explanation and a recommendation for the given insight.

Input Example

```
title: "SNMP Contact Lost: No SNMP reply from device <*> caused by SNMP
Error: Timeout [4098], last uptime was <*> <*>"
index: "*****-extremenetworks"
extra info: {}
```

Output Example

```
explanation: "The Simple Network Management Protocol (SNMP) has lost
contact with a specific device, denoted by the first variable token.
The cause of this issue is an SNMP Error: Timeout [4098]. This means
that the device did not respond to the SNMP request within the expected
time frame. The last uptime of the device is represented by the
remaining variable tokens."
recommendation: "Firstly, check the network connectivity between the
SNMP manager and the device. If the network is fine, verify the SNMP
configurations on the device, ensuring that the SNMP community strings
match between the device and the SNMP manager. Also, check if the
device is overloaded, as this could cause SNMP timeouts. If the issue
persists, consider increasing the SNMP timeout value on the SNMP
manager."
```

2.5 Text-to-ES

The text-to-ES tool is designed to provide a natural language interface for querying ES. Due to the inherent complexity of log databases, the nested nature of queries, and the general difficulty of the ES query language, generating correct queries is a challenging task. To improve performance, text-to-ES employs few-shot prompting. The prompt for text-to-ES incorporates the following elements: Persona, Input Description, Task Description, Refusal, Rules, Schema, and Examples. This combination is particularly effective in helping the model understand the context, the structure of the data, and can learn from examples. The complete prompt is detailed in Section A.2. One crucial aspect of the text-to-ES tool is its design, which abstracts the underlying complexity, providing developers with a straightforward interface that integrates seamlessly into their applications, as illustrated in Figure 2.3.

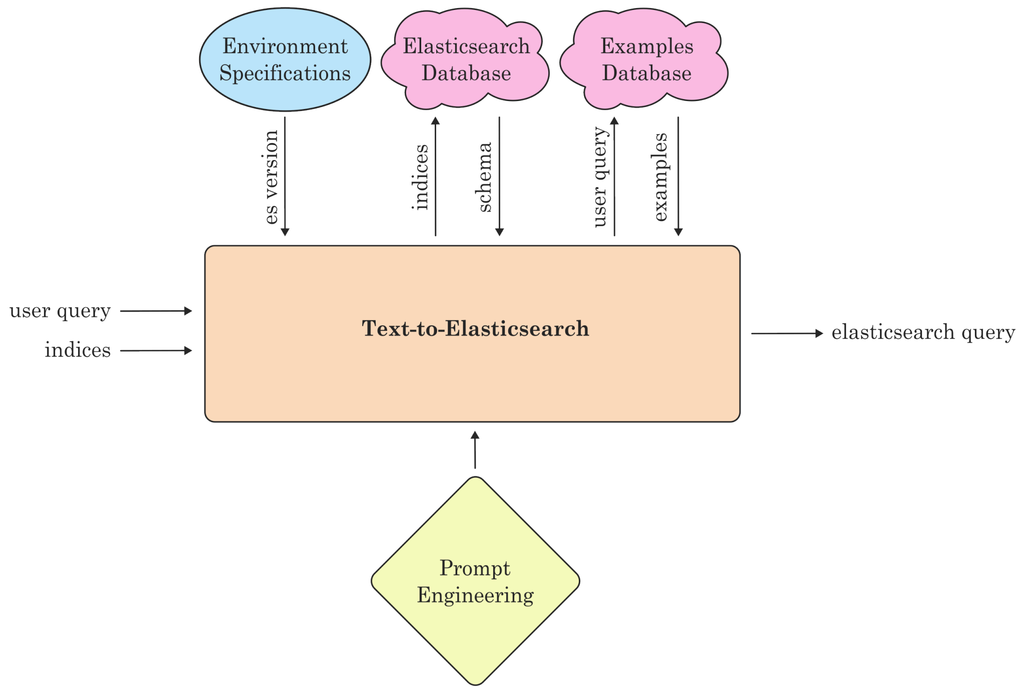

Figure 2.3: Text-to-ES tool. The tool accepts a natural language query and the indices to search, and utilizes prompt engineering and system specifications to generate an Elasticsearch query.

2.5.1 Improving Query Accuracy and Efficiency

A significant challenge in translating natural language queries into ES queries is making sure that the model accurately understands the structure of the database. The Schema prompt element is critical in this process, as it provides the model with essential mapping information, detailing the available fields and their data types. This knowledge is essential for generating accurate queries. Additionally, the ES version number is provided to the model to help it generate queries that are compatible with the version of the ES running on the server, further enhancing the accuracy of the output.

In addition to schema information, specific rules are enforced to help maintain the quality and efficiency of the generated queries:

- `Wrap all the queries in filter context`: This involves placing all the elements of the query in a filter context, which tells Elasticsearch to execute the query without scoring the results. This optimization reduces the processing load on the system and speeds up query execution.
- `Retrieve only the necessary fields`: This rule limits the data retrieved by the query to only what is essential, reducing the amount of data transferred. This is particularly useful when dealing with large documents with many fields.
- `Output only the elasticsearch dictionary`: This prevents the model from generating unnecessary output such as explanations of the query, reducing the token usage and simplifying the parsing of the response.
- `Compress the dictionary by removing whitespaces`: By eliminating unnecessary spaces in the output, this rule minimizes token usage, cutting costs, and speeding up response times.

These rules collectively help optimize the queries for performance and cost-effectiveness, while still delivering the required information efficiently. To increase the accuracy of the generated queries, the Examples prompt element is used to perform few-shot prompting. This element utilizes a semantic similarity example selector to choose the most relevant examples based on the user's query. By guiding the model with examples that closely match the user query, the Examples element significantly boosts the accuracy of the output. This approach is particularly beneficial because it ensures the model is influenced by relevant examples, whereas randomly selected examples could potentially decrease performance if they are not aligned with the user's query.

Together, these elements—Schema, Rules, and Examples—work together to help the text-to-ES tool generate accurate and efficient queries.

2.5.2 Enhancing Query Reliability

Given that the text-to-ES tool's interface is directly exposed to end-users, it is crucial to handle invalid or unprocessable queries effectively. The Refusal prompt element plays a key role in helping the model recognize when it should reject certain inputs. This is particularly important because users may submit queries that are either invalid or beyond the model's capability to process accurately. In such cases, the model is designed to reliably generate an appropriate error message rather than attempting to produce an incorrect query. This error message is later parsed in the output validation step, where it raises an error that can then be caught by developers and displayed to users in a clear and user-friendly manner.

To strengthen the tool's robustness, the text-to-ES tool employs the *leading word* technique, particularly useful for text completion models. This involves appending a specific starting text, such as `{"`, to the end of the original prompt to guide the model in generating a properly formatted and compressed JSON object without new lines or spaces. When the model reads the prompt, it recognizes that the output should continue in this format. This technique can be adapted for various tasks, such as using `SELECT` to start an SQL query or `import` for a Python script. By setting the appropriate leading word, the model is more likely to produce consistently structured and accurate responses, thereby improving the reliability of the tool.

Example: The following example demonstrates the text-to-ES tool in action, where the user query is translated into an Elasticsearch query. The input includes the user query and the indices to search, and the output is the generated Elasticsearch query.

```
Input Example

user query: "Find the top 3 hosts that generated the most ERROR level
logs for each month in 2023."
indices: ["*****-iis"]
```

```
Output Example

es query: {"query":{"bool":{"filter":[{"term":{"level.keyword":"ERROR"
}},{"range":{"@timestamp":{"gte":"2023-01-01","lt":"2024-01-01"}}}]}},
"aggs":{"monthly_top_hosts":{"date_histogram":{"field":"@timestamp",
"calendar_interval":"month"},"aggs":{"top_hosts":{"terms":{
"field":"host.keyword","size":3}}}}},"size":0}
```

2.6 SmartSearch

The SmartSearch tool is designed to provide a powerful and flexible query engine by combining traditional keyword-based *syntactic* querying with embedding-based *semantic* querying. This dual approach enables more nuanced and effective searches, allowing users to retrieve logs that match not only specific keywords but also broader meanings within their queries. Unlike tools like text-to-ES, which focus solely on translating natural language queries into ES queries, SmartSearch incorporates semantic understanding to handle complex queries that require both precise keyword matches and concept-based searches.

For example, consider the following query: "Display logs related to network connectivity issues from host server1". Note that it contains both syntactic and semantic elements. Specifically, the string "from host server1" corresponds to the syntactic part of the query, requiring exact matches, while the string "network connectivity issues" corresponds to the semantic part, referring to a broader concept. To effectively manage the complexity of handling both syntactic and semantic queries, SmartSearch utilizes a multi-step reasoning process implemented through *prompt chaining*. The first step in this chain is handled by the Query Auditor tool, as described in the following section.

2.6.1 Query Auditor: Parsing and Splitting Queries

The Query Auditor is designed to parse the natural language query and separate it into its syntactic and semantic parts. The Query Auditor operates using the following prompt elements: Persona, Input Description, Task Description, Template, Refusal, Rules, Schema, and Examples. The full prompt is available in Section A.3. This structure is carefully designed to guarantee that the model correctly parse the user's query and can effectively split it into its relevant parts.

The Schema element equips the model with useful information about the database structure, including key names and data types. This helps accurately map the user's query to the appropriate fields within the database, making syntactic query generation more precise. The Examples element further refines this process by utilizing a semantic similarity selector to choose relevant examples that closely match the user's query, significantly improving the model's performance. The Rules element plays a vital role in guiding the model to remove irrelevant parts of the query, distinguish between keyword-based searches and semantic searches, and produce an output that is concise, free of redundancies, and correctly aligned with the database schema. Additionally, it provides specific instructions for handling quoted phrases, ensuring they are included in the syntactic query as exact matches. These rules are essential for maintaining the accuracy and efficiency of the query auditing process.

To maintain reliability and efficiency, the Query Auditor avoids using a JSON format for its output, since, as already mentioned, JSON can increase token usage and lead to higher costs and processing times. Instead, the Template prompt element defines a simpler format with delimiters (`<|SYN|>` and `<|SEM|>`) that facilitate easy parsing and integration of syntactic and semantic parts into subsequent tools, and competion delimiter (`<|END|>`) to signal the end of the output. Additionally, the Refusal prompt element enhances reliability by instructing the model to generate an error message when a query cannot be processed accurately, allowing developers to provide clear and helpful feedback to users.

The Query Auditor simplifies query parsing by offering a user-friendly interface that abstracts the process's complexities. This design, consistent with other tools in the SmartSearch suite, allows developers to interact with it easily without needing to understand the underlying mechanics. The structure of the Query Auditor is shown in Figure 2.4.

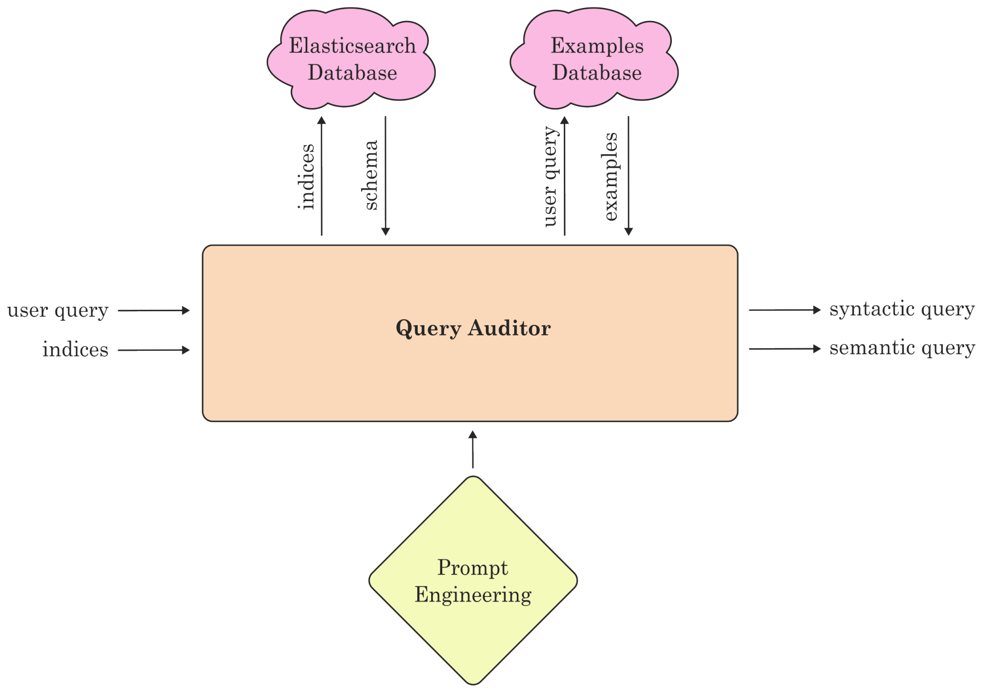

Figure 2.4: Query Auditor tool. The tool is responsible for taking a natural language query and indices as input, parsing the query into syntactic and semantic parts, and generating a structured output that can be used by subsequent tools.

Example: This execution example demonstrates the Query Auditor tool, where the user query is split into its syntactic and semantic parts. The input includes the user query and the indices to search, and the output is the syntactic and semantic parts of the query.

```
Input Example

user query: "Hello! We have been encountering some network problems.
Were any ERROR level logs that may be related to this generated today?"
indices: ["*****-iis"]
```

```
Output Example

syntactic query: "Show logs that are ERROR level generated today"
semantic query: "network problems"
```

2.6.2 Processing Syntactic Queries with Text-to-ES

Once the Query Auditor has separated the syntactic part of the query, this output is passed to the text-to-ES tool for further processing. Text-to-ES, as previously discussed, is designed to translate natural language into ES queries. By chaining the output of the Query Auditor into text-to-ES, SmartSearch helps accurately translate the syntactic elements of the query into a format that Elasticsearch can process. This two-step approach—extracting the syntactic part of the query and then processing it with the text-to-ES tool—enables SmartSearch to handle complex queries effectively.

2.6.3 Handling Semantic Queries with Semantic Search

For the semantic component of the query, SmartSearch employs a Semantic Search tool. This tool processes the semantic part of the query by converting it into an embedding, a vector representation of the query's meaning. These embeddings are then compared against precomputed embeddings of log patterns, which have been generated and stored in a specialized database known as a vectorstore. This vectorstore is designed to efficiently handle and retrieve data based on the semantic similarity of these embeddings. In this process, the Semantic Search tool identifies the most relevant log patterns by matching the vector representation of the user's query with the stored embeddings in the vectorstore. By focusing on the meaning rather than just keywords, this approach allows SmartSearch to retrieve logs that align with the broader context of the query, resulting in more nuanced and accurate search outcomes.

2.6.4 Merging Results

The final step in the SmartSearch process involves merging the results from the syntactic and semantic searches. The output from text-to-ES, which includes the Elasticsearch query, is combined with the most relevant log patterns identified by the Semantic Search tool. This is achieved by intersecting the results, where the ES query is complemented with an additional filter that targets only the logs corresponding to the semantically similar patterns. This intersection allows the final set of results to match the specific keywords provided by the user while also aligning with the broader context of their query, resulting in highly relevant and precise search outcomes.

Additionally, the SmartSearch tool utilizes the intermediate results from the Query Auditor in order to provide a natural language *feedback* to the user. This feedback allows user to understand how the query was processed and what parts of the query were used in the search. This feature enhances the user experience by providing transparency and insight into the search process. The feedback additionally serves as a validation mechanism, allowing users to verify that their query was correctly interpreted and processed.

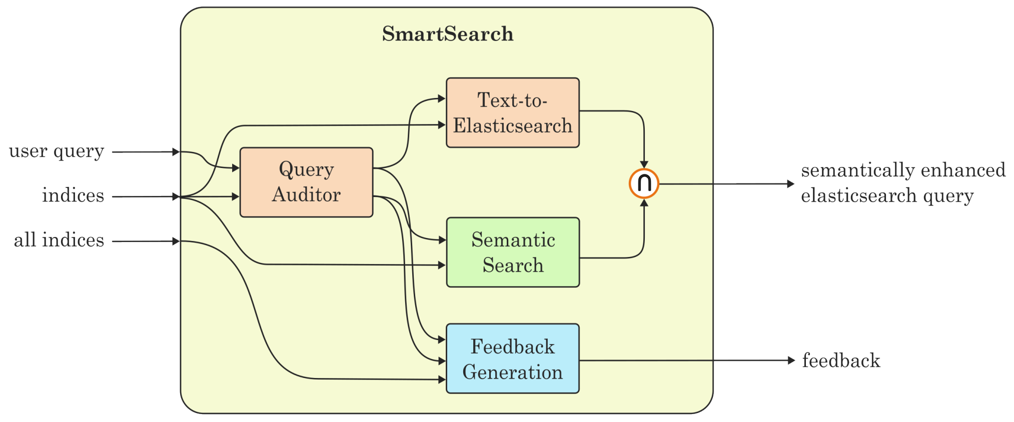

Figure 2.5: SmartSearch tool. The tool combines Query Auditor and Text-to-ES tools for syntactic query processing, and Semantic Search for semantic query processing, to provide a powerful and flexible search engine for log data.

By integrating these tools—Query Auditor, Text-to-ES, and Semantic Search—as seen in Figure 2.5, SmartSearch provides a powerful tool for log analytics, enabling users to perform complex queries that combine precise keyword matching with deep semantic understanding. Beyond its powerful capabilities, SmartSearch also offers a user-friendly interface, consistent with the other tools in the ModuGPT framework. This multi-step, chained approach delivers highly accurate and relevant results to users, making SmartSearch an indispensable tool for advanced log analysis.

Example: SmartSearch interface is examplified with the following execution, where the user query is processed to generate an Elasticsearch query that combines syntactic and semantic elements. The input includes the user query, the indices to search, and a flag used to format the feedback, and the output is the Elasticsearch query and a natural language feedback to the user.

```
Input Example

user query: "Hello! We have been encountering some network problems.
Were any ERROR level logs that may be related to this generated today?"
indices: ["*****-iis"]
all indices: False
```

```
Output Example

es query+: {"query":{"bool":{"filter":[{"term":{"level.keyword":"ERROR"
}},{"range":{"@timestamp":{"gte":"now/d","lt":"now/d+1d"}}}]}},
{"terms":{"message_pattern_id.keyword":[
"341a9df5-8d85-43fe-ae79-cf10bad2f932"]}}}
feedback: "Show logs that are ERROR level generated today related to
network problems from *****-iis"
```

Chapter 3

Implementation

This chapter discusses in detail the implementation of the ModuGPT framework[1]. It provides an overview of the development environment, the design of ModuGPT, and the key components and utilities that make up the framework.

3.1 Development Environment

3.1.1 Programming Language

The choice of programming language plays a key role in the efficiency, flexibility, and scalability of a project. Python was selected for this work due to its strong support for GenAI applications and its extensive ecosystem of libraries and frameworks. Python's high-level nature allows for rapid development and iteration, making it ideal for research and prototyping. While not as fast as low-level languages such as C++ or Rust, Python's performance is sufficient for this project, where most of the runtime is spent on LLM interactions and API calls. Additionally, Python aligns with Logmind's existing technology stack, guaranteeing consistency and smooth integration with current systems. This alignment simplifies deployment, reduces compatibility issues, and leverages the team's existing expertise.

3.1.2 Framework

Instead of using existing LLM-development frameworks like LangChain [52] or LlamaIndex [53], a custom framework called ModuGPT was developed. This decision was based on several key

[1]ModuGPT source code: `https://bitbucket.org/ketjuni2/modugpt_public/src/main/`

factors. First, existing frameworks often include unnecessary features and layers of abstraction that can add complexity and overhead, making the system harder to debug and maintain. ModuGPT was designed to avoid these issues by being streamlined and efficient. Second, the unique needs of this project, especially the customization required for LLM interactions in log analytics, demanded flexibility that general-purpose frameworks could not provide. ModuGPT was built to meet these specific needs without the constraints of broader frameworks.

Finally, ModuGPT was designed with modularity and extensibility in mind. Its modular structure allows for the easy addition of new tools and features as the project evolves. Each module within the DriverAgent handles a specific task, such as input validation or prompt generation, making it simple to develop new tools by extending existing components. This modular approach makes sure that the framework can easily adapt to future needs without major code changes.

3.1.3 Choice of LLMs

The choice of LLMs was crucial for the performance of the tools developed in this project. Initial experiments used Llama 2 [54] models (7B and 13B variants) on a laptop. These models, quantized to fit within available memory, performed adequately for simple tasks like basic text-to-SQL translation. However, as tasks became more complex, particularly with all text-to-ES and advanced text-to-SQL translation, the limitations of these models became evident, leading to poor performance and unreliable results. More powerful models like Llama 2 70B were not feasible due to the high computational costs. The need for cloud-based GPUs made them too expensive for Logmind's scale. This led to a shift towards more cost-effective, closed-source API-based models provided by OpenAI [55].

Several models were tested, including `gpt-3.5-turbo`, `gpt-3.5-turbo-instruct`, `gpt-4`, and `gpt-4-turbo`. Among these, `gpt-3.5-turbo-instruct` was the only text completion model, which allowed for the use of the leading word technique in the text-to-ES tool. The other models were chat models, with `gpt-4` being particularly effective for more complex tasks like insight-to-text generation and query auditing.

Recently, the introduction of OpenAI's `gpt-4o` and `gpt-4o-mini` offered considerably lower costs. Consequently, all tools, except text-to-ES, were migrated to `gpt-4o-mini`. The migrated tools showed improved performance at a reduced cost. The text-to-ES tool retained `gpt-3.5-turbo-instruct` to utilize the invaluable leading word technique.

3.2 Design of ModuGPT

The modularity of the ModuGPT framework is a central consideration of its design. Each module within the framework is designed to perform a specific function, ensuring that the development of new tools is both straightforward and efficient.

3.2.1 Modular Architecture

The DriverAgent[2] component orchestrates the flow of data through the ModuGPT pipeline. It is composed of several modules, each responsible for a distinct aspect of the LLM interaction process. These modules[3] are designed to be easily subclassed, allowing developers to create new tools by simply extending existing classes and combining them into a cohesive tool within the ModuGPT framework.

The LLM module[4] within ModuGPT is also highly modular, supporting integrations with various providers. Currently, the framework supports OpenAI [55] and Microsoft Azure [56], but the architecture is designed to allow easy extension to other providers, as well as the possibility of using local or cloud-based models. This abstraction allows developers to interact with different LLMs through a consistent API, regardless of the underlying model or provider.

3.2.2 Context and Config Classes

The Context class[5], a wrapper around a dictionary, is used to pass data between different components. It supports intuitive bracket notation for easy access while also handling validation, such as checking for key existence and data types. This centralized validation makes the system more robust and easier to maintain. All agents within ModuGPT and most modules use Context objects with a standardized interface, which maintains consistent data flow and reduces the chance of errors.

Similarly, the Config class[6] is another dictionary-based wrapper, but it focuses on managing configuration options. It allows for dot notation access, making it easier for developers to see available options and use IDE completions. Each module, tool, and utility in ModuGPT has its own specific Config class, which defines relevant configuration variables and applies consistent validation checks. The ConfigRetriever class further simplifies accessing these configurations, enhancing the overall developer experience.

[2]`https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/agent/driver.py`
[3]`https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/module/`
[4]`https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/module/llm/`
[5]`https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/context.py`
[6]`https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/config.py`

3.2.3 Database Integrations

Database integration is a key feature of ModuGPT, especially given the variety of databases used within Logmind. The framework provides a unified interface that simplifies interactions with different databases by abstracting their complex APIs.

ModuGPT includes several DatabaseEngine classes, each tailored to a specific type of database. Currently, Elasticsearch [57], MongoDB [58], and Clickhouse [59] are supported. These classes offer a consistent API, making it easier to work with the underlying databases. In addition, ModuGPT defines DatabaseOperations classes, which standardize operations like caching, configuration, embedding, evaluation, schema management, and example retrieval. These classes allow developers to switch between different database engines while having access to the same operations without needing to rewrite code.

In ModuGPT, each Database class[7] inherits from a DatabaseEngine and a DatabaseOperations class, creating a comprehensive interface for developers. This design makes sure that all necessary functionalities are encapsulated within a single class, simplifying the process of database interaction and reducing the learning curve for developers.

3.2.4 Additional Utilities

Beyond the core components and modules, ModuGPT implements a range of additional utilities that enhance the framework's functionality and developer experience. These utilities include:

- Embeddings[8]: Provides a unified interface for integrating different embedding APIs. Currently, OpenAI [55] and Microsoft Azure [56] embeddings are supported, with the system designed for easy extension to other providers.
- Errors[9]: Defines a hierarchy of exception classes specific to ModuGPT. All errors within the framework inherit from BaseModuGPTError, ensuring consistent and manageable error handling across the system.
- Messages[10]: Offers classes for System, User, and Assistant messages, along with functions for converting these messages into the appropriate formats for chat and text completion models.

[7] `https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/database/`
[8] `https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/embedding.py`
[9] `https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/error.py`
[10] `https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/message.py`

- Schema[11]: Manages the extraction and formatting of database schemas, optimizing them for use with LLMs by compressing and structuring the information to reduce token usage.
- Selectors[12]: Implements various example selectors, with the Semantic Similarity Selector being the primary tool for selecting relevant examples based on semantic similarity.
- Semantic Search[13]: Abstracts the complexities of semantic search, providing a clean interface for embedding queries, accessing vector stores, and retrieving the most relevant results.
- Tasks[14]: Facilitates the scheduling and execution of long-running tasks, improving efficiency in operations that require significant computational resources.
- Tokenizers[15]: Interfaces with different tokenizers, currently integrating with Tiktoken for OpenAI models, to manage prompt length and maintain compatibility with LLM constraints.
- Vectorstores[16]: Provides an interface for different vector stores, with Facebook AI Similarity Search (FAISS) [33] currently supported and the architecture designed for easy extension to other libraries.

These utilities greatly expand the capabilities of ModuGPT, providing developers with the tools needed to implement sophisticated LLM-driven applications while maintaining ease of use and flexibility.

[11]https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/schema.py
[12]https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/selector.py
[13]https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/semantic_search.py
[14]https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/task.py
[15]https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/tokenizer.py
[16]https://bitbucket.org/ketjuni2/modugpt_public/src/main/modugpt/util/vectorstore.py

Chapter 4

Evaluation

Evaluating *natural language generation* (NLG) is essential for understanding the effectiveness of prompts and prompt engineering, but it presents significant challenges due to the complexity of natural language. A variety of methods exist for assessing NLG tasks. The most reliable approach is human evaluation, where human judges assess the quality of the generated text. Although this method is considered the gold standard, it is both expensive and time-consuming, and it is also subject to various biases [60, 61].

Traditional evaluation techniques often rely on n-gram overlap metrics, such as BLEU [62], ROUGE [63], and METEOR [64]. However, these metrics are inadequate for evaluating NLG tasks due to their low correlation with human judgments [65]. To address these shortcomings, deep learning advancements have led to the development of BERT-based methods like BERTScore [66], which show better alignment with human evaluations. Despite these improvements, BERT-based methods still face limitations in their performance and the scope of their application [67].

Beyond traditional metrics and human evaluation, recent advancements in LLMs have introduced new approaches to NLG evaluation. One notable method is using an *LLM-as-a-judge*. Studies suggest that LLMs can evaluate NLG tasks at a level comparable to human judges [68, 69]. However, this approach is not without its limitations. For instance, LLMs often prefer text generated by the same model [70], exhibit a tendency to favor longer responses [71], and display a position bias, where options presented earlier in the prompt are likely to receive higher scores [72]. These biases and limitations must be carefully considered when using LLMs as judges in NLG evaluation.

4.1 Evaluation Setup

The evaluation aims to verify the effectiveness of the design and implementation by comparing the performance of the developed tools—Insight-to-Text, Text-to-ES, and SmartSearch (referred to as *eval agents*)—against *control agents.* The evaluation process employs a combination of custom metrics and LLM-as-a-judge methods to comprehensively assess the tools' accuracy and reliability. The evaluation is conducted on carefully curated datasets to ensure the robustness of the results across diverse log sources and query types.

4.1.1 Evaluation Methods

Based on the nature of the task and the type of output generated by the tools, two distinct evaluation methods are utilized: *Reference Evaluation* and *A/B Testing.*

Reference Evaluation is applied to tasks where outputs can be objectively compared to a predefined set of correct answers, known as a *golden dataset.* This method is preferred as it does not rely on LLM-as-a-judge and provides a more reliable and consistent evaluation process. Reference Evaluation is used for evaluating the Text-to-ES and SmartSearch tools. Two custom metrics, *SetSimilarityMetric* and *QuerySimilarityMetric*, are developed to quantify the accuracy of the generated outputs.

SetSimilarityMetric compares two sets of items using the Jaccard similarity index [73]. As shown in Equation 4.1, the Jaccard index $J(A, B)$ measures the similarity between two sets A and B by calculating the ratio of the size of their intersection to the size of their union. A value of 1 indicates that the sets are identical, while a value of 0 indicates no overlap. This provides a quantitative measure of how closely the generated results match the expected results.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.1}$$

QuerySimilarityMetric assesses the accuracy of tool-generated queries by comparing the logs they retrieve against those from the golden queries. For filter queries, both queries are run on the database, and the resulting sets of documents are compared using the Jaccard similarity index, as described above. QuerySimilarityMetric also includes a specialized evaluation for aggregation queries, supporting most *metric* and *bucket* aggregations. For metric aggregations, the normalized absolute difference between the values retrieved by the generated query and the golden query is calculated, as shown in Equation 4.2. For bucket aggregations, the metric evaluates each key individually by applying the same formula to that key's value, then averaging the scores across all keys to provide an overall score for the aggregation query. Finally, due to the complexity of evaluating them, *pipeline* aggregations, some specific metric and bucket aggregations, and multi-level aggregations are not supported by the QuerySimilarityMetric.

$$\text{Metric Aggregation Score} = \frac{|v_1 - v_2|}{\max(v_1, v_2)} \tag{4.2}$$

A value of 1 in the QuerySimilarityMetric for both filter and aggregation queries signifies that the generated and golden queries have identical results, while a value of 0 means there is no commonality. This metric offers a precise and clear assessment of the alignment between the generated query and the expected query.

A/B Testing is employed for tasks where output quality is subjective and not easily measurable against a strict reference. A/B testing utilizes LLM-as-a-judge to evaluate the generated texts. In this evaluation method, the control and eval agents' outputs are presented to the LLM judge separately, and the judge assigns a score from 1 to 5 to each text based on task-specific criteria. The scores are then compared to determine which agent produced higher-quality outputs. A/B testing is used for evaluating the Insight-to-Text tool, as the quality of the generated explanations and recommendations is subjective. However, it is important to address the limitations and biases associated with LLM-as-a-judge evaluation, such as model preference, positional bias, and length preference, to ensure the reliability of the results.

LLM-as-a-judge evaluation is conducted using a different model (`gpt-4o`) from the one used to generate the outputs (`gpt-4o-mini`). This is done to avoid model preference bias. To mitigate positional bias, the generated texts from the control and eval agents are presented to the judge LLM in separate prompts, instead of being presented to the model in a single prompt and asking it to choose the better one. This approach prevents the judge from having a positional preference for the control or eval agent outputs. Finally, the length preference bias is not a concern in this evaluation, as the eval agent outputs are consistently shorter than the control agent outputs, putting the eval agent at a disadvantage.

4.1.2 Consistency and Reproducibility

Throughout the evaluation process, careful attention is given to maintaining the consistency and reproducibility of results. For each task, the `seed` parameter is set to a fixed value, and the `temperature` parameter is set to 0 to maintain consistency across runs. According to the OpenAI API [55], with these settings, the outputs of the LLM are deterministic, as long as the `system_fingerprint` response parameter remains the same. During the evaluation, this response parameter is monitored to make sure that the same model iteration is used consistently, allowing for deterministic results. Therefore, all tool results are assumed to be deterministic in the evaluation process. This assumption is useful for reducing the cost of API usage, that is, to avoid having to assess the uncertainty of the LLM outputs using multiple repeated experiments.

To achieve reliable LLM-as-a-judge evaluations, relying only on a single score from the LLM is not sufficient. Instead, the judge is run multiple (10) times with a `temperature` setting of 0.5 to introduce some variation. The scores from these runs are then averaged, resulting in a more dependable assessment of the generated texts. This approach helps to reduce the impact of randomness in the LLM's scoring and provides a more accurate evaluation of the text quality.

4.1.3 Dataset Preparation

The evaluation of the tools is conducted using datasets specifically curated to cover a wide range of log types, query complexities, and system behaviors. These datasets include logs from Network Firewall logs, Windows Event logs, Networking logs, Java application logs, and Web Server logs, ensuring that the evaluation results generalize well across different scenarios commonly encountered in log analytics.

- Insight-to-Text: The dataset consists of 100 insights drawn from the five different log sources mentioned above, with 20 insights from each source. This diverse mix is selected to assess the tools across a spectrum of log formats and content.
- Text-to-ES: A dataset of 100 natural language queries is prepared, each paired with a golden Elasticsearch query and the index to run the query on. The number of filtering and aggregation queries is equal across all sources, with 10 filter and 10 aggregation queries per source, allowing the evaluation to cover different aspects of Elasticsearch query generation. It is important to note that only single-level aggregations are used in this dataset due to the complexity of evaluating multi-level aggregations.
- SmartSearch: The dataset for SmartSearch includes 100 examples, with each example containing a natural language query, a golden output containing an Elasticsearch query and the most relevant pattern IDs. As with the other tasks, the examples are drawn from the five different log sources and include both filtering and single-level aggregation queries.

These datasets will not be made available, as the queries and insights contain sensitive customer information. This decision protects the privacy and confidentiality of the data used in the evaluation process.

4.2 Insight-to-Text Evaluation

The Insight-to-Text tool is evaluated using A/B Testing, comparing the output of the control agent (Section A.4), which lacks prompt engineering, with that of the eval agent (Section A.1), which utilizes carefully crafted prompts. The judge (Section A.7) is asked to score the generated outputs based on specific criteria: accuracy, clarity, conciseness, and relevance.

Metric	**Value**
Eval Wins	95
Draws	1
Control Wins	4
Eval Mean (Std)	**4.17 (± 0.40)**
Control Mean (Std)	**3.28 (± 0.38)**

Table 4.1: Insight-to-Text Results. The table shows the performance comparison of the control and eval agents. The first three rows show the count of each outcome. The last two rows show the mean scores and standard deviations across all examples. Eval agent consistently beats the control agent.

Evaluation results (Table 4.1) show that the eval agent outperforms the control agent in 95 out of 100 examples. Control agent wins in 4 examples, and there is a single draw. The number of draws is low due to the fact that judge agent is ran 10 times for each example, and the scores are averaged. The mean score of the eval agent is 4.17 (± 0.40), while the mean score of the control agent is 3.28 (± 0.38). These results indicate that the eval agent, which incorporates prompt engineering, produces higher-quality outputs compared to the control agent, which lacks this feature.

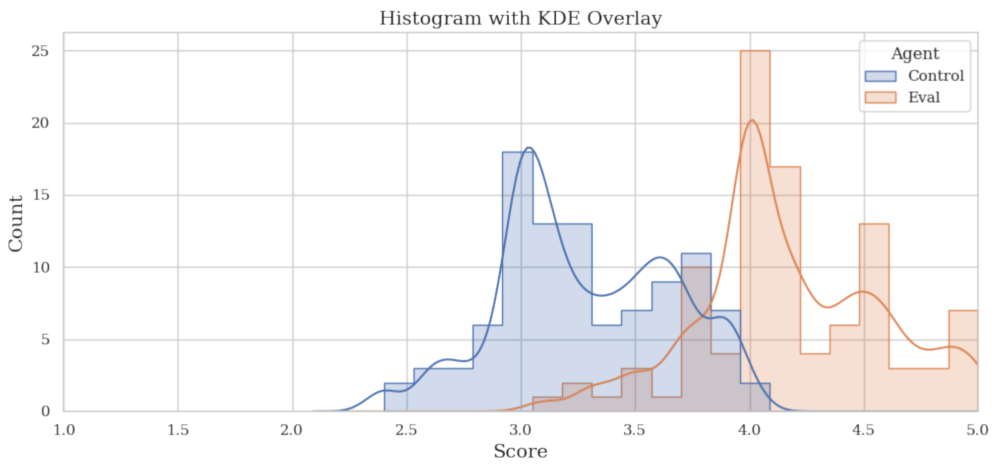

Figure 4.1: Insight-to-Text Results. The figure shows the distribution of scores for the eval and control agents received from the judge. The eval agent's scores are consistently higher.

Figure 4.1 illustrates the distribution of scores for the eval and control agents. Eval agent score distribution is skewed towards higher scores, with a peak around 4.0, while the control agent peaks around 3.0. It is interesting to note that neither agent received scores below 2.0, suggesting either that the generated outputs are of reasonable quality or that the judge is lenient in scoring. It is also important to note that eval agent received the perfect 5.0 score 3 times. This indicates that the output of the eval agent got scored 5/5 in all 10 of the independent judge runs for those specific examples, which is a strong indicator of the high quality of the generated text. The control agent, on the other hand, did not receive a perfect score in any of the examples. This further supports the conclusion that the eval agent outperforms the control agent in generating high-quality insights.

4.3 Text-to-ES Evaluation

The Text-to-ES tool is assessed using the Reference Evaluation method, leveraging the QuerySimilarityMetric to compare the results of queries generated by the control agent (Section A.5), which lacks prompt engineering, with those generated by the evaluation agent (Section A.2), which incorporates prompt engineering. Resulting queries are then separately compared against the golden queries to evaluate accuracy.

Category	**FM**	**PM**	**NM**	**Mean (Std)**
Eval - Filter	50	0	0	1.00 (±0.00)
Eval - Aggregation	50	0	0	1.00 (±0.00)
Eval - Total	**100**	**0**	**0**	**1.00 (±0.00)**
Control - Filter	3	14	33	0.21 (±0.34)
Control - Aggregation	10	5	35	0.25 (±0.43)
Control - Total	**13**	**19**	**68**	**0.23 (±0.39)**

Table 4.2: Text-to-ES Results. The table shows the performance comparison of the control and eval agents compared to the golden queries. The eval agent completely outperforms the control agent in both filter and aggregation queries. FM: Fully Matched, PM: Partially Matched, NM: Not Matched.

In the Table 4.2, perfect matches indicate the generated queries are identical to the golden queries, receiving a score of 1.00. No matches mean the generated queries are completely wrong, receiving a score of 0.00. Partial matches indicate some overlap between the generated and golden queries, receiving a score between 0.00 and 1.00.

It is evident from the results that the eval agent significantly outperforms the control agent in both filter and aggregation queries. The eval agent achieves a perfect score of 1.00 (±0.00) in both categories, perfectly translating all 100 of the natural language queries into elasticsearch queries. In contrast, the control agent scores 0.21 (±0.34) for filter queries and

0.25 (±0.43) for aggregation queries, demonstrating a poor performance. Control agent results show that only 13 out of 100 queries are fully matched and 19 are partially matched, while the remaining 68 are completely wrong. Out of the 68 failures, 4 were due to model producing an invalid json object, 16 were due to model producing an invalid Elasticsearch query, and rest were due to model producing an incorrect Elasticsearch query. This poor performance can be attributed to several factors, such as the lack of schema knowledge, absence of proper guidance, and missing examples to learn from.

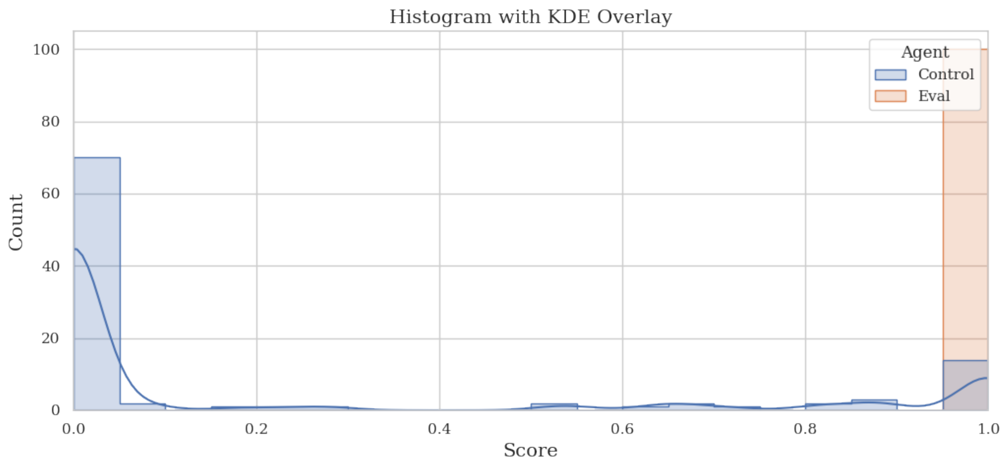

Figure 4.2: Text-to-ES Results. The figure shows the distribution of scores for the eval and control agents calculated by comparing the generated queries against the golden queries. The eval agent recieves a perfect score, while the control agent's distribution has a high peak around 0.0.

Figure 4.2 clearly demonstrates the poor performance of the control agent compared to the eval agent. The control agent's score distribution is heavily skewed towards lower scores, with a peak around 0.0. Even though eval agent received a perfect score of 1.0 in all examples, it is important to note that this is partly possible due to the nature of the dataset, which does not contain multi-level aggregations. It has been empirically observed that the eval agent can struggle with complex multi-level aggregations, ocasionaly producing incorrect queries. However, due to the absence of such queries in the dataset it achieves a perfect score in this evaluation. Nevertheless, this does not diminish the significance of the results, as the eval agent's performance on filter queries and single-level aggregations is still a strong indicator of its effectiveness. This evaluation demonstrates the effectiveness of prompt engineering in improving the accuracy of Elasticsearch queries generated from natural language queries.

4.4 SmartSearch Evaluation

The evaluation of SmartSearch is conducted using the Reference Evaluation method in two phases, reflecting the dual nature of the tool. This evaluation compares the performance of eval agent (Section A.3 and Section A.2) utilizing prompt chaining with that of a control agent (Section A.6), which employs the best prompt engineering techniques but only makes a single call to the LLM. This comparison aims to assess the impact of prompt chaining on the tool's effectiveness across its syntactic and semantic components.

- Syntactic Part: This part is evaluated using the QuerySimilarityMetric, comparing the control agent (single LLM call) with the evaluation agent (prompt chaining). The comparison focuses on the accuracy of the Elasticsearch queries generated by each agent.
- Semantic Part: The semantic component is evaluated using the SetSimilarityMetric, comparing the pattern IDs retrieved by each agent with the golden pattern IDs. The comparison focuses on how effectively each agent captures the broader meaning of the queries.

Category	**FM**	**PM**	**NM**	**Mean (Std)**
Eval - Filter	34	8	8	0.74 (±0.42)
Eval - Aggregation	28	7	15	0.60 (±0.47)
Eval - Total	**62**	**15**	**23**	**0.67 (±0.45)**
Control - Filter	11	3	36	0.24 (±0.43)
Control - Aggregation	9	2	39	0.20 (±0.40)
Control - Total	**20**	**5**	**75**	**0.22 (±0.41)**

Table 4.3: Results for the Syntactic Part of SmartSearch. The table shows the performance comparison of the control and eval agents for the syntactic part of the SmartSearch task. Eval agent heavily outperforms the control agent across both query categories. FM: Fully Matched, PM: Partially Matched, NM: Not Matched.

Detecting the syntactic part of a natural language query, and translating it to an Elasticsearch query is a challenging task that requires multi-step reasoning. The results in Table 4.3 support this claim, showing that the eval agent, which splits the task into subtasks with prompt chaining, outperforms the control agent, which attempts to solve the task in a single step. The eval agent achieves a mean score of 0.67 (±0.45), with 62 fully matched and 15 partially matched queries. In contrast, the control agent scores 0.22 (±0.41), with only 20 fully matched and 5 partially matched queries. It is possible to observe that aggregation queries are more challenging for both agents, especially for the eval agent, which encounters a significant drop in performance at 0.60 (±0.47) compared to filter queries at 0.74 (±0.42). This drop in performance is peculiar and requires further investigation to determine the root cause. Another interesting observation is that out of control agent's 75 failures, 3 were due to mode generating an invalid json object, even though the model benefits from the same prompt engineering techniques as the eval agent, which never produced an invalid json object. This suggests that splitting the task into subtasks can help the model follow the requested structure more accurately.

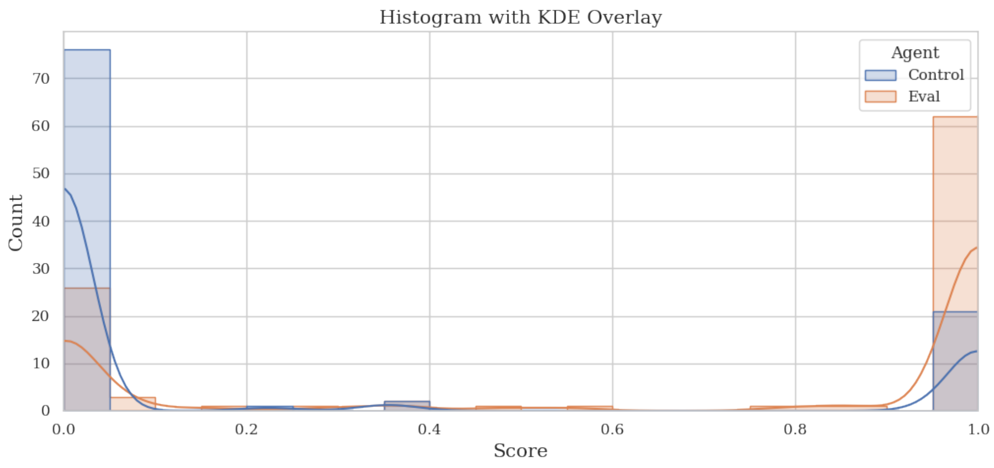

Figure 4.3: Results for the Syntactic Part of SmartSearch. The figure shows the distribution of scores for the eval and control agents calculated by comparing the generated Elasticsearch queries against the golden queries. The eval agent's distribution peaks around 1.0, while the control agent's distribution peaks around 0.0.

One important aspect of both agents' results is the low number of partial scores. This suggests that there was rarely a minor misunderstanding of the natural language query, and the model either fully understood the query or failed to understand it completely, as evidenced by Figure 4.3. The eval agent's score distribution is skewed towards higher scores, with a high peak around 1.0, while the control agent's distribution has a high peak around 0.0. This indicates that the eval agent is more consistent in generating accurate Elasticsearch queries compared to the control agent. The results of the syntactic part evaluation demonstrate the effectiveness of prompt chaining in improving the accuracy of tasks that require multi-step reasoning.

Category	FM	PM	NM	Mean (Std)
Eval	22	66	12	**0.66 (±0.34)**
Control	23	44	33	**0.49 (±0.42)**

Table 4.4: Results for the Semantic Part of SmartSearch. The table shows the performance comparison of the control and eval agents for the semantic part of the SmartSearch task. These results are the closest between the two agents, with the eval agent still outperforming the control agent. FM: Fully Matched, PM: Partially Matched, NM: Not Matched.

The closer gap between the eval and control agents in the semantic evaluation (Table 4.4) is expected since this task doesn't require multi-step reasoning like the syntactic part. The eval agent scores a mean of 0.66 (±0.34) with 22 fully matched and 66 partially matched queries, while the control agent scores 0.49 (±0.42) with 23 fully matched and 44 partially matched queries. Despite not needing multi-step reasoning, the eval agent still slightly outperforms, suggesting that the control agent's heavier workload in the syntactic part may impact its semantic performance.

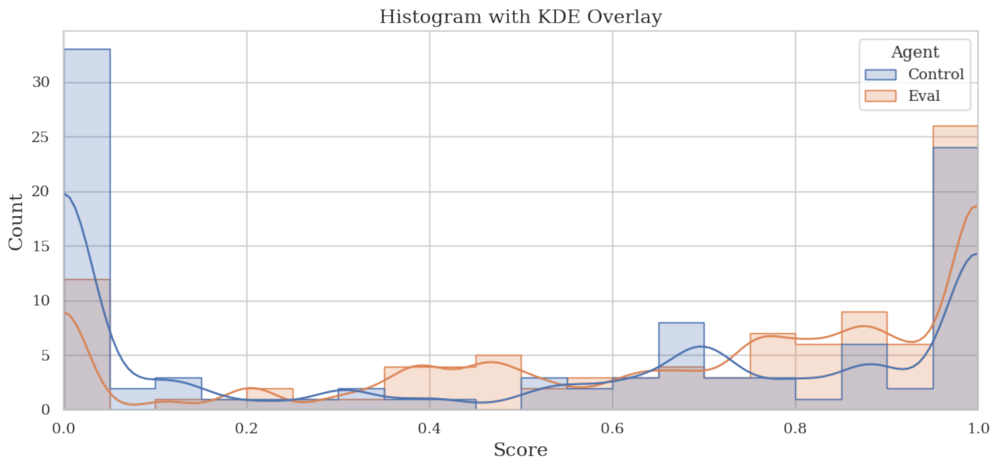

Figure 4.4: Results for the Semantic Part of SmartSearch. The figure shows the distribution of scores for the eval and control agents calculated by comparing the generated pattern IDs against the golden pattern IDs. Control agent has a considerably higher peak around 0.0 compared to the eval agent.

The higher number of partial matches in the semantic part (Figure 4.4) is expected due to the challenge of fully matching pattern IDs through semantic search. Especially eval agent, with 66 partial matches, demonstrates that it can partially capture the semantic meaning of the natural language query. Overall, the eval agent outperforms the control agent in both syntactic and semantic parts of the SmartSearch task, highlighting the effectiveness of prompt chaining for complex multi-step reasoning.

Chapter 5

Conclusion

The rapid expansion of modern IT systems has led to the generation of large amounts of log data, making efficient log analytics increasingly important. Traditional methods for log analysis, which often rely on manual inspection or simple rule-based approaches, struggle to keep up with the volume, complexity, and variability of logs in large-scale systems. This thesis has addressed these challenges by integrating GenAI models with traditional log analysis techniques, introducing a suite of tools within the ModuGPT framework designed to enhance the accessibility, interpretability, and actionable value of log data.

The key contributions of this thesis include the development of the *ModuGPT* framework, a modular system that simplifies the creation of applications that interact with LLMs. The ModuGPT framework introduces *Prompt Elements*, a structured catalog of reusable components that streamline prompt engineering and improve the consistency and effectiveness of LLM interactions. The framework also includes three specialized tools: Insight-to-Text, Text-to-ES, and SmartSearch, each designed to address specific challenges in log analytics.

Insight-to-Text enriches log insights by generating natural language explanations and actionable recommendations, making it easier for users to understand and act on log data. The evaluation of Insight-to-Text showed that the evaluated agent, leveraging advanced prompt engineering, consistently outperforms the control agent across key metrics of accuracy, clarity, conciseness, and relevance. This result highlights the importance of well-crafted prompts in guiding LLMs to generate high-quality outputs.

Text-to-ES provides a natural language interface for querying Elasticsearch, addressing the complexity of writing accurate and efficient queries. The evaluation showed that the eval agent, which benefited from prompt engineering and schema knowledge, outperformed the control agent in generating both filter and aggregation queries. The results highlighted the challenges of translating natural language into Elasticsearch queries and the effectiveness of the proposed solution in addressing these challenges.

SmartSearch enhances traditional search functionality by associating semantic search and keyword-based methods. This tool uses prompt chaining to effectively split natural language queries into syntactic and semantic components, increasing the accuracy of both parts. The evaluation of SmartSearch showed the effectiveness of prompt chaining in handling complex, multi-step reasoning tasks, with the eval agent outperforming the control agent in processing both syntactic and semantic components of the task.

In conclusion, these tools represent a significant advancement in the field of log analytics, addressing the limitations of existing methods and making log data more accessible and actionable for a broader range of users. The integration of Generative AI models, particularly LLMs, into the log analysis workflow offers a powerful new approach to understanding and managing large-scale log data. By improving the contextualization and usability of log analysis tools, this work aligns with the evolving demands of AI-driven IT operations.

5.1 Future Work

Future work could explore several promising directions:

1. Ablation Studies: One area for future research is conducting ablation studies to understand the impact of individual prompt elements on model performance. By systematically removing or altering specific elements, we can gain insights into their contributions and explore how the order of prompt elements affects the outcomes. This analysis could lead to more refined and effective prompt engineering techniques.

2. Extension to Prompt Elements: Another avenue for exploration is the development of additional generic prompt elements that can be applied across various tasks. Identifying and implementing new prompt elements could expand the flexibility and effectiveness of the ModuGPT framework, allowing it to address an even broader range of log analytics challenges.

3. Fine-Tuning: Investigating the effect of fine-tuning LLMs on performance could be of interest. By comparing the performance of fine-tuned models against non-fine-tuned models, we can assess the benefits and trade-offs of this approach. Additionally, exploring different fine-tuning techniques across various models could help optimize their performance in specific log analytics tasks.

4. Improving Evaluation: Future work could also focus on refining the evaluation methods used in this thesis. Incorporating considerations of LLM response speed and cost into the evaluation process could provide a more comprehensive assessment of the tools'

practicality and efficiency. For generated queries, it would be useful to measure not only their correctness but also the speed and optimization of the queries, ensuring that the tools not only generate accurate results but are also performant in real-world scenarios.

5. Topic Generation: Exploring methods for clustering and labeling log patterns represents another promising direction. Developing a tool that can generate groups of topics from a customer's dataset and assign every insight to one of these topics could significantly improve the contextualization and understanding of log data. This topic generation tool could enhance the interpretability of insights, making log analytics more actionable and user-friendly.

Bibliography

[1] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. “Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (2013), pp. 1245–1255.

[2] Andriy Miranskyy, Abdelwahab Hamou-Lhadj, Enzo Cialini, and Alf Larsson. “Operational-log analysis for big data systems: Challenges and solutions”. In: *IEEE Software* 33.2 (2016), pp. 52–59.

[3] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. “Experience report: System log analysis for anomaly detection”. In: *27th International Symposium on Software Reliability Engineering (ISSRE).* IEEE. 2016, pp. 207–218.

[4] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. “A lightweight algorithm for message type extraction in system application logs”. In: *IEEE Transactions on Knowledge and Data Engineering* 24.11 (2011), pp. 1921–1936.

[5] Masayoshi Mizutani. “Incremental mining of system log format”. In: *10th International Conference on Services Computing.* IEEE. 2013, pp. 595–602.

[6] Min Du and Feifei Li. “Spell: Streaming parsing of system event logs”. In: *16th International Conference on Data Mining (ICDM).* IEEE. 2016, pp. 859–864.

[7] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. “Drain: An online log parsing approach with fixed depth tree”. In: *24th International Conference on Web Services (ICWS).* IEEE. 2017, pp. 33–40.

[8] Ying Fu, Meng Yan, Jian Xu, Jianguo Li, Zhongxin Liu, Xiaohong Zhang, and Dan Yang. “Investigating and improving log parsing in practice”. In: *30th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM. 2022, pp. 1566–1577.

[9] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. “LILAC: Logparsing using LLMs with adaptive parsing cache”. In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 137–160.

[10] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. “Deeplog: Anomaly detection and diagnosis from system logs through deep learning”. In: *SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1285–1298.

[11] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and et al. “Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs”. In: *28th International Joint Conference on Artificial Intelligence*. IJCAI. 2019, pp. 4739–4745.

[12] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, and et al. “Robust log-based anomaly detection on unstable log data”. In: *27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2019, pp. 807–817.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of deep bidirectional transformers for language understanding”. In: *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. ACL. 2019, pp. 4171–4186.

[14] Haixuan Guo, Shuhan Yuan, and Xintao Wu. “LogBERT: Log anomaly detection via BERT”. In: *International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[15] Yukyung Lee, Jina Kim, and Pilsung Kang. “LAnoBERT: System log anomaly detection based on BERT masked language model”. In: *Applied Soft Computing* 146 (2023), p. 110689.

[16] Crispin Almodovar, Fariza Sabrina, Sarvnaz Karimi, and Salahuddin Azad. “LogFiT: Log anomaly detection using fine-tuned language models”. In: *IEEE Transactions on Network and Service Management* 21.2 (2024), pp. 1715–1723.

[17] Stuart J Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Pearson, 2016.

[18] Iqbal H Sarker. “Machine learning: Algorithms, real-world applications and research directions”. In: *SN computer science* 2.3 (2021), p. 160.

[19] Simon Haykin. *Neural networks: A comprehensive foundation*. Prentice Hall PTR, 1998.

[20] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[21] Prashant Johri, Sunil K Khatri, Ahmad T Al-Taani, Munish Sabharwal, Shakhzod Suvanov, and Avneesh Kumar. “Natural language processing: History, evolution, application, and future work”. In: *3rd International Conference on Computing Informatics and Networks: ICCIN 2020*. Springer. 2021, pp. 365–375.

[22] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, and et al. "A survey on evaluation of large language models". In: *ACM Transactions on Intelligent Systems and Technology* 15.3 (2024), pp. 1–45.

[23] Peter F Brown, Vincent J Della Pietra, Peter V deSouza, Jenifer C Lai, and Robert L Mercer. "Class-based n-gram models of natural language". In: *Computational linguistics* 18.4 (1992), pp. 467–480.

[24] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536.

[25] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017), pp. 5998–6008.

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

[28] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and et al. "Training language models to follow instructions with human feedback". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 27730–27744.

[29] OpenAI. "GPT-4 technical report". In: *arXiv preprint arXiv:2303.08774* (2023).

[30] Paul F Christiano, Jan Leike, Tom B Brown, Miljan Martic, Shane Legg, and Dario Amodei. "Deep reinforcement learning from human preferences". In: *Advances in Neural Information Processing Systems* 30 (2017), pp. 4302–4310.

[31] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. "A prompt pattern catalog to enhance prompt engineering with ChatGPT". In: *arXiv preprint arXiv:2302.11382* (2023).

[32] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and et al. "Least-to-most prompting enables complex reasoning in large language models". In: *arXiv preprint arXiv:2205.10625* (2022).

[33] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: *IEEE Transactions on Big Data* 7.3 (2019), pp. 535–547.

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality". In: *Advances in Neural Information Processing Systems* 26 (2013), pp. 3111–3119.

[35] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. "Next-generation database interfaces: A survey of LLM-based text-to-SQL". In: *arXiv preprint arXiv:2406.08426* (2024).

[36] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. "Evaluating the text-to-SQL capabilities of large language models". In: *arXiv preprint arXiv:2204.00498* (2022).

[37] Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S Yu. "A comprehensive evaluation of ChatGPT's zero-shot text-to-SQL capability". In: *arXiv preprint arXiv:2303.13547* (2023).

[38] Shuaichen Chang and Eric Fosler-Lussier. "How to prompt LLMs for text-to-SQL: A study in zero-shot, single-domain, and cross-domain settings". In: *arXiv preprint arXiv:2305.11853* (2023).

[39] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and et al. "Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs". In: *Advances in Neural Information Processing Systems* 36 (2023).

[40] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. "Text-to-SQL empowered by large language models: A benchmark evaluation". In: *arXiv preprint arXiv:2308.15363* (2023).

[41] Shuaichen Chang and Eric Fosler-Lussier. "Selective demonstrations for cross-domain text-to-SQL". In: *arXiv preprint arXiv:2310.06302* (2023).

[42] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. "Enhancing text-to-SQL capabilities of large language models: A study on prompt design strategies". In: *Findings of the Association for Computational Linguistics: EMNLP*. ACL. 2023, pp. 14935–14956.

[43] Ruoxi Sun, Sercan Ö Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and et al. "SQL-PaLM: Improved large language model adaptation for text-to-SQL (extended)". In: *arXiv preprint arXiv:2306.00739* (2023).

[44] Mohammadreza Pourreza and Davood Rafiei. "DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction". In: *Advances in Neural Information Processing Systems* 36 (2023).

[45] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and et al. "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task". In: *arXiv preprint arXiv:1809.08887* (2018).

[46] Chenyan Xiong, Russell Power, and Jamie Callan. “Explicit semantic ranking for academic search via knowledge graph embedding”. In: *26th International Conference on World Wide Web*. IW3C2. 2017, pp. 1271–1279.

[47] Qingyao Ai, Yongfeng Zhang, Keping Bi, Xu Chen, and W Bruce Croft. “Learning a hierarchical embedding model for personalized product search”. In: *40th International SIGIR Conference on Research and Development in Information Retrieval*. ACM. 2017, pp. 645–654.

[48] Niklas Muennighoff. “SGPT: GPT sentence embeddings for semantic search”. In: *arXiv preprint arXiv:2202.08904* (2022).

[49] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. “deGraphCS: Embedding variable-based flow graph for neural code search”. In: *ACM Transactions on Software Engineering and Methodology* 32.2 (2023), pp. 1–27.

[50] Sondos Mahmoud Bsharat, Aidar Myrzakhan, and Zhiqiang Shen. “Principled instructions are all you need for questioning LLaMA-1/2, GPT-3.5/4”. In: *arXiv preprint arXiv:2312.16171* (2023).

[51] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. “Lost in the middle: How language models use long contexts”. In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 157–173.

[52] LangChain. *LangChain*. Accessed: 2024-07-22. URL: `https://www.langchain.com/`.

[53] LlamaIndex. *LlamaIndex*. Accessed: 2024-07-22. URL: `https://www.llamaindex.ai/`.

[54] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).

[55] OpenAI. *OpenAI API*. Accessed: 2024-07-30. URL: `https://platform.openai.com/docs/api-reference/introduction/`.

[56] Microsoft Corporation. *Azure OpenAI Service*. Accessed: 2024-07-30. URL: `https://learn.microsoft.com/en-us/azure/ai-services/openai/`.

[57] Elastic NV. *Elasticsearch*. Accessed: 2024-08-02. URL: `https://www.elastic.co/elasticsearch/`.

[58] MongoDB Inc. *MongoDB*. Accessed: 2024-08-02. URL: `https://www.mongodb.com/`.

[59] ClickHouse Inc. *ClickHouse*. Accessed: 2024-08-02. URL: `https://clickhouse.com/`.

[60] Chris van der Lee, Albert Gatt, Emiel van Miltenburg, and Emiel Krahmer. “Human evaluation of automatically generated text: Current trends and best practice guidelines”. In: *Computer Speech & Language* 67 (2021), p. 101151.

[61] Jan Deriu, Alvaro Rodrigo, Arantxa Otegi, Guillermo Echegoyen, Sophie Rosset, Eneko Agirre, and Mark Cieliebak. "Survey on evaluation methods for dialogue systems". In: *Artificial Intelligence Review* 54 (2021), pp. 755–810.

[62] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "BLEU: A method for automatic evaluation of machine translation". In: *40th annual meeting of the Association for Computational Linguistics*. ACL. 2002, pp. 311–318.

[63] Chin-Yew Lin. "ROUGE: A package for automatic evaluation of summaries". In: *Text Summarization Branches Out*. ACL. 2004, pp. 74–81.

[64] Satanjeev Banerjee and Alon Lavie. "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments". In: *Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. ACL. 2005, pp. 65–72.

[65] Elior Sulem, Omri Abend, and Ari Rappoport. "BLEU is not suitable for the evaluation of text simplification". In: *arXiv preprint arXiv:1810.05995* (2018).

[66] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. "BERTScore: Evaluating text generation with BERT". In: *arXiv preprint arXiv:1904.09675* (2019).

[67] Mingqi Gao, Xinyu Hu, Jie Ruan, Xiao Pu, and Xiaojun Wan. "LLM-based NLG evaluation: Current status and challenges". In: *arXiv preprint arXiv:2402.01383* (2024).

[68] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. "ChatGPT outperforms crowd workers for text-annotation tasks". In: *Proceedings of the National Academy of Sciences* 120.30 (2023), e2305016120.

[69] Petter Törnberg. "ChatGPT-4 outperforms experts and crowd workers in annotating political twitter messages with zero-shot learning". In: *arXiv preprint arXiv:2304.06588* (2023).

[70] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. "G-EVAL: NLG evaluation using GPT-4 with better human alignment". In: *arXiv preprint arXiv:2303.16634* (2023).

[71] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and et al. "Judging LLM-as-a-judge with MT-bench and chatbot arena". In: *Advances in Neural Information Processing Systems* 36 (2024).

[72] Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. "Large language models are not fair evaluators". In: *arXiv preprint arXiv:2305.17926* (2023).

[73] Paul Jaccard. "Étude comparative de la distribution florale dans une portion des Alpes et des Jura". In: *Bull Soc Vaudoise Sci Nat* 37 (1901), pp. 547–579.

Appendix A

Prompts

A.1 Insight-to-Text

```
System Message

You are a highly intelligent and skilled Site Reliability Engineer. You
are specialized in analyzing logs from {source_name}. {source_desc}.
You will be given a log template and additional log details. <*>
indicates a variable token in the log template. Your task is to analyze
the log template and utilize the additional log details to provide
explanation and recommendation. You are talking to a member of the IT
team. You must follow the rules which are delimited by triple hash
signs ###
Explanation should clearly describe the meaning and potential causes
Explanation should clarify the abbreviations and technical terms
Explanation should describe what the methods and errors are
Explanation should be at most 4 sentences
Recommendation should be relevant for the given log template
Recommendation should avoid seeking assistance from others
Recommendation should be at most 4 sentences
###
```

```
User Message

Additional Log Details: {extra_info}
Log Pattern: {title}
```

A.2 Text-to-ES

```
System Message

You are a highly intelligent and skilled Elasticsearch query creator.
You will be given a natural language query. Your task is to translate
the natural language query into an Elasticsearch query
(version={es_version}). If you cannot create a valid Elasticsearch
query, then you must output the following: {{"FAILED"}}. You must
follow the rules which are delimited by triple hash signs ###
Wrap all the queries in filter context
Retrieve only the necessary fields
Output only the elasticsearch dictionary
Compress the dictionary by removing whitespaces
### The query will run on a database whose schema is delimited by
triple backticks. ```
{schema}
```
```

```
User Message

{examples[0].input}
```

```
Assistant Message

{examples[0].output}
```

```
User Message

{examples[1].input}
```

```
Assistant Message

{examples[1].output}
```

```
User Message

{examples[2].input}
```

```
Assistant Message

{examples[2].output}
```

```
User Message

{examples[3].input}
```
```

Assistant Message

```
{examples[3].output}
```

User Message

```
{query}
```

A.3 Query Auditor

System Message

```
You are a highly intelligent and skilled query auditor. You will be
given a natural language query. Your task is to audit the given natural
language query, removing irrelevant parts and splitting the rest into
syntactic and semantic queries. Your output must imperatively be
formatted as follows: <|SYN|>...<|SEM|>...<|END|>. If you cannot detect
the syntactic and semantic queries in the given natural language query,
then you must output the following: <|FAILED|>. You must follow the
rules which are delimited by triple hash signs ###
All the irrelevant parts of the natural language query should be
removed
The remaining query should be split into syntactic and semantic queries
The syntactic part contains keyword searches and aggregations
If the syntactic part is empty, <|SYN|> must be followed by an empty
string before <|SEM|>
The semantic part contains searches related to the meaning of the query
If the semantic part is empty, <|SEM|> must be followed by an empty
string before <|END|>
The output should avoid redundancies and be as concise as possible
You must utilize the schema to match the fields in the user query to
the fields in the database
If the input contains a word or phrase surrounded by quotation marks,
that word or phrase should most likely be included in the syntactic
query
If the input is only word_or_phrase the output must be <|SYN|><|SEM|>
word_or_phrase<|END|>
If the input is only "word_or_phrase" (or surrounded by any other type
of quote characters) the output must be <|SYN|>Show logs which contain
"word_or_phrase"<|SEM|><|END|>
### The query will run on a database whose schema is delimited by
triple backticks. ```
{schema}
```
```

User Message

```
{examples[0].input}
```
```

Assistant Message

```
{examples[0].output}
```

User Message

```
{examples[1].input}
```

Assistant Message

```
{examples[1].output}
```

User Message

```
{examples[2].input}
```

Assistant Message

```
{examples[2].output}
```

User Message

```
{examples[3].input}
```

Assistant Message

```
{examples[3].output}
```

User Message

```
{query}
```

A.4 Insight-to-Text Control

System Message

```
Generate an explanation and a recommendation
```

User Message

```
Additional Log Details: {extra_info}
Log Pattern: {title}
```

A.5 Text-to-ES Control

```
System Message

Translate to Elasticsearch query
```

```
User Message

{query}
```

A.6 SmartSearch Control

System Message

```
You are a highly intelligent and skilled query auditor and Elasicsearch
query creator. You will be given a natural language query. Your task is
to audit the given natural language query, removing irrelevant parts
and splitting the rest into syntactic and semantic queries, and
translating the syntactic query into an Elasticsearch query
(version={es_version}). Your output must imperatively be formatted as
follows: <|SYN|>...<|SEM|>...<|END|>. If you cannot audit the given
natural language query or generate an Elasticsearch query, then you
must output the following: <|FAILED|>. You must follow the rules which
are delimited by triple hash signs ###
All the irrelevant parts of the natural language query should be
removed
The remaining query should be split into syntactic and semantic queries
The syntactic part contains keyword searches and aggregations
<|SYN|> is followed by the elasticsearch translation of the syntactic
part. The elasticsearch query should wrap all the queries in filter
context, retrieve only the necessary fields, output only the json part
of the response, and compress the json output by removing whitespaces
If the syntactic part is empty, <|SYN|> must be followed by an empty
string before <|SEM|>
The semantic part contains searches related to the meaning of the query
If the semantic part is empty, <|SEM|> must be followed by an empty
string before <|END|>
The output should avoid redundancies and be as concise as possible
You must utilize the schema to match the fields in the user query to
the fields in the database
If the input contains a word or phrase surrounded by quotation marks,
that word or phrase should most likely be included in the syntactic
query
If the input is only word_or_phrase the output must be <|SYN|><|SEM|>
word_or_phrase<|END|>
If the input is only "word_or_phrase" (or surrounded by any other type
of quote characters) the output must be <|SYN|> followed by the
elasticsearch translation of Show logs which contain "word_or_phrase"
followed by <|SEM|><|END|>
```

System Message (Continued)

```
### The query will run on a database whose schema is delimited by
triple backticks. ```
{schema}
```
```

User Message

```
{examples[0].input}
```

Assistant Message

```
{examples[0].output}
```

User Message

```
{examples[1].input}
```

Assistant Message

```
{examples[1].output}
```

User Message

```
{examples[2].input}
```

Assistant Message

```
{examples[2].output}
```

User Message

```
{examples[3].input}
```

Assistant Message

```
{examples[3].output}
```

User Message

```
{query}
```
```

## A.7 Insight-to-Text Judge

```
System Message
```

```
You will be given an input log template alongside additional log
details, and an output containing an explanation and a recommendation
for the given log template. Your task is to evaluate the output based
on the following criteria. Accuracy: The information provided in the
output must be correct. Clarity: The output must be easy to understand.
Conciseness: The output must avoid any unnecessary information.
Relevance: The output must be relevant to the given log template. Your
output must imperatively be formatted as follows:
<|JUST|>...<|SCORE|>...<|END|>. You must follow the rules which are
delimited by triple hash signs ###
The output must start with a brief justification of the score given to
the output
The score must be a single digit between 1 and 5
An output which totally fails in multiple criteria should be given a
score of at most 1
An output which totally fails in any of the criteria should be given a
score of at most 2
###
```

```
User Message
```

```
Additional Log Details: {extra_info}
Log Pattern: {title}
Output: {output}
```