

Master's Thesis

Submitted to the High Performance Computing Research Group
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

LLM in CubeGUI for HPC Performance Analysis

AASHIKA M SURESH
4011568
Master's in Computer Science

Supervisors: Prof. Dr. Christian Plessl
Prof. Dr. Marco Platzner
Date: July 3, 2026

Contents

Abstract **iii**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Score-P . . . 3
1.3 CubeGUI . . . 4
1.3.1 Call Paths and Call Trees . . . 5
1.4 Artificial Intelligence (AI) vs Machine Learning (ML) vs Generative AI (GenAI) . . . 9
1.5 Research Gap . . . 11
1.6 Thesis Objective . . . 11
1.7 Contributions . . . 12
1.8 Thesis Structure . . . 13

2 Related Work and Fundamentals **14**
2.1 Performance Analysis and Performance Problems . . . 14
2.1.1 HPC fundamentals . . . 14
2.1.2 Performance Problems . . . 14
2.1.3 Why Load Imbalance? . . . 15
2.2 Load Imbalance . . . 16
2.3 LLM Construction . . . 17
2.3.1 Transformer architecture . . . 17
2.3.2 Pretraining vs instruction tuning . . . 18
2.3.3 LLM Limitations and Challenges . . . 19

3 Using LLM for Performance Analysis **22**
3.1 Why LLMs Can Help . . . 22
3.2 Framing Performance Analysis as Language . . . 23
3.2.1 Converting metrics into structured prompts . . . 23
3.2.2 Representing imbalance as a performance problem . . . 23
3.3 Design Principles for the LLM response . . . 23

3.4 Risks . . . 25

4 Interaction with LLM **27**
4.1 Data Extraction and Processing . . . 27
4.2 Prompt Engineering . . . 29
4.3 Context Handling and Session Drifts . . . 33
4.3.1 Alternative approach: Agent-Based Integration . . . 34
4.4 API Communication . . . 35
4.4.1 Overview of Communication Model . . . 35
4.4.2 Request Construction . . . 36
4.4.3 Transmission and Sequential Execution . . . 36
4.4.4 Response Handling . . . 36
4.4.5 Asynchronous Execution Model . . . 36
4.4.6 Error Handling and Robustness . . . 37

5 CubeGUI Integration of AI Driven Analysis **38**
5.1 Plugin Architecture . . . 38
5.2 UI Integration . . . 39
5.3 Why an externally hosted LLM . . . 40

6 Evaluation **42**
6.1 Evaluation Goals . . . 42
6.2 Datasets . . . 42
6.3 Evaluation Metrics . . . 43
6.4 Experimental Setup . . . 44
6.5 Results . . . 45
6.6 Discussion . . . 47
6.6.1 Strengths . . . 47
6.6.2 Limitation . . . 47

7 Conclusion and Future Work **49**
7.1 Summary . . . 49
7.2 Future Work . . . 50

Bibliography **52**

Abstract

High-performance computing (HPC) applications often suffer from performance issues such as communication overhead, synchronization delays, load imbalance, serialization, etc., which are traditionally identified through manual and expert-driven analysis of profiling data. This work investigates the utility of large language models (LLMs) for analyzing performance problems in parallel programs with load imbalance analysis as a representative use case.

To explore the use of Large Language Models (LLM) in parallel applications, structured performance data is extracted from CubeGUI (a GUI that displays performance data for parallel programs including MPI and OpenOpenMP applications). CubeGUI is used to extract profiling data, including call trees and system trees. Call trees represent the hierarchical structure of function calls during program execution and system trees capture the underlying hardware and process hierarchy on which the program executes. In this study, execution time is used as the primary metric for analysis. A prototype AI plugin is developed using the Blablador API within CubeGUI, enabling the iterative processing of large-scale call tree data.

The key focus of the LLM's response is imbalance detection, prompt engineering and context design to improve model reliability, reduce hallucinations and ensure consistent interpretation of HPC-specific terminology. The plugin calculates the most relevant regions in the call tree for imbalance detection and the LLM detects the imbalance, calculates imbalance percentage, generates plausible explanations for imbalance and recommends methods for validating its conclusions using the call tree data provided.

Experimental evaluation on multiple trace profiles demonstrates that LLMs can achieve good accuracy in identifying load imbalance and providing meaningful diagnostic insights when supplied with sufficient structured context.

Overall, this thesis highlights the potential of LLMs as supportive tools for analyzing HPC performance problems, reducing manual effort and offering a novel and automated approach to diagnosing complex performance issues in parallel applications.

1. Introduction

This chapter introduces the motivation and background for the use of Large Language Models (LLMs) in High-Performance Computing (HPC) performance analysis. We discuss the importance of performance analysis in large-scale parallel applications and the challenges associated with interpreting profiling data. The chapter then presents the problem statement, introduces the Score-P and CubeGUI tools used throughout this work and relevant concepts from artificial intelligence, machine learning and generative AI are briefly reviewed. Finally, the existing research gap, the objectives, scope and main contributions of the thesis are outlined.

HPC is the use of supercomputers and clustered computing systems to solve complex, large-scale problems that are beyond the capability of standard computers. Some of such use cases of HPC systems are climate modeling, physics and computational chemistry. These systems achieve high computational performance through massive parallelism, distributing workloads across thousands of compute nodes that operate concurrently on different parts of a problem. Despite this computational capability, HPC applications remain fundamentally complex to design, optimize and analyze due to their distributed nature and sensitivity to execution inefficiencies [1].

Performance analysis plays a critical role in understanding and optimizing the behavior of HPC applications. As applications are executed across large number of processes, even minor inefficiencies in computation or communication can amplify across the system [2]. This makes it essential to analyze not only the execution time but also the behavior across individual processes. Without systematic performance analysis, it becomes difficult to identify bottlenecks that emerge under parallel execution.

The cost of inefficiencies in supercomputing environments is significant. Poorly optimized applications can lead to substantial under-utilization of expensive computational resources, including idle compute nodes, increased execution time and inefficient use of energy. Given that supercomputing centers operate under strict resource allocation and energy constraints, even small performance improvements can translate into considerable cost savings and improved throughput for multiple concurrent users [2]. As a result, identifying and eliminating performance inefficiencies is not only a technical requirement but also an economical necessity.

Artificial intelligence (AI) and in particular LLMs, have recently emerged as powerful tools for pattern recognition, reasoning and natural language-based analysis across various domains. LLMs are capable of processing structured and unstructured data and generating human-like interpretations, making them increasingly relevant for domains that require analytical reasoning over complex datasets. Their ability to synthesize information and provide explanations makes them promising for assisting in technical domains such as software and performance analysis.

Motivated by these capabilities, there is growing interest in exploring the use of AI and LLMs for HPC performance analysis. Traditional performance analysis tools provide detailed visualizations of execution behaviour but rely heavily on expert interpretation to derive meaningful insights. LLMs offer the potential to bridge this gap by assisting in the interpretation of profiling data, identifying performance anomalies and generating plausible explanations for observed behavior. In this context, they may serve as supportive tools that reduce manual effort and enhance the efficiency of performance debugging workflows in large-scale parallel applications.

1.1 Problem Statement

Despite advances in HPC profiling and measurement technologies, performance analysis remains a challenging and expert-driven task. Modern parallel applications generate large volumes of performance data across thousands of processes, making it difficult to systematically identify the causes of performance problems. Problems such as communication overhead, load imbalance and synchronization delays often emerge only at scale and require reasoning across multiple levels of execution behavior.

Although profiling (see Table 1.1 for difference between traces and profiles) tools can collect and visualize detailed runtime information, interpreting this information remains largely a manual process. Analysts must correlate execution-time metrics across processes, inspect hierarchical call structures and formulate hypotheses about potential causes of observed inefficiencies. This process is time-consuming and requires significant expertise in both parallel programming and performance analysis.

The challenge is therefore the ability to transform performance data into meaningful explanations. While profiling tools can partly indicate where performance problems occur, understanding why they occur often requires substantial manual

reasoning. Consequently, performance diagnosis remains one of the most labor-intensive stages of HPC application optimization.

This work investigates whether LLMs can assist in the detection and interpretation of load imbalance. Specifically, it explores the use of LLMs to analyze structured profiling data and generate explanations for observed performance behavior. Load imbalance is selected as a representative use case because it is both a common source of scalability loss and a performance issue that can be identified through execution-time distributions across processes.

To evaluate this approach, a prototype plugin is integrated into CubeGUI [3] and applied to profiling data and process-level performance metrics. Since HPC performance profiles can become significantly larger than the context window of modern LLMs, the data is processed iteratively and transformed into a structured representation suitable for language-model reasoning.

A further challenge lies in designing prompts that are sufficiently informative and concise. Excessively large or unstructured prompts can increase token consumption, reduce response quality and increase the likelihood of hallucinations. Therefore, an important aspect of this work is the development of prompts that enable reliable analysis of HPC performance data.

Feature	**Traces**	**Profiles**
Purpose	Record a timeline of events as they happen.	Summarize where execution time is spent.
Output	Chronological event log.	Statistical performance data.
Typical use	Debug task switches, interrupts, state changes and RTOS behavior.	Find CPU hotspots and optimize code.
Data volume	High.	Lower.
View style	Timeline or event sequence.	Percentages, call counts, and execution-time statistics.

Table 1.1: Comparison of traces and profiling techniques used in embedded software analysis.

1.2 Score-P

Performance analysis of parallel applications is based on collecting and examining execution data in a structured way. In the HPC ecosystem, this is typically done using tools that capture runtime behavior through instrumentation, tracing and profiling. One commonly used tool for this purpose is Score-P [4], which

provides a unified framework for measuring the performance of parallel applications.

Score-P instruments an application to record runtime information such as execution times. This instrumentation can be applied either manually or automatically depending on the level of control required. During execution, Score-P can generate traces and profiles. Tracing captures a detailed sequence of events over time, preserving the exact order in which operations occur. Profiling, on the other hand, aggregates performance data such as execution time over defined regions of the program.

In this work, we use profiles which includes metrics like execution time, number of visits to specific regions and communication related information, all of which form the basis for further performance analysis.

1.3 CubeGUI

The content of this specific section was derived from the explanation generated by claude ai.

CubeGUI is a visualization tool built on top of the Score-P measurement infrastructure [3] designed to support the exploration of profiling data.

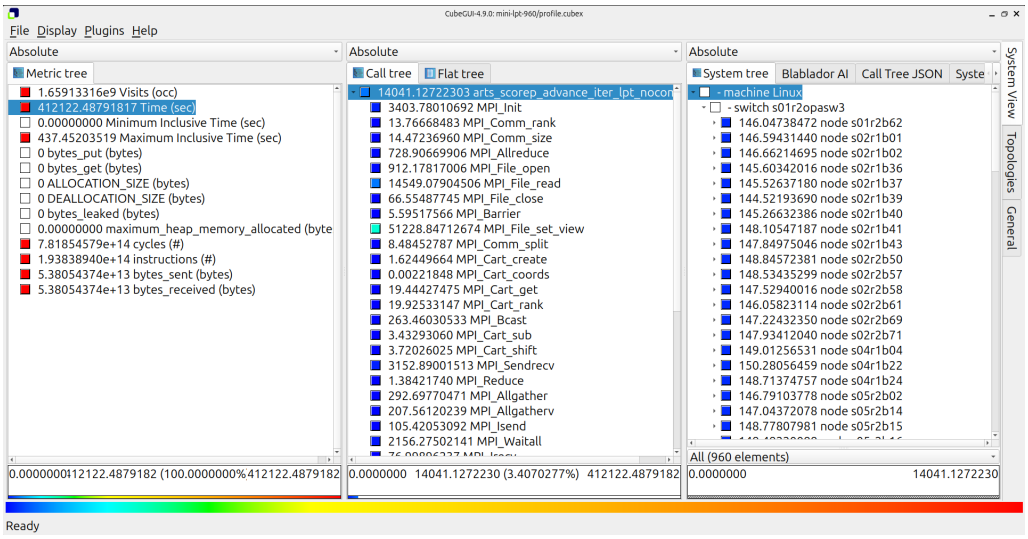

Figure 1.1: Real-time example

CubeGUI presents profiling information through three primary components (see Figure 1.1): a metric-selection view, a hierarchical call tree and a system tree representing the execution resources. By combining these views, users can

examine both the distribution of performance metrics within the application and their variation across processes and hardware resources.

1.3.1 Call Paths and Call Trees

Performance metrics collected by Score-P, such as execution time, are mapped onto both a call tree and a system tree. These complementary views allow users to analyze where application time is spent and how performance behavior varies across MPI ranks and compute resources.

Since CubeGUI stores and presents profiling information using calling-context information, it is important to distinguish between call paths and call trees. These concepts determine how performance metrics are associated with specific execution contexts within an application. To illustrate them, we consider the simple C program below:

```
#include <stdio.h>

void log_result(double value) {
    printf("Result: %f\n", value);
}

double square(double x) {
    return x*x;
}

double hypotenuse(double a, double b) {
    double a2 = square(a); // call 1 to  square()
    double b2 = square(b); // call 2 to  square()
    return square(a2 + b2); // call 3 to  square() (intentional bug: sh
}

void compute_and_print(double a, double b) {
    double result = hypotenuse(a,b);
    log_result(result);
}

int main() {
    compute_and_print(3.0,4.0); // first pair
    compute_and_print(5.0,12.0); // second pair
```

```
    log_result(42.0); // direct call - different call path
    return 0;
}
```

Listing 1.1: Example hypotenuse program used to illustrate call paths and call trees

The program computes hypotenuse and prints the results. Several functions are called repeatedly and some functions are reached through different execution routes.

Call Paths

A call path represents a single root-to-leaf sequence of function calls that occurs during execution. Every distinct execution route corresponds to a separate call path, even if the same functions appear multiple times.

Examples of call paths generated by the program 1.1 are:

```
call_tree_example
 -> main
    -> compute_and_print
       -> hypotenuse
          -> square

call_tree_example
 -> main
    -> compute_and_print
       -> log_result

call_tree_example
 -> main
    -> log_result
```

Thus, the same function may belong to multiple call paths depending on the calling context in which it is reached.

Call Tree

A call tree is a compact representation obtained by merging structurally identical call paths into a single hierarchy. Rather than storing every invocation separately,

repeated calls are represented by aggregated metrics such as execution time and invocation count.

The corresponding call tree for the example program 1.1 is shown in figure 1.2. The same function may belong to multiple call paths. For example, the function `log_result()` appears in two different call paths: one where it is invoked from `compute_and_print()` and another where it is called directly from `main()`. Although both nodes represent the same function, they correspond to different calling contexts.

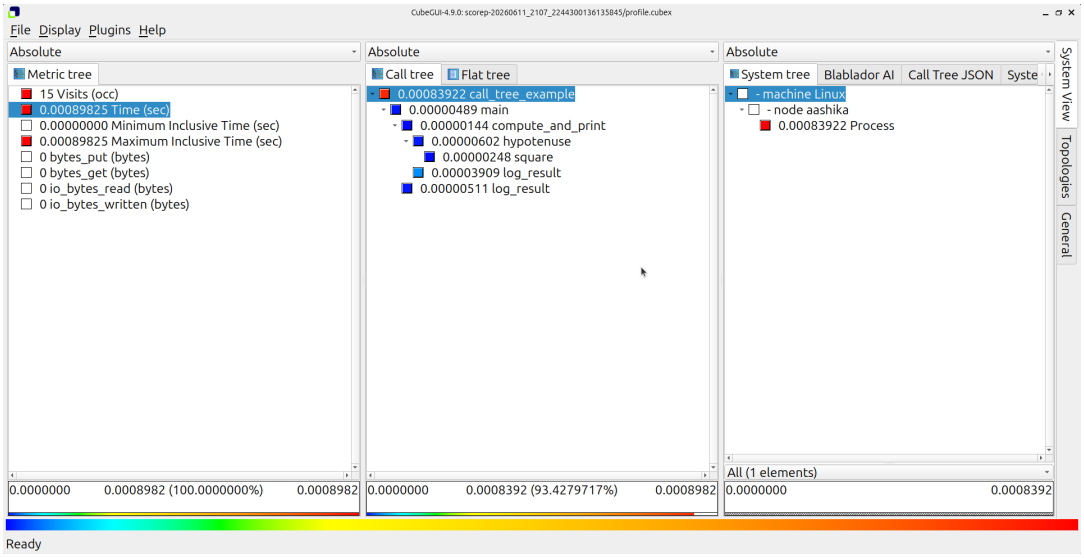

Figure 1.2: Call tree representation of the example program in CubeGUI.

Property	**Call Path**	**Call Tree**
Definition	One concrete root-to-leaf chain	All paths merge into one tree
Granularity	Maximum - every instance	Aggregated - grouped by structure
Repeated calls	Each repetition is a separate path	Expressed as invocation count
Calling context	Fully preserved	Preserved (unlike flat profiles)
Typical use	Detailed trace analysis	Interactive profile browsers (eg. Cubegui)

Table 1.2: Call paths vs. call trees.

CubeGUI internally represents profiling information using a call-tree structure because it preserves calling context while remaining compact enough for interactive analysis. Understanding this representation is therefore essential for interpreting the performance data discussed throughout the remainder of this thesis.

Connection to CubeGUI

The call tree captures the calling context of the application and allows users to identify where execution time and other performance metrics are accumulated. The system tree represents the hardware and process hierarchy, including compute nodes, processes and threads. Together, these views provide a structured representation of both the application behavior and the underlying execution environment.

The central panel of CubeGUI displays the call tree, while the left panel provides access to the available performance metrics. Metrics such as execution time, visit counts, communication volume and I/O activity can be selected independently and visualized within the same call-tree structure. For parallel applications, CubeGUI additionally supports different aggregation operators across MPI processes, including sum, mean, minimum and maximum values.

Inclusive and Exclusive Metrics

Performance metrics displayed in CubeGUI can be interpreted either inclusively or exclusively. Understanding this distinction is important when analyzing call trees because the same node may contribute to the overall runtime in different ways.

An inclusive metric represents the total cost associated with a call-tree node, including the time spent within the function itself as well as the time spent in all descendant functions. In contrast, an exclusive metric only accounts for the cost incurred directly within the function body and excludes contributions from child nodes.

For example, consider a function `A()` that calls a function `B()`. If `A()` executes for 10 seconds in total and 7 of those seconds are spent inside `B()`, then the inclusive time of `A()` is 10 seconds, while its exclusive time is 3 seconds. The inclusive and exclusive times of `B()` are both 7 seconds if `B()` does not call any additional functions.

When examining parallel profiling data, the interpretation becomes more complex because CubeGUI can aggregate metrics across multiple MPI processes. Depending on the selected aggregation operator, a child node may display a larger numerical value than its parent 1.3. This does not indicate an inconsistency in the call tree. Instead, it results from the aggregation of measurements collected from different processes. For example, a communication routine executed by many MPI ranks may accumulate a larger summed execution time than the parent call path in which it appears. Consequently, aggregated values should be interpreted

in the context of both the selected metric and the aggregation strategy.

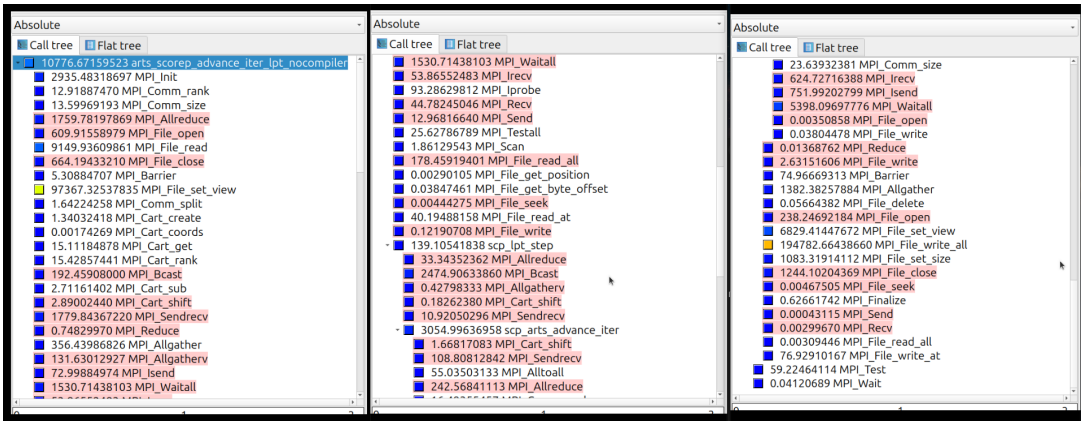

Figure 1.3: The entire call tree where some child values are larger than the parent

The profiling data represented in CubeGUI forms the basis for the analysis performed in this thesis. In particular, the hierarchical call-tree structure and system tree are used as input for the proposed LLM-based reasoning approach.

One possible solution would be to modify CubeGUI itself by introducing alternative color schemes, imbalance-specific heat maps or globally normalized color scales. While such changes could improve the visibility of load imbalance, they would not fundamentally solve the performance analysis problem. Visualization techniques can highlight where differences occur, but they cannot explain how problematic those values are, why those differences exist, determine whether they are significant, identify the affected ranks or suggest how the observed behavior should be investigated.

Rather than replacing CubeGUI's visualization mechanisms, this work complements them with automated detection and reasoning capabilities. These limitations highlight the gap between performance data visualization and performance interpretation, motivating the need for approaches that can assist in automated analysis over structured profiling data.

1.4 Artificial Intelligence (AI) vs Machine Learning (ML) vs Generative AI (GenAI)

AI is a broad field and has had many definitions and explanations over the last few decades. To put it simply, AI refers to the use of technologies to build machines and computers that have the ability to mimic cognitive functions associ-

ated with human intelligence, such as being able to see, understand and respond to spoken or written language, analyze data, make recommendations and more.

ML is a subset of AI that automatically enables a machine to learn and improve from experience. Instead of explicit programming, ML uses algorithms to analyze large amounts of data, learn from the insights and then make informed decisions. ML algorithms improve performance over time as they are trained by being exposed to more data. ML models are the output, or what the program learns from running an algorithm on training data. The more data used, the better the model will get [5], [6].

GenAI is the technology behind LLMs (llama, chatgpt claude, midjourney, etc). GenAI technology is a newer use of AI and it goes beyond analysis of new data. It can generate natural language responses to user prompts, for example, generate images or videos based on user requests. A GenAI model is trained using massive sets of sample data that teach the model how a language such as English works and what information is stored in the data provided. AI training is extremely compute-intensive, requiring many hours and many graphics processing units (GPUs) to fully train the model. Generative AI models typically don't continue to learn after their initial training. Instead, they use their understanding of language and the information they've been trained on to either answer queries or analyze new data and provide answers about it in conversational language. Trained models can be fine-tuned to perform well in specific domains. For instance, an LLM specialized in HPC and performance analysis can provide more relevant support to developers and researchers working with parallel applications. However, a model typically has no inherent knowledge of organization-specific information. To overcome this limitation, Retrieval-Augmented Generation (RAG) can be used to provide the model with relevant external documents or knowledge bases during inference, allowing it to generate responses based on information beyond its original training data. So, if you want to know what new features exist the newer release of CubeGUI compared to the previous version, you can provide the repository or the new user-guide via RAG and the LLM can generate the answer.

Therefore, AI is the overarching field that aims to create systems capable of performing tasks that typically require human intelligence. ML is a subset of AI that enables systems to learn from data and improve their performance over time without being explicitly programmed. GenAI is a specialized branch of ML that focuses on creating new content, such as text, images or code, by learning patterns from large datasets.

1.5 Research Gap

Recent advancements in AI and ML have led to an increasing interest in their application within HPC. Several studies have explored the use of AI-driven techniques for performance modeling, optimization and workflow management. In particular, LLMs have shown promise in tasks such as code generation [7], [8], automatic parameter tuning and supporting HPC workflows by interacting with tools and coordinating their execution.

However, existing work in this area primarily focuses on improving how applications are written, configured or executed [8], not how their performance is analyzed after execution or point to where the problem lies. Approaches such as auto-tuning aim to optimize performance parameters, while code generation techniques assist in producing efficient implementations. Similarly, workflow automation leverages AI to manage and coordinate complex computational pipelines. While these contributions are valuable, they do not address the challenge of interpreting performance data generated by profiling tools.

In contrast, performance analysis in HPC remains reliant only on traditional tools that provide detailed visualizations but require manual and expert-driven interpretation. Despite the availability of structured profiling data like hierarchical call trees, there is limited work on applying LLMs directly to this data for the purpose of reasoning about performance behavior. In particular, the integration of LLMs into existing performance analysis tools, where they can assist users in identifying bottlenecks, explaining observed behaviour and suggesting validation strategies remains largely unexplored.

This gap (see Table 1.3 for summary) highlights an opportunity to investigate the use of LLMs not only as generative or optimization tools, but as interpretative systems that can operate on structured performance data. Building upon this, this work explores how LLMs can be used to assist in performance analysis tasks, with a focus on identifying load imbalance, reasoning about its potential causes and suggesting methods to validate these reasons within the context of existing profiling workflows.

1.6 Thesis Objective

1. Investigate whether LLMs can interpret structured performance data and assist in identifying performance issues, with load imbalance as a representative use case.

Existing AI/ML Applications in HPC	This Thesis
Focus on improving code generation, compilation and optimization of HPC applications.	Focus on post-execution performance analysis and interpretation of profiling data by pointing which ranks are effecting the execution time.
Emphasis on auto-tuning and parameter optimization for performance improvement.	Emphasis on understanding performance behavior and identifying inefficiencies.
Limited support for interpreting profiling results such as call trees and system-level metrics.	Direct analysis of structured profiling data including call trees and system trees.
No systematic mechanism to validate or refute performance hypotheses.	Suggests validation strategies for hypothesized performance issues.

Table 1.3: Summary of the research gap between existing AI applications in HPC and the proposed LLM-based performance analysis problems approach

2. Evaluate and automate the ability of LLMs to detect imbalance and also provide plausible explanations and suggest methods to validate or refute these hypotheses.

3. Develop effective and concise prompt designs to ensure reliable and consistent LLM responses while analyzing performance data.

4. Integrate an LLM-based plugin into CubeGUI to enable interaction with structured profiling data while addressing context window limitations through iterative data processing.

5. Evaluate the effectiveness and accuracy of the proposed approach using both benchmark and real-world performance profiles.

1.7 Contributions

This thesis makes the following contributions:

1. **Design Structured Prompt for HPC Performance Analysis:** A dedicated prompt engineering approach is developed to enable LLMs to effectively interpret structured HPC performance data. The scheme supports the identification of load imbalance and facilitates reasoning about potential causes, while also guiding the model on how to generate responses.

2. **Persistent Context Handling Framework for Performance profiles:** A framework is introduced to handle large HPC performance profiles in an iterative

manner, addressing the limitations of LLM context windows. This approach ensures that hierarchical relationships within call trees are preserved across multiple queries, enabling coherent analysis of profiling data.

3. **Prototype LLM Integration into CubeGUI:** A prototype AI plugin is implemented within CubeGUI. This integration allows LLM-based reasoning to be applied directly within an established HPC performance analysis workflow.

4. **Evaluation Methodology for LLM-Based Performance Analysis:** Various benchmark and real world profiles are tested and the LLM results are validated against manual expert analysis.

The overall architecture of the proposed approach is explained in Chapter 4.

1.8 Thesis Structure

This thesis is organized as follows.

Chapter 2 provides the theoretical foundations required for this work introducing the fundamentals of HPC performance analysis, common performance problems with a particular focus on load imbalance and it presents the underlying concepts of LLMs including its limitations.

Chapter 3 continues to further explain LLMs with the rationale for applying LLMs to HPC performance analysis. It discusses how performance-analysis problems can be represented as structured language tasks, presents the design principles used to guide model behavior and outlines the risks and limitations associated with LLM-based reasoning.

Chapter 4 describes the interaction framework between CubeGUI and the LLM.

Chapter 5 presents the implementation of the proposed approach within CubeGUI. It discusses the plugin architecture, user-interface integration and the design decisions behind using an externally hosted LLM service.

Chapter 6 evaluates the proposed system using benchmark and real-world HPC profiling datasets. The evaluation methodology, datasets, metrics, experimental setup, results and limitations of the approach are discussed.

Lastly, Chapter 7 summarizes the main findings of the thesis, reviews the contributions and outlines directions for future work.

2. Related Work and Fundamentals

This chapter provides the theoretical background for this thesis. It first presents the fundamentals of HPC performance analysis and discusses common performance problems encountered in parallel applications. Particularly highlighting load imbalance, which is selected as the main performance issue investigated in this work. The chapter then introduces the fundamentals of LLMs including their architecture, training methodology and limitations, thereby providing the necessary foundation for the next chapter on applying LLMs to load imbalance analysis.

2.1 Performance Analysis and Performance Problems

Performance analysis in HPC is the process of measuring, analyzing and understanding how efficiently a parallel application uses computing resources with the goal of identifying bottlenecks and improving performance. [9]

To put it simply, it answers questions like: where the program is spending time, why it isn't scaling well, are all cores doing equal work and where the delays are coming from. Small inefficiencies amplify across hundreds of cores and scalability depends on balanced workload and distribution making performance debugging essential for efficient resource usage.

2.1.1 HPC fundamentals

HPC applications rely on parallel execution models to achieve scalability, high performance, efficient resource utilization and reduced execution time.. Two commonly used paradigms are Message Passing Interface (MPI), which distributes work across multiple processes with explicit communication and OpenMP, which enables shared-memory parallelization through threads. While these models allow applications to scale across large systems, they also introduce complexities in execution behavior that make performance analysis non-trivial [10].

2.1.2 Performance Problems

In real-world HPC applications, ideal scalability is rarely achieved due to a variety of performance inefficiencies that emerge during parallel execution. These issues are often subtle and cannot be identified by looking at overall execution

time alone. Instead, they require analysis of how computation, communication and resource usage are distributed across processes.

Several common categories of performance problems can be observed in parallel applications [2]. Load imbalance occurs when work is unevenly distributed where some processes take more time than others leaving faster processes idle. Communication overhead arises when excessive or inefficient data exchange between processes dominates execution time. Synchronization delays occur when processes are forced to wait at coordination points such as barriers, leading to idle time and reduced parallel efficiency.

In addition to these, I/O bottlenecks can significantly impact performance when multiple processes access shared storage resources concurrently, resulting in contention and delays. Memory bandwidth limitations may also restrict performance, particularly in data-intensive applications where the rate of data movement becomes a limiting factor rather than computation itself.

These performance problems often interact with each other, making diagnosis more challenging. For example, what appears as a communication delay may in fact be caused by an underlying load imbalance, where some processes reach communication points later than others. As a result, effective performance analysis requires not only identifying where inefficiencies occur but also understanding their underlying causes.

2.1.3 Why Load Imbalance?

Among the various performance issues discussed, load imbalance is chosen as the primary focus of this work. It is one of the most fundamental challenges in parallel computing. Even in well-designed applications, small variations in workload distribution can lead to significant performance degradation when scaled across a large number of processes. As discussed in Section 5.3.9 of [11], scalability losses in parallel applications are often strongly influenced by imbalance effects, where delays in a subset of processes propagate across the entire execution and reduce overall efficiency.

Load imbalance is also particularly suitable for analysis using structured profiling data in CubeGUI. Imbalance can be observed through differences in execution time across processes and across hierarchical call paths, making it accessible through high-level performance metrics without requiring detailed hardware-level measurements. This makes it easier to evaluate whether LLMs can interpret performance data effectively.

At the same time, identifying the presence of imbalance is only part of the

problem. Understanding its root causes requires reasoning across multiple levels of the application, including function-level behavior, communication patterns and execution flow. This combination of observable patterns and underlying complexity makes load imbalance an appropriate and representative use case for exploring the capabilities of LLM-assisted performance analysis.

2.2 Load Imbalance

Building on the previous section, formally, load imbalance refers to a condition in parallel applications where the computational workload is unevenly distributed across processes, resulting in variations in execution time. In an ideal scenario, all processes should complete their assigned work simultaneously. However, when some processes take longer than others, faster processes remain idle while waiting, leading to inefficient utilization of computational resources and degraded overall performance.

Load imbalance can be categorized into static and dynamic imbalance. Static imbalance arises from an uneven distribution of work at the start of execution, often due to poor data partitioning or algorithm design whereas dynamic imbalance develops during runtime as a result of varying execution paths, data-dependent behavior or changing workload conditions across processes. While static imbalance can often be anticipated and mitigated during design, dynamic imbalance is more difficult to detect and typically requires detailed runtime analysis. Dynamic imbalance is often difficult to predict because workload variations may emerge during execution as a consequence of communication behavior, data dependencies or irregular computational patterns [11].

Several factors contribute to load imbalance in HPC applications. Uneven data distribution is a common cause, where input data is not partitioned equally across processes, leading to disproportionate workloads. Branch divergence can also introduce imbalance when different processes follow different execution paths based on conditional logic, resulting in varying execution times. Communication-related waiting is another factor, where delays in data exchange cause some processes to stall while others continue execution. Such waiting effects are particularly important in large-scale parallel systems, where delays introduced by a subset of slower processes can propagate across synchronization operations and affect the execution efficiency of the entire application. Additionally, task granularity plays an important role, tasks that are too coarse may not distribute evenly, while overly fine-grained tasks can introduce overheads that offset the benefits

of parallelism.

To quantify load imbalance, performance analysis typically relies on execution time-based metrics. A simple and commonly used approach is to compare the maximum execution time across processes with the mean execution time, where larger deviations indicate greater imbalance.

A typical example of load imbalance can be observed in programs using MPI barriers (see Figure 1.1 for reference). This behavior often becomes more pronounced at scale, where even small differences in execution time can accumulate across many processes.

As explained in the previous chapter, load imbalance is reflected through variations in execution time across processes within the call tree. Higher-level call paths may appear balanced but deeper inspection often reveals discrepancies at the level of child function calls. The colour gradient in CubeGUI highlights regions with higher imbalance more prominently. By navigating the call tree hierarchy and examining execution time distributions across each rank in the system tree, users can identify where imbalance occurs and how it propagates through the application structure.

2.3 LLM Construction

LLMs are a class of generative AI models designed to process and generate human-like text. They are trained on massive collections of textual data and learn statistical relationships between words, phrases and contextual patterns. Rather than storing explicit rules, LLMs learn how language is structured and use this learned representation to predict the most likely continuation of a sequence of text. This capability allows them to perform tasks such as summarization, reasoning, explanation generation and question answering across a wide range of domains.

2.3.1 Transformer architecture

Modern LLMs are primarily based on the Transformer [12] architecture, which introduced a more efficient way of processing sequential data compared to earlier neural network approaches. Instead of processing text one word at a time, transformers process sequences in parallel, allowing them to capture long-range dependencies more effectively and scale to significantly larger models and datasets. This architecture forms the foundation of most current LLMs, including LLMs such as GPT, LLaMA and Claude. This article explains an in-depth understanding of the transformer model [13]

Before text can be processed by an LLM, it must first be converted into smaller units known as tokens. Tokens may represent complete words, subwords or punctuation depending on the tokenizer used by the model. For example, a long technical identifier or function name may be split into multiple tokens. The model operates internally on these token representations rather than raw text. Since every input and output consumes tokens, the number of tokens directly affects the amount of information the model can process at a time.

A key mechanism within Transformer models is the attention mechanism. Attention enables the model to determine which parts of the input are most relevant when generating a response. Instead of treating all input information equally, the model dynamically focuses on specific tokens based on contextual relevance. This capability is particularly important for tasks that involve reasoning over structured information, relationships between entities or hierarchical data.

2.3.2 Pretraining vs instruction tuning

The capabilities of LLMs are largely developed through two stages: pretraining and instruction tuning. During pretraining, the model learns general language patterns and contextual relationships from very large datasets. However, pretrained models are not necessarily optimized for following user instructions or producing reliable conversational responses. Instruction tuning is a later stage in which the model is further trained on carefully designed prompts and responses to improve its ability to follow tasks, answer questions and behave more consistently in interactive settings. Figure 2.1 is the general Transformer Architecture, detailing Encoder and Decoder stacks with their respective sub-layers and data flow and Figure 2.2 illustrates the basic LLM architecture for easier understanding.

Briefly, the Transformer architecture consists of two main components: an encoder and a decoder, each built from multiple stacked layers containing multi-head self-attention and feed-forward neural networks. The encoder processes the input sequence by learning contextual representations of each token, allowing the model to capture relationships between words regardless of their position in the text. The decoder generates the output sequence one token at a time by combining masked self-attention, which prevents access to future tokens during generation, with cross-attention that incorporates information from the encoder's representations. While the original Transformer architecture employs both encoder and decoder stacks for sequence-to-sequence tasks such as machine translation, most modern large language models (LLMs), including GPT-based mod-

els, primarily use a decoder-only architecture for autoregressive text generation. During pretraining, this decoder learns to predict the next token from large-scale text corpora, whereas instruction tuning further refines the pretrained model to better follow user instructions and produce more helpful, coherent, and contextually appropriate responses.

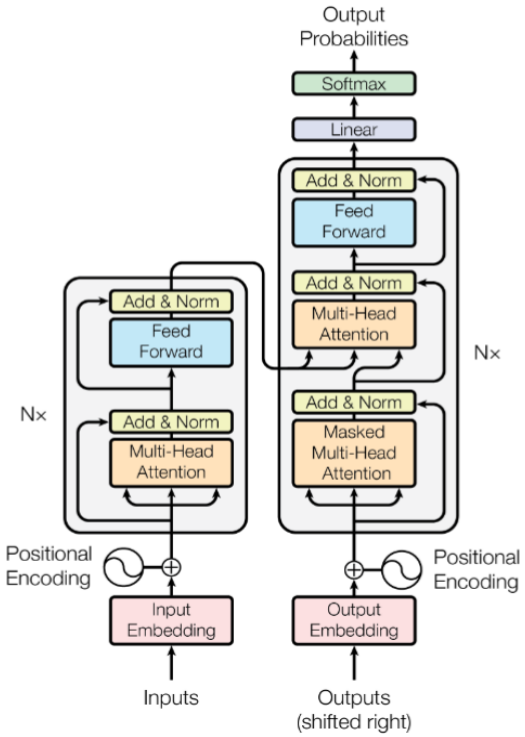

Figure 2.1: General Transformer Architecture [12]

2.3.3 LLM Limitations and Challenges

Although LLMs have many capabilities, they still lack in some areas/have some limitations, all of which are handled in this work. Some of the limitations are listed below.

1. **Context window:** Context window is the maximum amount of input text the model can process at once. If the provided input exceeds this limit, earlier information may be truncated or ignored [14]. This becomes particularly relevant in HPC performance analysis, where profiling data such as call trees and process-level metrics can become extremely large. As a result, large performance profiles often need to be divided into smaller segments and processed iteratively.

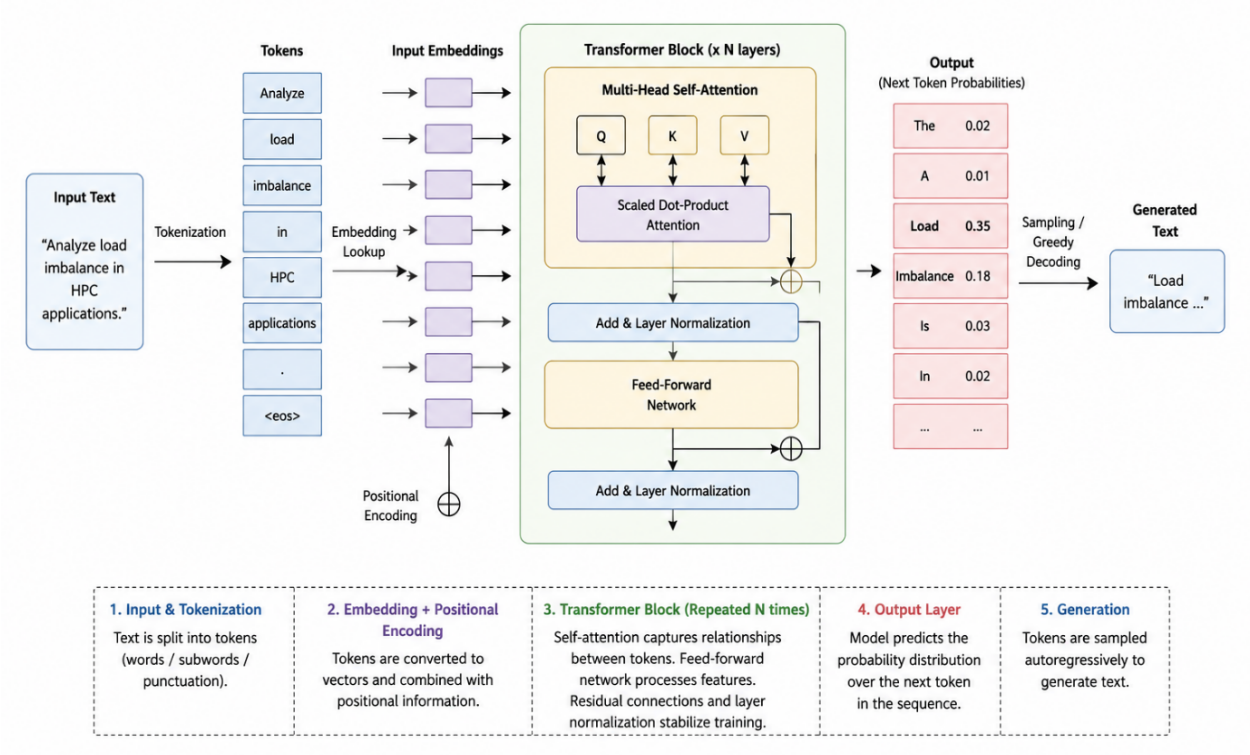

Figure 2.2: Basic LLM Architecture, generated with Claude AI

2. **Hallucination:** A hallucination [15] is when the model generates information that appears plausible but is incorrect or unsupported by the provided input. Hallucinations become more likely when prompts are ambiguous, excessively large or lacking sufficient context. Since this work relies on reasoning over structured HPC performance data, minimizing hallucinations is important to ensure that generated explanations and diagnostic suggestions remain meaningful and technically relevant.

3. **Prompt sensitivity:** LLMs are also highly sensitive to prompt design. Small changes in wording, structure or ordering of information can significantly affect the quality and consistency of the generated responses. Therefore, prompt engineering plays an important role in this thesis. The prompts must remain concise [16] enough to fit within context limitations while still providing sufficient structured information for the model to reason about performance behavior accurately.

   In addition, multi-turn interactions with LLMs introduce further challenges. As conversations become longer, models may gradually lose important contextual information or produce inconsistent reasoning across multiple queries. This issue becomes particularly relevant when analyzing large hierarchical

profiling datasets iteratively, where maintaining continuity between successive interactions is necessary for coherent analysis.

4. **Context drift and session volatility:** The model gradually shifts focus away from previously established information during extended interactions. Closely related to this is session volatility, where responses to similar prompts may vary across different sessions or interactions despite identical inputs.

3. Using LLM for Performance Analysis

This chapter discusses the conceptual basis for applying LLMs to HPC performance analysis. Here we explain why LLMs are potentially suitable for interpreting structured profiling data and assisting in reasoning about performance behavior. The discussion also introduces how performance analysis problems can be represented in a form that is understandable to language models and outlines the design principles used to guide reliable model responses.

3.1 Why LLMs Can Help

Although LLMs were originally developed for natural language processing tasks, their ability to reason over structured information and generate human-readable explanations makes them applicable beyond traditional text analysis. As explained in the previous chapter, HPC profiling data contains structured relationships between call paths, processes and execution metrics, making it feasible for LLM-assisted interpretation.

LLMs ability to reason about relationships helps performance analysis as it often requires correlating execution-time differences across MPI ranks, understanding hierarchical call path structures and identifying patterns that indicate performance inefficiencies. Rather than analyzing individual metrics in isolation, LLMs can process structured contextual information and generate explanations that resemble the reasoning process used by performance analysts.

LLMs are also able to translate low-level profiling information into natural language explanations. Traditional performance analysis tools primarily expose metrics and visualizations, leaving interpretation to the user. In contrast, an LLM can describe why a particular execution pattern may indicate load imbalance, suggest possible causes and propose methods for further investigation. This capability can reduce the effort required to interpret large profiling datasets and make performance analysis more accessible.

The goal of this work is not to replace existing performance analysis tools or expert knowledge. Instead, LLMs are investigated as supportive reasoning systems that complement traditional profiling workflows by assisting with the interpretation of structured performance data and generating hypotheses that can be validated by the user.

3.2 Framing Performance Analysis as Language

3.2.1 Converting metrics into structured prompts

HPC profiling data primarily consists of numerical measurements organized within hierarchical structures such as call trees and system trees. While these representations are well suited for human exploration through visualization tools, LLMs require information to be presented in a textual form. Consequently, profiling data must be transformed into a structured representation that preserves both the performance metrics and the relationships between them.

In this work, profiling information is represented as structured prompts containing execution metrics, MPI rank information and call-path context. Rather than exposing the model to graphical visualizations, relevant performance observations are converted into compact data. This allows the LLM to reason about performance behavior using the same underlying information available to the user while avoiding unnecessary details that would increase prompt size.

A call path represents a specific route through the call tree from the root of the application to a particular function. By associating runtime measurements with individual call paths, performance behavior can be analyzed at different levels of granularity. This is especially relevant for load imbalance analysis, as performance issues often originate in specific regions of the application rather than uniformly across the entire execution.

3.2.2 Representing imbalance as a performance problem

Performance anomalies are represented through metric values. For example, differences in execution time between MPI ranks, average and maximum runtimes and imbalance severity metrics provide a clearer indication of uneven workload distribution. Presenting profiling data together with rank-level runtime distributions enables the LLM to reason about relationships between processes and identify patterns that may indicate performance inefficiencies.

3.3 Design Principles for the LLM response

The effectiveness of LLM-assisted performance analysis depends strongly on how profiling information is presented to the model. Simply providing large amounts of raw performance data does not necessarily produce reliable or meaningful analysis. In many cases, excessively large or poorly structured data can lead to

inconsistent reasoning, loss of important context or hallucinated conclusions. For this reason, this work follows a set of design principles for the LLM intended to improve response consistency, maintain interpretability and guide the model toward evidence based reasoning over structured HPC profiling data.

During the early stages of research, several LLMs such as Gemini, GPT, LLama, Mistral-AI were evaluated for their suitability in HPC performance analysis tasks. Although multiple models were capable of identifying imbalance patterns, GPT-OSS 120B consistently produced the most reliable responses. In particular, it adhered more closely to the requested output format, generated concise explanations, avoided introducing unnecessary information and showed a lower tendency to infer unsupported conclusions from its general knowledge of HPC systems. These characteristics made it particularly well-suited for the structured and evidence-driven analysis required by this work. Therefore, **GPT-OSS 120B** was selected as the model for all subsequent experiments and evaluations.

The design principles that we settled on to produce the required and best results are as follows:

1. **Structured Prompt Design:** Important information such as call paths, metric values, process-level runtimes and analysis instructions are clearly separated within the prompt. This organization helps the model distinguish between contextual information, observed profiling data and the expected analysis task. Since HPC performance data is inherently hierarchical, preserving this structure within the prompt also improves the model's ability to reason about relationships between different execution regions.

2. **Concise Prompt Design ("Caveman Language"):** Rather than using long and highly descriptive prompts, this work employs concise and simplified instructions, sometimes referred to as "caveman language" [17]. Concise prompts reduce ambiguity, make important information easier for the model to identify and lower the likelihood of accidentally introducing contradictory instructions. They also help ensure that the primary analysis task remains prominent, whereas excessively large prompts may dilute the focus of the model with unnecessary contextual information.

   Additional explanatory text often provided little benefit and occasionally encouraged the model to infer unsupported conclusions. By restricting prompts to essential profiling information and clearly defined instructions, the model was encouraged to derive conclusions directly from the supplied data. This approach reduced hallucinations, reduced context size and im-

proved the consistency and technical reliability of the generated responses.

3. **Controlled Output Format:** The model is constrained to generate responses in a predefined format which is easily readable in the CubeGUI plugin. The prompts explicitly specify how imbalance severity, affected ranks, possible causes and validation suggestions should be reported. Restricting the response structure improves consistency across different analyses and simplifies both interpretation and evaluation of the generated results. It also reduces the likelihood of irrelevant or overly verbose explanations.

4. **Constrained Reasoning to Reduce Hallucination:** Unsupported assumptions about workload characteristics, communication behavior or hidden execution patterns are intentionally restricted. For example, imbalance is inferred strictly from runtime differences between ranks rather than speculative application behavior. The prompts also constrain the model to provide evidence-driven explanations and concrete validation suggestions grounded in the observed profiling data. This helps reduce hallucinated conclusions and improves the technical reliability of generated responses.

5. **Domain-Specific Guidance for HPC Analysis:** Since general-purpose LLMs do not possess an inherent understanding of HPC execution semantics, the prompts include domain-specific guidance regarding concepts such as imbalance severity, synchronization behavior and rank-level execution differences. Explicit rules are provided to define how imbalance should be interpreted and how affected ranks should be identified. Embedding such constraints into the prompt aligns the generated analysis closely with CubeGUI and HPC performance analysis practices.

Together, these design principles form the foundation of the proposed LLM-assisted performance analysis approach. They provide a structured framework for transforming hierarchical profiling data into a form that can be processed reliably by language models while minimizing ambiguity, reducing hallucination and maintaining consistency across large-scale iterative analyses.

3.4 Risks

The risk of incorrect load imbalance detection is low because it is calculated by the LLM with a predefined load imbalance formula used in CubeGUI environment. The identification of imbalanced call paths relies on metrics extracted di-

rectly from the profiling data. Throughout the evaluation, no incorrect imbalance classifications were observed for the tested datasets.

However, the explanatory components of the analysis are inherently less certain than the imbalance detection itself as it has no access to the source-code due to token limitations. Explanations, investigation suggestions and proposed fixes should be interpreted as hypotheses rather than verified root causes.

Each call path is analyzed independently to avoid context drift and reduce hallucinations. While this improves consistency, it prevents the model from building a global understanding of the application across the entire call tree. Incorporating source code, previous analyses or complete execution traces into the model context could potentially improve reasoning in some cases, but would also increase token consumption and may encourage unsupported inferences from unrelated information. For large HPC applications containing thousands of functions and extensive source code bases, such an approach is currently impractical due to context window limitations.

As language models continue to improve, future systems may be able to incorporate larger amounts of contextual information while maintaining reliable and evidence-based reasoning. Until then, restricting the analysis to structured profiling data remains an effective strategy for reducing hallucinations and maintaining consistent diagnostic behavior.

4. Interaction with LLM

This chapter describes how profiling information extracted from CubeGUI is transformed into a representation suitable for LLM-based reasoning. Since HPC profiling data is hierarchical, large in scale and often exceeds practical LLM context limits, an intermediate processing layer is required before analysis can be performed. The chapter therefore presents the data extraction process, the prompt design strategy, the context management approach and the rationale behind the chosen interaction model. These design decisions form the foundation for the CubeGUI integration presented in Chapter 5.

4.1 Data Extraction and Processing

The overall workflow begins with extracting profiling information from CubeGUI through the plugin interface. The plugin traverses each call tree and its corresponding system tree to associate execution metrics with individual MPI ranks for each call path. During traversal, runtime values are collected for every rank belonging to a specific call path, allowing process-level execution behavior to be analyzed independently.

Call Tree Traversal

The call tree traversal is performed recursively. Starting from the top-level call nodes, each child node is visited and analyzed separately. For every call path, the plugin iterates through all system tree leaf nodes representing MPI ranks. The corresponding runtime value for the currently selected call path and rank combination is then retrieved from CubeGUI. This produces a runtime distribution for each call path, which forms the basis for imbalance analysis.

Reducing Token Explosion

Rather than forwarding the complete profiling dataset to the LLM, the GUI calculates the call paths that contribute most to the runtime, then sends those call paths and their corresponding system trees to the LLM. For each call path, LLM computes imbalance severity percentage, slowest ranks and reasoning.

The **runtime contribution** of each call path is calculated by the GUI as:

$$C = \frac{T_{\text{collapsed}}}{T_{\text{total}}} \times 100 \tag{4.1}$$

where:

- C = runtime contribution of the call path (%)
- $T_{\text{collapsed}}$ = collapsed (inclusive) runtime of the call path
- T_{total} = total runtime of the application

The **imbalance severity** is computed by the LLM using the load imbalance metric adopted by the POP Cube environment. This metric measures the deviation of the slowest MPI rank from the average runtime across all ranks and is defined as:

$$I = \frac{T_{\text{max}} - T_{\text{avg}}}{T_{\text{max}}} \times 100 \tag{4.2}$$

where:

- I = imbalance severity (%)
- T_{max} = maximum runtime across all MPI ranks
- T_{avg} = average runtime across all MPI ranks

The metric provides a local view of load imbalance and is intended to characterize the runtime distribution among MPI ranks for an individual call path. The rank with the highest runtime is additionally identified as the worst-performing rank. Alongside these summary metrics, the complete per-rank runtime table is preserved to provide sufficient evidence for the model to reason about imbalance behavior.

The extracted information is then converted into an LLM-readable textual format. Only one call path is processed per request, which significantly reduces token usage while also constraining the reasoning scope of the model. Treating each call path independently prevents the model from mixing information between unrelated execution regions and improves response consistency.

Several strategies are used to reduce prompt size:

1. Only runtime-related metrics relevant to imbalance analysis are extracted.
2. Each API request processes exactly one call path at a time.
3. Most contributing call paths are computed locally bu the GUI instead of sending all the data to the LLM.

4. The most contributing call pahs along with their corresponding system trees is sent to the LLM.

5. Non-essential profiling metadata is excluded from prompts.

6. Recursive traversal is used selectively to focus only on meaningful execution regions.

By performing lightweight in the GUI before sending the data to the LLM, the model can focus primarily on interpretation and reasoning rather than being fed large amounts of data that will lead to token explosion.

Another important aspect of the pipeline is performance overhead. Since the plugin operates interactively inside CubeGUI, the extraction process must remain lightweight enough to avoid disrupting the user workflow. Sequential processing is therefore used to analyze call paths incrementally instead of issuing large batch requests simultaneously.

Overall, the data extraction and preprocessing pipeline acts as an intermediate layer between CubeGUI and the LLM. It transforms complex hierarchical profiling information into concise and structured prompts that remain computationally manageable while preserving sufficient execution detail for meaningful performance reasoning.

4.2 Prompt Engineering

The design principles for LLM responses outlined in Chapter 3 demonstrate that prompt engineering plays a central role in determining the quality and consistency of generated outputs. Since LLMs are highly sensitive to prompt structure and wording, carefully designed prompts are essential for producing reliable and reproducible analyses. In this work, the prompt was intentionally structured to constrain the model's behavior, minimize ambiguity and ensure that every analyzed call path produces a consistent response suitable for integration into CubeGUI.

The prompt is provided as a system prompt, giving it the highest priority during inference. It first establishes the model's role by explicitly stating: **You analyze EXACTLY ONE call path.**

This instruction narrows the scope of the task and prevents the model from attempting to reason about the entire application execution. Since the plugin sends one request per selected call path, restricting the model to a single analysis simplifies both the prompt and the interpretation of the generated response.

Next, the prompt defines the expected input variables:

1. **call-path:** the full hierarchical name of the analyzed region,
2. **collapsed-runtime:** the inclusive runtime of the call path,
3. **expanded-runtime:** the exclusive runtime of the call path and
4. **runtimes:** a vector containing the runtime measured on every MPI rank.

Explicitly describing the available input fields reduces uncertainty and helps the model associate each value with its intended meaning.

Rather than relying on the model's interpretation of the runtime data, the prompt explicitly instructs it to compute several statistical quantities such as **average runtime, minimum runtime, maximum runtime, standard deviation and imbalance percentage**.

The imbalance percentage is further defined using the exact mathematical expression ensuring that every analysis follows the same definition of load imbalance. Providing the computation directly in the prompt removes ambiguity and improves consistency across analyses.

The prompt also specifies a fixed output structure consisting of six fields: **call-path, severity, slowest-ranks, reason, investigate and fix**.

Using a predefined response template encourages deterministic outputs that can easily be displayed within the CubeGUI interface. Since each response follows the same format, users can quickly compare different call paths without interpreting varying response styles.

A particularly important component of the prompt is the set of instructions governing the investigate and fix fields. Instead of allowing generic performance advice, the prompt explicitly distinguishes between acceptable and unacceptable responses by providing both positive and negative examples.

For the investigate field, the model is instructed to describe how the imbalance should be verified using profiling data. Examples such as comparing per-rank iteration counts, workload sizes, communication delays or synchronization times encourage recommendations that are directly applicable to HPC performance analysis. Conversely, vague suggestions such as profile the application or check performance are explicitly prohibited.

Similarly, the fix field is constrained to recommend concrete load-balancing strategies rather than generic optimization advice.

These examples serve as few-shot guidance, illustrating the desired level of specificity without requiring complete example conversations.

Finally, the prompt concludes with several formatting constraints:

These constraints reduce output variability and simplify downstream processing. Since the plugin directly displays the generated response in CubeGUI, enforcing a strict textual format avoids unnecessary parsing and improves readability.

Overall, the prompt 4.1 combines several prompt engineering techniques, including explicit role assignment, clear task specification, structured input descriptions, mathematical definitions, output templates, example-based guidance and formatting constraints. Together, these techniques improve the reliability, consistency, and practical usefulness of the generated load imbalance analyses while keeping the interaction between CubeGUI and the language model deterministic and straightforward.

```
You analyze EXACTLY ONE call path.

Input:
- call_path
- collapsed_runtime
- expanded_runtime
- runtimes

IMPORTANT:
runtimes[i] corresponds to rank i.

You must compute:
- average runtime
- minimum runtime
- maximum runtime
- standard deviation
- imbalance percentage

Definition:
imbalance_percent = ((max_runtime - average_runtime) / max_runtime) * 100

Output format:

call_path: "<call_path>"
severity: <percent>%
slowest_ranks: <rank list>
```

```
reason: <one sentence explaining what the runtime distribution indicates>
investigate: <one sentence describing exactly what profiling data should be
     compared>
fix: <one sentence describing a realistic balancing action>

------------------------------------------------
INVESTIGATE/FIX RULES (VERY IMPORTANT)

The investigate sentence MUST explain HOW to verify the imbalance using
    profiling data.

GOOD investigate examples:
- compare per-rank iteration counts and workload sizes for the listed ranks
- inspect message sizes and synchronization delays on the slowest ranks
- compare computational work assigned to the slowest ranks

BAD investigate examples:
- profile the application
- inspect the program
- check performance

The fix sentence MUST describe a SPECIFIC balancing strategy.

GOOD fix examples:
- redistribute mesh partitions more evenly across MPI ranks
- rebalance particle ownership among the slowest ranks
- reduce synchronization caused by overloaded ranks

BAD fix examples:
- optimize the code
- improve performance
- rebalance workload

------------------------------------------------
PROCESSING RULES
- Output ONLY the required format
- No markdown
- No bullet points
- No extra commentary
- No JSON
```

Listing 4.1: System prompt for the HPC load imbalance analysis:

Prompt sensitivity experiments performed during development revealed that even small wording changes could significantly affect output quality. For example, prompts that vaguely requested "performance analysis" often caused the model to speculate about unrelated bottlenecks or generate overly generic optimization advice. Similarly, allowing unrestricted responses frequently resulted in verbose explanations disconnected from the actual runtime evidence.

In contrast, prompts that explicitly constrained reasoning to runtime differences produced significantly more stable and technically relevant responses. Adding clear instructions such as "DO NOT invent causes unsupported by runtime data" reduced hallucinated explanations considerably. Explicitly distinguishing between imbalanced ranks and waiting ranks improved the correctness of rank classification.

Another important observation was that shorter (caveman language but a little more understandable) and more focused prompts generally produced more reliable outputs than excessively detailed prompts. Overloading the model with large instructions and unnecessary context often degraded response quality, particularly when analyzing many call paths sequentially. This reinforced the importance of concise but information-dense prompt construction throughout the system design.

Overall, prompt engineering in this work is not treated as a secondary implementation detail, but as a fundamental component of the analysis pipeline itself. The effectiveness of the LLM depends not only on the profiling data provided, but also on how carefully the reasoning process is constrained and guided through prompt design.

4.3 Context Handling and Session Drifts

Modern API-based language models process each request independently unless prior context is explicitly included in the prompt. This stateless interaction model simplifies integration but introduces constraints when dealing with large-scale HPC profiling data. In particular, hierarchical call trees generated by CubeGUI may contain significantly more information than can be efficiently represented within a single prompt. As a result, the analysis must be decomposed into smaller units that fit within the available context window.

Although each request is independent at the API level, language models may still exhibit implicit continuity in interpretation when similar structured inputs are processed sequentially. In the context of HPC traces, this can lead to inconsistent assumptions if multiple call paths are analyzed within a single conversation or without strict input isolation. However, in this work, this effect is mitigated by enforcing a strictly single call-path per request design.

To address these limitations, we adopt a strictly iterative and externally managed context-handling strategy. Instead of treating the profiling dataset as a large conversation, the analysis is divided into smaller independent reasoning tasks. Each API request processes exactly one call path together with its associated runtime metrics. This design intentionally minimizes unnecessary contextual dependencies and reduces the likelihood of context contamination between unrelated call paths.

Sequential processing is used to submit call paths with their corresponding system tree one after another to the LLM. This ensures that execution remains deterministic from the perspective of the user interface while preventing unnecessary coupling between unrelated analyses.

The context handling strategy can therefore be characterized as a hybrid architecture:

- Structural traversal and state management are handled entirely within the plugin.
- The LLM receives data for a single call path at a time.
- No conversational memory is required, as all analysis is stateless.

4.3.1 Alternative approach: Agent-Based Integration

An alternative design would be agent-based architecture in which the LLM would have direct access to CubeGUI. Instead of receiving preprocessed profiling data, the language model would dynamically interact with CubeGUI through tool calls, request additional profiling information, traverse the call tree and iteratively refine its analysis based on intermediate observations.

Although such an approach offers greater flexibility for interactive performance investigations, it was not adopted for this work for 3major reasons:

1. **Reduced scalability** In an agent-based architecture, the LLM would typically require multiple interactions with CubeGUI before producing a diagnosis. While this can improve exploratory investigations, it significantly

increases the number of model-tool exchanges required for each call path which would cause CubeGUI to become considerably slower and more resource-intensive, resulting in longer loading and analysis times.

2. **Sufficient information is already available.** Each analysis request contains the runtime statistics required to identify load imbalance and generate corresponding investigation and balancing recommendations. Since the diagnosis is performed independently for each call path, additional access to CubeGUI would not necessarily provide information that is relevant to the intended task. Consequently, the additional overhead of an agent-based architecture would not be justified by a proportional improvement in analysis quality.

The adopted pipeline provides a clear separation of responsibilities. CubeGUI remains the authoritative source of profiling data and extracts the call paths contributing most to runtime, while the LLM operates as a stateless load imbalance detector and reasoning component that receives validated inputs and generates human-readable analyses. This design improves reproducibility, reduces implementation complexity and ensures that each call path is analyzed independently.

Overall, a stateless pipeline is more appropriate for the objectives of this work. While an agent-based architecture could be advantageous for interactive root-cause analysis and exploratory performance investigations, the chosen approach is better suited for large-scale automated analysis of complete call trees.

4.4 API Communication

4.4.1 Overview of Communication Model

The communication with the LLM is implemented as a sequential request pipeline integrated into the CubeGUI plugin. After profiling data has been extracted from CubeGUI, each call path is transformed into a compact plaintext representation containing statistics of the associated runtime distribution.

Unlike traditional interactive LLM systems, the communication pattern in this system follows a request–response pipeline with strict sequential chaining. Each request corresponds to a single call path produced during traversal of the CubeGUI call tree. This design ensures that API interaction remains tightly coupled to the profiling workflow rather than supporting open-ended conversational context.

4.4.2 Request Construction

Each API request is constructed as a JSON payload following the chat-completions format. It consists of a system message encoding the analysis constraints and a user message containing the current call path together with precomputed runtime statistics.

These statistics are generated inside the plugin during call-tree traversal and include derived descriptors of the per-rank runtime distribution, such as the number of ranks, average runtime, median runtime, minimum runtime, maximum runtime, standard deviation, imbalance percentage, worst-performing rank, detected slow ranks, and a summary of the overall runtime distribution. The call path and its associated metrics are combined into a formatted plaintext block before transmission, rather than being encoded as a structured statistics object.

4.4.3 Transmission and Sequential Execution

Requests are sent using HTTP POST operations through Qt's networking framework. The authentication token is included in the HTTP Authorization header, while the model identifier is specified in the JSON request body.

Each request is triggered only after the previous call path has been processed, resulting in a deterministic sequential execution model over the full set of extracted call paths. Although execution is sequential at the application level, each request is treated as independent from the perspective of the language model.

4.4.4 Response Handling

Responses from the API are returned in JSON format and parsed using Qt's JSON utilities. The plugin expects a chat-completions-compatible response structure and extracts the generated output from the message content field of the first completion choice.

The request payload is manually assembled using Qt's QJsonObject and QJsonArray classes, serialized using QJsonDocument, and transmitted via HTTP POST. The response is then decoded and displayed in the CubeGUI analysis panel.

4.4.5 Asynchronous Execution Model

To maintain responsiveness within the CubeGUI environment, all network operations are executed asynchronously. The plugin triggers the next request only

after receiving and processing the previous response. This ensures that UI interaction remains uninterrupted while preserving sequential consistency across the analysis pipeline.

4.4.6 Error Handling and Robustness

The system includes basic robustness mechanisms to handle failures in external communication. These include validation of API credentials prior to request dispatch, detection of network-level transmission errors, handling of invalid or incomplete API responses, and safeguards against empty or malformed output payloads.

In each failure case, the plugin records the error state and displays a corresponding message in the analysis panel. This allows users to identify communication or response-processing failures and inspect the reported error information before restarting the analysis if necessary.

5. CubeGUI Integration of AI Driven Analysis

This chapter describes the plugin architecture, outlines the UI and reasons using an external LLM.

5.1 Plugin Architecture

The proposed system extends CubeGUI through a custom plugin that integrates LLM based reasoning directly into the existing performance analysis workflow. The plugin operates as an additional analytical layer on top of CubeGUI, allowing users to combine visual performance exploration with automated reasoning.

CubeGUI provides a plugin interface that allows external components to interact with profiling data, GUI elements and internal tree structures. The implemented plugin uses this interface to access the call tree and system tree generated from Score-P profiling data. Through the plugin API, the system can traverse profiling hierarchies, select call paths and retrieve process-level runtime metrics associated with MPI ranks.

The architecture of the plugin is centered around three primary components:

1. Data extraction and traversal
2. Prompt construction and API communication
3. Response handling and GUI visualization

The plugin begins by recursively traversing the CubeGUI call tree. For each call path, runtime values are collected from all system tree leaf nodes representing MPI ranks. The extracted runtime data is then processed locally to compute statistical descriptors of the runtime distribution, including average runtime, median runtime, minimum runtime, maximum runtime, standard deviation, imbalance percentage, worst-performing rank, detected slow ranks and a summary of the runtime distribution.

After preprocessing, the plugin constructs a structured prompt described in the previous chapter. The prompt is then sent to the external LLM through an HTTP-based API request. Once a response is received, the LLM analysis is displayed inside the CubeGUI interface.

The primary trigger is the *Analyze Load Imbalance* button integrated into the GUI. Once activated, the plugin automatically starts traversing the call tree and initiates sequential LLM requests.

Figure 5.1 presents the workflow of the proposed LLM-assisted load imbalance analysis system. The architecture is divided into three logical layers: the CubeGUI user interface, the plugin processing layer and the external LLM service. User interaction triggers recursive call tree traversal and runtime metric extraction within the plugin. The collected performance data is preprocessed to compute imbalance statistics and transformed into structured prompts. These prompts are transmitted asynchronously to an external language model through a REST API. The returned analysis is subsequently parsed and presented within the CubeGUI interface. By separating data extraction, prompt generation, API communication and response handling into distinct stages, the architecture maintains modularity while minimizing disruption to the existing HPC profiling workflow.

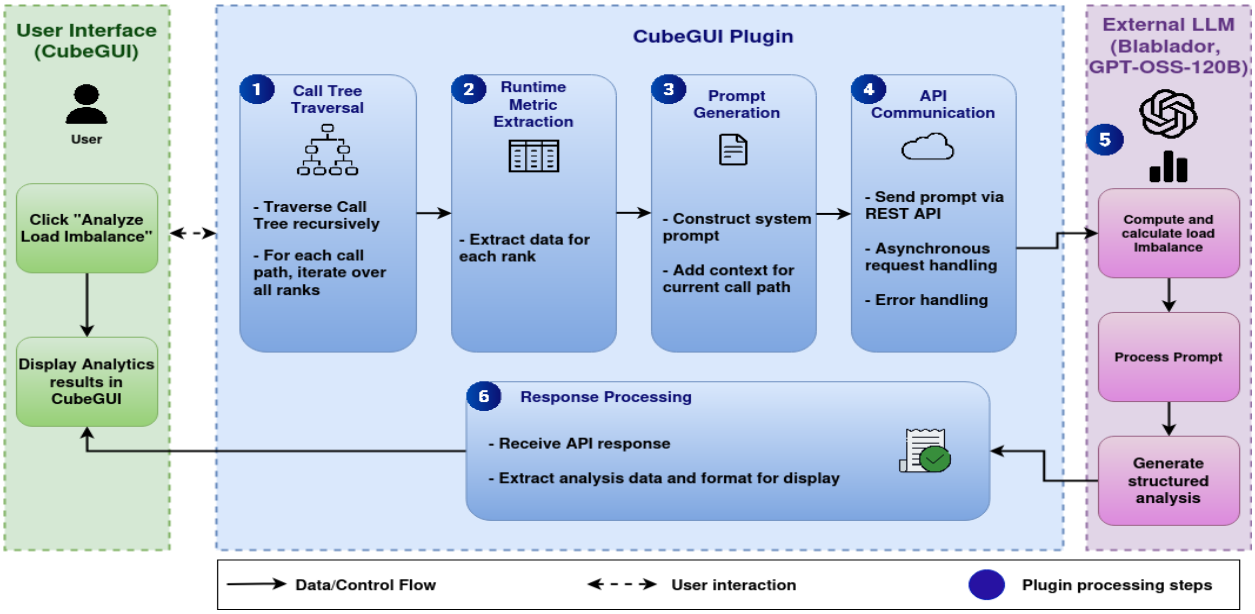

Figure 5.1: LLM-assisted load imbalance analysis workflow

5.2 UI Integration

The LLM functionality is added into CubeGUI as a plugin in order to preserve the existing profiling workflow. The AI analysis is presented within the same environment as the traditional performance views, eliminating the need to switch between external analysis tools and CubeGUI.

The plugin introduces a dedicated AI analysis panel within CubeGUI. This panel contains two primary interface components: An analysis trigger button and a read-only output display area.

The *Analyze Load Imbalance* button acts as the entry point for the automated analysis workflow. Once activated, the plugin traverses the currently loaded profiling data, extracts runtime metrics and sequentially submits analysis requests to the LLM. The generated responses are displayed inside a text output field integrated into the GUI. The responses include:

- Imbalance severity, Slowest ranks, Reasoning about observed behavior, Suggested investigation strategies, Potential balancing actions

The interface is intentionally designed to remain lightweight and minimal. Since CubeGUI already provides detailed visualization, the role of the AI panel is not to replace existing visual analysis but rather to augment it with interpretative reasoning.

The interaction workflow between the user, the plugin and the external LLM is illustrated in Figure 5.2

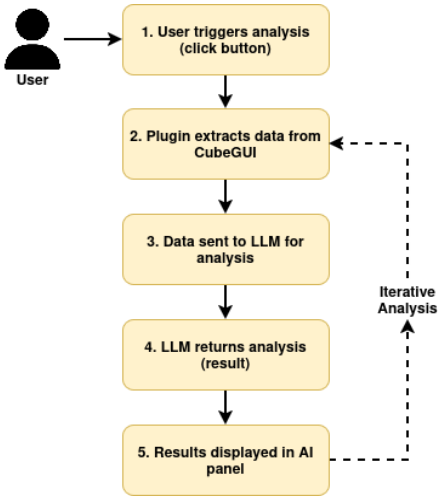

Figure 5.2: User interaction UML diagram

Constrained textual output is used rather than free-form conversational interaction. Since the primary goal is structured performance reasoning rather than chatbot-style communication, the interface prioritizes concise and technically relevant responses.

5.3 Why an externally hosted LLM

LLMs require substantial computational resources, including high-memory GPUs and specialized inference infrastructure. Running such models inside CubeGUI would significantly increase deployment complexity and hardware requirements.

External API-based models such as Blablador provide immediate access to state-of-the-art reasoning capabilities without requiring local model hosting or maintenance.

Using an external service also allows the system to benefit from improvements in model quality without modifying the plugin implementation itself. The plugin remains model-agnostic and can be adapted to different APIs or future LLMs with minimal changes.

Another important consideration is responsiveness. Although API communication introduces network latency, externally hosted inference services are often optimized for large-scale deployment and can provide scalable and operationally simpler inference compared to hosting and maintaining large language models locally.

6. Evaluation

This chapter presents the evaluation of the proposed LLM-assisted load imbalance analysis system. It defines the evaluation goals, describes the datasets and experimental setup used for testing, introduces the metrics applied to assess performance, and reports the obtained results. Finally, the findings are discussed in terms of strengths and limitations.

6.1 Evaluation Goals

The evaluations aims to answer the following:

- **Correctness of diagnosis:** Evaluation of whether the reported imbalanced ranks correspond to those observed through manual inspection along with non-hallucinating/coherent plausible fixes.
- **Consistency:** We measure whether repeated analyses of the same profiling data produce stable outputs despite the probabilistic nature of LLMs.
- **Usefulness:** Assessment of whether the generated explanations, investigation suggestions and remediation recommendations provide practical value to performance analysts.
- **Scalability:** Examination of whether the approach remains effective when applied to profiling datasets of varying sizes, ranging from small benchmark traces to large real-world HPC applications.

6.2 Datasets

The evaluation is conducted using both benchmark applications and real HPC profiles [18]. This combination provides controlled scenarios where imbalance is known beforehand as well as realistic workloads representative of production environments.

Table 6.1 summarizes some of the Cube profiles used during the evaluation. For most datasets, the proposed approach required approximately 1–5 minutes to complete the analysis after the user clicked the "Analyze Load Imbalance" button. This execution time remained largely consistent for system trees containing between 1 and 5,000 MPI ranks. However, for the largest dataset containing

8,192 ranks, the analysis time increased substantially to approximately 45 minutes. This increase is expected because the plugin collects per-rank runtime information for every selected call path and submits this information to the LLM for analysis. As the size of the system tree increases, the size of the runtime vectors included in the prompts grows proportionally, resulting in significantly larger requests and longer processing times for the LLM.

Application	No. of Roots	No. of Call paths	No. of ranks
Benchmark 1	1	7	6
Benchmark 2	1	35	10
Real-time 1	1	68	720
Real-time 2	1	3281	384
Real-time 3	1	4373	192
Real-time 4	1	6744	18
Real-time 5	1	8287	36
Real-time 6	2	712	192
Real-time 7	16	34	8,192

Table 6.1: Some of the profiles used for evaluation

Benchmark programs are used to create reproducible imbalance scenarios with known characteristics. Like applications with intentionally uneven workload distributions, synchronization delays or artificial waiting behavior. These profiles provide a controlled environment for validating whether the LLM identifies imbalance correctly.

In addition to synthetic benchmarks, real profiles collected using Score-P and visualized in CubeGUI are analyzed (ranging from 10 call paths to 10,000 call paths and upto 8,000 ranks are analyzed). These profiles contain naturally occurring performance variations and represent practical HPC performance analysis scenarios.

To ensure a diverse evaluation, datasets containing different forms of imbalance are included such as uneven computational workload distribution, process waiting caused by synchronization, communication-induced delays, mixed imbalance scenarios and balanced execution traces used as control cases. This variety allows assessment of the system across a broad range of performance behaviors.

6.3 Evaluation Metrics

Several metrics are used to assess the effectiveness of the proposed approach.

1. Agreement with Expert Analysis: The primary metric measures whether the imbalance reported by the LLM agrees with manual analysis performed using CubeGUI.

   The following aspects are compared:

   - Detection of imbalance
   - Identification of slowest ranks
   - Plausibility of generated explanations

   Since root causes cannot always be verified directly from profiling data alone, explanations are evaluated based on plausibility rather than absolute correctness.

2. Response Stability: To evaluate consistency, identical profiling inputs are analyzed multiple times. It should always produce similar conclusions across repeated executions.

3. Time reduction: The practical value of the approach is evaluated by comparing the effort required for manual inspection against AI-assisted analysis. Metrics include time required to locate imbalanced call paths manually, interpret profiling results, formulate investigation strategies. The objective is not to eliminate expert analysis but to reduce the effort required to reach an initial diagnosis.

6.4 Experimental Setup

All experiments are performed using the CubeGUI plugin implementation described in Chapter 5. The plugin extracts execution-time metric from CubeGUI call trees and system-tree process information before converting them into a structured textual representation suitable for LLM processing.

The external model used for evaluation is:

GPT-OSS-120B (OpenAI Open Model)

accessible through the Helmholtz Blablador API service [19].

Each call path is analyzed independently using the prompt structure described in Chapter 4. The stateless processing approach ensures that each diagnosis depends only on the profiling data associated with the current call path.

To prevent context-window limitations from affecting the analysis, profiling data is processed iteratively and transmitted one call path at a time. This design

keeps prompt sizes small and enables evaluation on large traces containing hundreds or thousands of call paths. The plugin configuration remains unchanged throughout all experiments to ensure reproducibility.

6.5 Results

1. **Interpretation Accuracy:** Across all evaluated datasets, the detected imbalances were consistent. The explanations generated were plausible for the provided data.

2. **Consistency:** To evaluate response stability, the same profile was analyzed multiple times using the same prompt configuration. The generated outputs were highly consistent across repeated executions. In particular, imbalance severity estimates, identification of slowest ranks mostly remained unchanged across runs. Minor variations were occasionally observed in the imbalance percentage (upto 1-2 percent) and wording of explanations, but these differences did not affect the underlying diagnostic conclusions.

3. **Practical Usefulness:** In addition to identifying problematic call paths, the system generated concise explanations based on the observed runtime distribution and suggested specific validation steps. Although the proposed fixes cannot be guaranteed to resolve the observed behavior without application-specific knowledge, they provide reasonable starting points for further investigation. This reduces the amount of manual interpretation required when navigating large call trees and performance profiles.

4. **Scalability:** The approach demonstrated reasonable scalability for large call trees by analyzing only the call paths that contribute more than 5% of the total runtime, as selected by CubeGUI. The current implementation supports system trees of up to 8,192 ranks. Larger system trees would exceed the model's context window and are therefore not currently supported.

The documents in [18] contain the AI results for multiple cube profiles and whether the AI responses were useful or not. Below 6.1 is a small example of the LLM results.

The following illustrates a response generated by the LLM:

```
Analyzing call tree...
==================================
```

```
REGION 1
-----------------------------------
call_path: "Invisible call tree root/
    arts_scorep_advance_iter_lpt_nocompiler"
severity: 5.2%
slowest_ranks: [21,41,42,53,54,70,73,74,85,86,90]
reason: A small subset of ranks (around 10) take noticeably longer than the
     bulk, indicating a
load-imbalance where those ranks hold more work.
investigate: compare the amount of mesh elements or particle counts
    assigned to ranks
21,41,42,53,54,70,73,74,85,86,90 with the average per-rank workload.
fix: repartition the domain so that the heavy mesh/particle blocks
    currently on the identified slow
ranks are split and redistributed to under-utilized ranks.

===================================
REGION 2
-----------------------------------
call_path: "Invisible call tree root/
    arts_scorep_advance_iter_lpt_nocompiler/MPI_File_set_view"
severity: 7.5%
slowest_ranks: [160]
reason: One rank (160) runs noticeably longer than the rest, indicating an
    I/O load imbalance for
that MPI process.
investigate: compare the amount of data each rank accesses and the size of
    the MPI_File_set_view
for rank 160 versus the other ranks using the I/O profiling counters.
fix: redistribute the file view offsets so that each rank reads a more
    equal portion of the file,
balancing the I/O workload across ranks.

===================================
REGION 3
-----------------------------------
call_path: "Invisible call tree
root/arts_scorep_advance_iter_lpt_nocompiler/scp_lpt_step/
    MPI_File_write_all"
severity: 2.5%
```

```
slowest_ranks: [160]reason: The runtimes are tightly clustered around 250
    ms with only rank 160 exceeding 257 ms,
indicating a modest load imbalance.
investigate: compare per-rank iteration counts and data volumes for rank
    160 against the average
ranks to see why it spends more time.
fix: redistribute a portion of the mesh or I/O workload from rank 160 to
    neighboring ranks to
equalize computation time.

Finished analyzing all selected regions.
```

Listing 6.1: A part of the response generated by the LLM for selected call paths.

6.6 Discussion

The evaluation demonstrates that LLMs can provide meaningful assistance during HPC performance analysis with concise prompts and when supplied with structured profiling data.

6.6.1 Strengths

It has the ability to consistently identify imbalanced areas in a program and summarize performance observations in a concise and interpretable manner. The generated investigation recommendations frequently point analysts toward useful follow-up actions, reducing the effort required for initial diagnosis.

Another important advantage is scalability. Although not the largest system tree (largest can be upto 300,000 ranks in system tree) but the application analyzes profiles with relatively large (8,192 system tree ranks). By processing call paths independently, the system reduces the context-window limitations.

6.6.2 Limitation

As the number of ranks increases, this array grows linearly. At around 8,192 ranks, the prompt is still within the LLM's maximum request size. For system trees with more than roughly 16,000 ranks, the runtime vector becomes too large, causing the API to reject the request with a "request too large" (context length) error before the model can process it. This behavior is rather viewed as a finding

of current LLMs rather than of the plugin's limitation, since it is fundamentally constrained by the model's maximum context window.

The most significant limitation is that the model does not possess direct knowledge of application source code or runtime semantics. As a result, proposed fixes should be interpreted as hypotheses rather than verified root causes. Although most suggestions are plausible, they cannot always be confirmed solely from profiling information.

A potential way to address this limitation would be to provide the LLM with additional information such as application source code or execution logic. However, this approach introduces several practical challenges. The code is usually very large to fit in the available context window of current LLMs. Furthermore, retaining large amounts of historical information across multiple interactions might increase the chances of hallucinations, as the model could begin drawing conclusions from unrelated parts of the code rather than from the profiling data currently under analysis.

For this reason, the implemented system intentionally treats each call path independently and limits the information provided to the model. This design reduces contextual interference and helps ensure that conclusions are derived primarily from observed runtime behavior.

While future LLMs with larger context windows, improved memory mechanisms and lower hallucination rates may enable deeper integration of source-code information, such capabilities remain beyond the scope of the current work.

7. Conclusion and Future Work

This last chapter summarizes the thesis and lists future works.

7.1 Summary

This thesis investigated the use of LLMs as supportive tools for HPC performance analysis, with load imbalance serving as the primary use case. To achieve this, a prototype AI plugin was developed and integrated into CubeGUI, enabling structured profiling data to be extracted, transformed and analyzed using an external LLM.

The work demonstrated that profiling information represented through call trees and process-level execution metrics can be understood by LLMs. A complete workflow was designed, covering data extraction from CubeGUI, preprocessing of profiling information, prompt generation, communication with the LLM and presentation of analysis results within the UI. Special attention was given to controlling model behavior through prompt engineering and structured output constraints in order to improve reliability and reduce hallucinations.

The evaluation showed that LLMs are capable of accurately and consistently identifying load imbalance from structured performance data. Across the evaluated benchmark and real-world profiles, the plugin filtered relevant call path contribution over 5% to the overall runtime and the LLM correctly identified the ranks contributing most strongly to execution-time deviations. The generated explanations were generally aligned with established HPC performance analysis practices and provided useful starting points for further investigation.

A key finding of this work is that LLM's effectiveness depends heavily on the quality and structure of the provided prompts. Large prompts resulted in hallucinations and inconsistent responses (different responses each time the same profile is analyzed). Using caveman prompting not only reduced the token usage but also gave consistent responses.

The work also highlights the importance of context management when applying LLMs to HPC performance analysis. Iterative processing and selective context preservation are necessary for scalable analysis of large performance traces.

Overall, the results show that LLMs are not a replacement for traditional profiling tools or expert analysis. Instead, they can serve as intelligent assistants that help users interpret profiling data, identify potential performance issues and

generate hypotheses for further investigation. This suggests that LLM-assisted performance analysis represents a good direction for reducing the manual effort in analyzing HPC load imbalance.

7.2 Future Work

Although the current work has provided good results, it is only a starting point and there is still scope for future work. The future works include:

1. **Extension to Broader Performance Bottlenecks**
   While this work focuses on load imbalance, the proposed approach could be extended to other HPC performance problems such as communication overhead, synchronization delays, memory bandwidth limitations and I/O bottlenecks. Incorporating additional performance metrics and hardware counter data may enable more comprehensive diagnosis across multiple layers of application execution.

2. **AI Agent-Based Interactive Analysis in CubeGUI**
   An important extension of the current system is the transition from a stateless analysis pipeline to an agent-based interaction model within the GUI. In such a setup, the LLM would not only receive call paths and system trees but would also be able to actively interact with CubeGUI, query additional profiling data, traverse call trees dynamically and refine its reasoning based on intermediate results. This would enable exploratory performance analysis workflows in which the model behaves as an interactive assistant rather than a single-pass analysis component.

3. **Context-Aware and History-Based Analysis**
   A further extension for when LLMs can handle history better is the development of context-aware analysis across multiple call paths. Instead of treating each call path independently, future systems could maintain intermediate summaries of previously analyzed regions and construct a global view of application behavior. This would enable the detection of performance patterns that span multiple levels of the call tree and improve the coherence of large-scale analysis.

4. **LLM-Assisted Performance Optimization**
   Beyond diagnostic capabilities, LLMs could also support performance engineering tasks by generating optimization suggestions at a higher level of ab-

straction, potentially assisting developers in identifying code regions where restructuring or parallelization changes may improve performance.

Bibliography

[1] NVIDIA Corporation. High-performance computing (hpc), n.d. URL https://www.nvidia.com/en-us/glossary/high-performance-computing/.

[2] Vinod Mohan. Hpc performance challenges and how to overcome them, June 2025. URL https://www.datacore.com/blog/hpc-performance-challenges-and-how-to-overcome-them/.

[3] Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool, 2015. ISSN 1877-0509. URL https://www.sciencedirect.com/science/article/pii/S187705091501128X. International Conference On Computational Science, ICCS 2015.

[4] *Score-P User Manual*. Score-P Project, n.d. URL https://perftools.pages.jsc.fz-juelich.de/cicd/scorep/tags/latest/pdf/scorep.pdf. Accessed: 2026-07-07.

[5] Alan Zeichick. Ai vs. gen ai vs. ml: Key differences, 2025. URL https://www.oracle.com/de/artificial-intelligence/ai-vs-gen-ai-vs-ml/.

[6] Google Cloud. Artificial intelligence vs. machine learning, nd. URL https://cloud.google.com/learn/artificial-intelligence-vs-machine-learning.

[7] Noujoud Nader, Patrick Diehl, Steve Brandt, and Hartmut Kaiser. *LLM and HPC: Benchmarking DeepSeek's Performance in High-Performance Computing Tasks*, page 626–638. Springer Nature Switzerland, November 2025. ISBN 9783032076120. doi: 10.1007/978-3-032-07612-0_48. URL http://dx.doi.org/10.1007/978-3-032-07612-0_48.

[8] Patrick Diehl, Noujoud Nader, and Deepti Gupta. Llm-hpc++: Evaluating llm-generated modern c++ and mpi+openmp codes for scalable mandelbrot set computation, 2026. URL https://arxiv.org/abs/2512.17023.

[9] Intel Corporation. Hpc performance characterization and analysis. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2024-2/hpc-performance-characterization-analysis.html, 2024.

[10] Bernd Mohr and Jens Domke. Parallel programming with mpi and openmp hands-on. In *ASEAN School on High-Performance Computing and Artificial Intelligence*, ACM ASEAN HPC AI '25, New York, NY, USA, 2026. Association for Computing Machinery. doi: 10.1145/3736273.3736289. URL https://doi.org/10.1145/3736273.3736289.

[11] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010. ISBN 9781439811924. doi: 10.1201/EBK1439811924. URL https://doi.org/10.1201/EBK1439811924.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964. doi: 10.48550/arXiv.1706.03762. URL https://arxiv.org/abs/1706.03762.

[13] IBM. Transformer models, 2026. URL https://www.ibm.com/think/topics/transformer-model.

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

[15] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Comput. Surv.*, 55(12), March 2023. ISSN 0360-0300. doi: 10.1145/3571730. URL https://doi.org/10.1145/3571730.

[16] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9), January 2023. ISSN 0360-0300. doi: 10.1145/3560815. URL https://doi.org/10.1145/3560815.

[17] Mehul Gupta. What is caveman prompt? reduce llm token usage by 60%. *Data Science in Your Pocket (Medium)*, April

2026. URL https://medium.com/data-science-in-your-pocket/what-is-caveman-prompt-reduce-llm-token-usage-by-60-6a552734a493.

[18] Aashika. Llm in cubegui for hpc performance analysis, 2026. URL https://doi.org/10.5281/zenodo.21258488.

[19] AI@JSC. Blablador api key access, 2026. URL https://sdlaml.pages.jsc.fz-juelich.de/ai/guides/blablador_api_access/.