JÖNKÖPING UNIVERSITY
School of Engineering

Task-Adapting LLMs for Software Reliability

Enhancing Memory Leak Detection with External Knowledge Augmentation and Fine-Tuning in LLM

PAPER WITHIN *Artificial Intelligence*
AUTHOR: *Athira Asalatha Rajendran, Bincy Annamma Saji*
TUTOR: *He Tan*
JÖNKÖPING May 2025

This exam work has been carried out at the School of Engineering in Jönköping in the subject area Computer Science. The work is a part of the two-year Master of Science in Engineering programme. The authors take full responsibility for opinions, conclusions and findings presented.

Examiner: Vladimir Tarasov
Supervisor: He Tan
Scope: 30 credits
Date: 2025-05-11

Mailing address:	Visiting address:	Phone:
Box 1026	Gjuterigatan 5	036-10 10 00 (vx)
551 11 Jönköping		

Table of Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
CD	Continuous Deployment
CI	Continuous Integration
DL	Deep Learning
FN	False Negative
FP	False Positive
FAISS	Facebook AI Similarity Search
GB	Gigabyte
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
KB	Knowledge Base
LAMeD	LLM-Assisted Memory Detection
LLM	Large Language Model
LLVM	Low Level Virtual Machine
LoRA	Low-Rank Adaptation
ML	Machine Learning
MLD	Memory Leak Detector
MLP	Multi-Layer Perceptron
PEFT	Parameter-Efficient Fine-Tuning
QA	Question Answering
RAM	Random Access Memory
RAG	Retreival-Augmented Generation
RQ	Research Question
SOTA	State-of-the-Art
TN	True Negative
TP	True Positive

Abstract

Detecting memory leaks in C programs remains a persistent and critical challenge. It is particularly applicable in systems that rely on complex libraries such as GLib and GStreamer. These libraries introduce unique memory ownership patterns and lifecycle rules that traditional analysis tools often fail to interpret correctly. Static analysis tools may miss tiny leaks or generate False Positives (FP) due to limited contextual understanding. Dynamic tools like Valgrind are having significant runtime overhead and are not suitable for all development environments. To address these challenges, this thesis proposes a hybrid memory leak detection system that explores the capabilities of Large Language model (LLM). The system combines two key strategies. A Low-Rank Adaptation (LoRA), a Parameter Efficient Fine-Tuning Technique (PEFT) that adapts the LLM using a small, task-specific dataset and Retrieval-Augmented Generation (RAG), which provides the model with external, domain-specific knowledge during inference. Together, these techniques allow the model to analyze memory safety both from learned examples and relevant documentation. An uncertainty scoring mechanism is also introduced to control the influence of LoRA based on how reliable the retrieved context is. If the external documentation is well-matched, the model relies more on it; if it is uncertain or incomplete, the model falls back on its fine-tuned knowledge. This adaptive approach helps to improve accuracy and reduces the risk of false predictions or hallucinations. The system was evaluated on a dataset of C code examples involving GLib and GStreamer. Comparative results show that the combination of LoRA and RAG, guided by uncertainty scoring, significantly outperforms baseline models in terms of accuracy, recall and precision. The proposed solution is not only lightweight and efficient but also explainable and suitable for integration into development workflows such as code reviews or Continuous Integration (CI) pipelines.

1. Introduction

Software reliability is a critical concern in modern computing systems (Fan Lin-bo et al., 2009). It particularly affects in embedded and low-level programming environments where performance and memory efficiency are most important. In languages such as C, developers are responsible for manually managing memory. This leads to issues like memory leaks very easily (Jain et al., 2020). These leaks can lead to reduced application performance, increased resource consumption and even system crashes. This makes the early detection and prevention of these memory leaks, a priority in software development(Jung et al., 2014a)(Cao et al., 2024).

Traditional methods and tools are there for detecting memory leaks. But they mostly rely on static analysis or runtime instrumentation. Even though these approaches are effective in many cases, they often fail when applied to customized software stacks or performance-constrained environments (Gangwar & Katal, 2021). Along with that they may lack the flexibility to adapt to the wide variability in code structures and library usage patterns across different projects.

In recent years, the field of Artificial Intelligence (AI) has opened new possibilities in programing field (Swamy Prasadarao Velaga, 2020)(Poldrack et al., 2023). This reflects mainly in LLMs for understanding and analyzing source code (Zhu et al., 2024). These models can learn from big code repositories. This will enable them to support tasks such as bug detection, code completion and documentation generation. However, the potential of LLMs to assist in low-level programming challenges, such as memory leak detection, remains largely unexplored (Giagnorio et al., 2025).

The goal of this thesis is to explore how LLMs can be effectively adapted to detect memory leaks in C programs. Most existing LLM models are trained on general-purpose programming tasks. They are not optimized for low-level system languages like C, where memory management issues are hard to find and critical (Mohammed et al., 2024). To make LLMs apt for this task, they can be adapted through fine-tuning on domain-specific data or augmented with relevant external knowledge. Fine-tuning allows the model to internalize memory management patterns. Meanwhile knowledge augmentation supplies real-context aware information (Soudani et al., 2024).Even though the external knowledge augmentation technique, RAG provides external, stack-specific knowledge at inference time, it does not directly adapt the model's internal representations to the specific task. As a result, the model may retrieve relevant information but still fail to make accurate predictions if it has not learned the underlying patterns of memory misuse (Barnett et al., 2024). Similarly, the PEFT technique, LoRA allows for efficient fine-tuning of the model on task-specific data. But it lacks access to real-time external context to generalize across different libraries, Application Programming Interface (APIs) or programming environments that were not included during training (Shuttleworth et al., 2024a).

To utilize the strengths of both fine-tuning and retrieval, this thesis introduces a hybrid approach that combines LoRA and RAG. LoRA captures task-specific memory patterns efficiently, while RAG injects relevant external knowledge during inference. Together, they make the model both adaptable and context aware. To further enhance reliability, an uncertainty-based adaptation mechanism is introduced. This mechanism dynamically balances the influence of LoRA and RAG based on the model's confidence. This ensures an effective memory leak detection across diverse C codebases, including those using GLib and GStreamer.

In this thesis work, a lightweight and adaptable framework has been developed to enable automated detection of memory leaks. This is an emphasis on applicability of LLMs to real-world systems programming tasks. The effectiveness of the proposed method is assessed through comparative evaluation against a baseline model, using standard classification metrics such as accuracy, recall and precision.

1.1 Problem Statement

Memory management remains one of the cornerstones of reliable software development (Jain et al., 2020). This affects mainly in low-level and embedded systems because, there the performance and resource control are critical. The higher-level languages normally offer automatic garbage collection. But the C programming language places the responsibility of memory allocation and deallocation entirely on the developer. Even though this design enables precise control over memory, it makes C programs always susceptible to the memory management bug and memory leaks (Zhiqiang Liu et al., 2015a).

A memory leak occurs when a program fails to release memory that is no longer in use (Jain et al., 2020). This will cause gradual resource exhaustion. This in turn can eventually lead to system instability or crashes. These issues are mainly problematic in systems that must run for long periods or operate under strict memory constraints. These systems include the embedded devices, networking equipment and real-time control systems (Poornima, 2013). Even though the memory leaks are a well-known issue, it continues to be one of the most difficult classes of bugs to identify, debug and eliminate. Their presence often becomes problematic only under specific runtime conditions (Zhiqiang Liu et al., 2015a). The actual cause of the memory leak can be hard to find because it is often hidden behind complex code structures or older parts of the codebase.

Several tools and techniques have been developed to find the memory leaks in C code. These can be broadly categorized into static analysis tools and dynamic analysis tools (Aslanyan et al., 2022). Clang static analyzer and Cppcheck (Yaozong et al., 2023) are examples of static analysis tools while Valgrind and AddressSanitizer are dynamic analysis tools (Azhari et al., 2021). These tools provide valuable insights about memory leaks scenarios. But on the other hand, they come with significant limitations also. Static analyzers often struggle to find the logic and structure of complex control flows, dynamic data structures or stack-specific macros and allocations that vary across projects. Meanwhile, dynamic analysis tools require running the application under special instrumentation layers (Gangwar & Katal, 2021) (Poornima, 2013). They may result in substantial performance overhead and limited real-time applicability which makes them impractical for some deployment environments.

Furthermore, both static and dynamic tools, generally do not fully serve their purpose in application-specific or stack-specific context in which the program is executed (Gangwar & Katal, 2021). For example, the C projects that use specialized libraries such as GLib or GStreamer. Here memory ownership conventions, container management and lifecycle control normally depend on non-standard patterns. So, the generic tools are not designed to interpret these scenarios. As a result, they might give wrong warnings or miss small memory leaks caused by incorrect use of certain library functions.

To overcome the limitations of traditional static and dynamic tools, AI techniques are also started to use in software analysis. AI offers the ability to learn pattern. They are able to make predictions and support automation without having a fixed hard coded rules (Andrzejak et al., 2017). In the context of memory leak detection, AI will be able to identify issues that static rules or runtime instruments might miss by learning from real-world usage patterns and contextual behaviors. AI-based systems can adapt and evolve with codebases over time, which is not possible for hardcoded tools.

One major application of AI in this field is Machine Learning (ML) and Deep Learning (DL) models (Andrzejak et al., 2017)(Cao et al., 2022). These approaches can normally treat memory leak detection as a classification problem. That is the source code samples are labeled as leaky or non-leaky and used to train models such as decision trees, support vector machines or neural networks. These models usually analyze and understand patterns in the code, like memory allocation, loops, pointers and how the code flows. After training these models, they can check new code and warn developers if something looks like a memory leak.

Even though traditional ML and DL models can make success in some controlled experiments in identifying memory leaks in codes, they face many limitations when applied to real world memory leak detection. Classifying source code as “leaky” or “non-leaky” is not a straightforward task. The memory-related issues mostly depend on runtime behavior, indirect function calls and deep library abstractions and that cannot be captured by simple labels. These models typically rely on surface-level patterns. But they will not be able to analyze the deeper context, such as function contracts, pointer aliasing or resource lifecycles. Another challenge is the scarcity and inconsistency of labeled datasets in real-world memory leaks. They are usually small, span multiple functions or files and are difficult to label reliably. This can lead to noisy data that misleads model learning (Cao et al., 2024). Additionally, traditional models cannot use external knowledge such as documentation or stack-specific conventions, which are critical for differentiating correct from incorrect memory usage. Training these models on long, real-world code brings another complexity. The longer code snippets may include multiple logic branches or combined behaviors that do not cleanly map to a single class label. This makes both the annotation and classification process ambiguous and less effective. Along with these, such models normally tend to lack generalizability. That is a classifier trained on one project often performs poorly on another without significant retraining. These combined limitations make conventional ML and DL approaches impractical for building a context-aware memory leak detection systems in diverse, real-world codebases.

LLMs offers an alternative to these approaches. They can learn general programming semantics and provide support for a variety of tasks such as code generation, refactoring and even bug detection. However, general-purpose LLMs is not trained or fine-tuned to understand the specific memory management patterns found in C or their related frameworks. Also, their utility in memory leak detection, especially in embedded or systems-level codebases remains severely limited. Moreover, using an LLM without domain adaptation have the risks in accuracy and trustworthiness (Mohammed et al., 2024). That is without understanding the stack-specific context or the actual library semantics, these models may hallucinate incorrect suggestions, overgeneralize behaviors or miss interpret memory violations. This lack of trustable and explainable behavior is a major barrier to their practical adoption in systems programming.

These challenges make a clear research gap in designing lightweight and context-aware memory leak detection systems that uses LLMs effectively. A system that can learn from task-specific memory leak patterns, retrieve stack-aware documentation at inference time and adjust its behavior based on uncertainty would fill this gap. Such a system must operate efficiently even on modest hardware and support practical use cases, such as assisting developers during code review or catching memory issues early in the development cycle. This can avoid heavy runtime instrumentation or deep toolchain integration.

The thesis addresses this gap by proposing a novel approach that combines PEFT (Ramesh et al., 2024) and RAG (Wei et al., 2025) into an LLM-based framework for memory leak detection. The system is made to be flexible and easy to understand. It uses uncertainty scoring to decide how confident the model is and to adjust its behavior based on the context. This approach helps to connect traditional static analysis tools with the strengths of LLMs especially in a field where this kind of combination has not been explored much yet.

1.2 Objectives

The research aims to develop a context-aware memory leak detection system with the use of the capabilities of LLMs. It can overcome the limitations of existing static, dynamic and ML based methods. The focus is on creating an LLM based model to effectively understand memory usage patterns specific to C code. Within C code, it deals the behavior of common libraries like GLib and GStreamer. The domain knowledge about this libraries and memory leak patterns enables the LLMs to make confident and context-driven decisions when classifying memory-related behavior.

Most existing LLM models are trained on general-purpose programming tasks. They are not optimized for low-level system languages like C. To address this, LLMs can be adapted through fine-tuning on domain-specific data or augmented with relevant external knowledge. Fine-tuning allows the model to internalize memory handling patterns specific to C code, while knowledge augmentation provides real-time, context-aware information that supports decision-making in diverse and unfamiliar environments.

Among the popular augmentation methods, RAG allows the model to bring external, stack-specific knowledge during inference. However, RAG does not influence the model's internal parameters. So, it lacks the ability to refine the model's reasoning for complex memory usage scenarios. It may retrieve relevant information, but still produce inaccurate predictions if the model has not learned the previously underling memory management patterns. On the other hand, fine-tuning techniques like LoRA enable efficient adaptation of large models using small, targeted datasets. It helps to encode memory leak patterns directly into the model's internal representation without requiring full model retraining. But it lacks access to real-time external knowledge, which makes it difficult to generalize across unfamiliar libraries APIs or domain-specific usage patterns that do not present in the training data. Considering these complementary strengths and limitations, combining fine-tuning and retrieval mechanisms becomes a logical step toward building a more effective solution. While LoRA ensures that the model is fine-tuned to specific memory leak behaviors, RAG equips it with dynamic access to external documentation and library-specific context.

To achieve this, the thesis proposes a hybrid framework that combines a PEFT with RAG. This helps the system stay flexible while still using important, stack-specific information from external documentation to understand the code better.

The proposed system uses LoRA for fine-tuning the LLM efficiently with a small amount of task-specific data. This makes it possible to adapt the model to understand leak-prone and non-leaky memory patterns in C code without using large computational resources. The model also uses RAG for deeper analysis for the prediction. This means it can pull in relevant information from external documentation while analyzing code. This will mainly be the specific GLib or GStreamer functions for handling memory. It can help the model to make more accurate and informed decisions in real-time.

In recent research, combining retrieval and fine-tuning has developed more than simple inference-time augmentation. A common technical strategy involves incorporating retrieved documents directly into the training phase. Here the fine-tuning data consists of both the original task input and the relevant retrieved context. This allows the model to learn associations between input patterns and external knowledge. This may help to effectively train LLM to interpret and integrate retrieved information rather than treating it as an auxiliary prompt. In some architectures, the retriever is fixed, and the model is optimized to make use of its outputs. There exists another approach that the retrieval mechanism is trained jointly or adaptively to improve alignment between what is retrieved and what the model expects. These approaches help the model develop deeper context awareness while retaining efficient fine-tuning advantages. This makes them suitable for tasks that demand both reasoning and real-time adaptability.

1.3 Research Question

To systematically approach the proposed idea, this thesis defines two specific Research Questions (RQs). These questions are designed to guide the formulation and evaluation of an innovative and unique method that integrates PEFT with RAG. The study applies this method to the task of memory leak detection in C code, using implementation and performance analysis to validate its effectiveness compared to baseline models.

The RQs addressed in this thesis are

R1. How to combine knowledge augmentation and PEFT to make an LLM task-specific for memory leak detection in GLib and GStreamer- base C code?

R2. How do accuracy, recall and false positive rates vary when applying a task-specific LLM to memory leak detection in GLib and GStreamer-based C code, particularly one enhanced through the combination of knowledge augmentation and PEFT?

1.4 Scope and Limitations

This thesis focuses on the detection of memory leaks in C codebases, particularly those which use the GLib and GStreamer libraries. The system is designed to adapt a general-purpose LLM for memory leak detection through a hybrid approach. It combines LoRA and RAG. The model is further improved by an uncertainty scoring mechanism to adjust inference behavior based on prediction confidence.

The study is limited to static code analysis without executing the programs. It does not involve dynamic runtime monitoring, profiling or actual memory usage measurement during execution. The focus is on detecting memory leaks from static C source code samples. Another limitation relates to the input size capacity of the LLM using for the model. Token limitations are applicable based that LLM taken for the model. So, the system is best suited for analyzing functions, modules, commits or moderately sized code blocks rather than entire large-scale software projects in a single pass. For very large files, a pre-processing step to divide the code into manageable units would be required.

Additionally, the effectiveness of LoRA fine-tuning depends on the quality and diversity of the labeled training data. A small or less diverse fine-tuning dataset may limit the model's ability to generalize across various memory usage patterns or library conventions. On the retrieval side, the performance of RAG is influenced by the size quality and structure of the retrieval database. Large-scale documentation sets may introduce latency or require more advanced indexing to maintain efficient access during inference.

Despite these limitations, the proposed system offers a flexible, efficient and explainable alternative to traditional memory leak detection methods, specifically in low-level and embedded programming environments where stack-specific knowledge plays a critical role.

1.5 Thesis Structure Overview

The thesis follows a clear and structured format that guide through the development, implementation and evaluation of the proposed system. This thesis is organized into eight chapters that together explain the motivation, design and results of the study.

Chapter 1 introduces the topic, outlines the problem, presents the RQs and objectives. Chapter 2 gives background on memory management in C, traditional detection tools and the role of LLMs, along with related work. Chapter 3 explains the research methodology, including how the experiment was designed, how data was prepared and how the system was built and evaluated. Chapter 4 describes the full system implementation, including how LoRA, RAG and uncertainty scoring were combined. Chapter 5 details the technical tools and setup used, such as the knowledge base structure, embedding methods and prompt formatting. Chapter 6 presents the results of testing different model setups and explains how well the system performed. Chapter 7 discusses what the results mean, the practical value of the system and the challenges faced. Finally, Chapter 8 concludes the thesis and suggests directions for future improvement and expansion.

2. Background and Related work

2.1 Memory Management in C

Memory management is a critical part of programming in C (Younan et al., 2010). Higher level languages automatically handle the allocation and deallocation of the memory. But C language does not automatically manage memory for the developer. The programmer must manually request memory from the system and release it when it is no longer required. In C, memory is usually allocated using functions like malloc() and calloc(). These functions reserve a block of memory on the heap for the program to use. When the program is finished using the memory, it must explicitly free it by using the free() function. If memory that was allocated is not properly freed, it stays reserved even after it is no longer needed. This is called as a memory leak (Jain et al., 2020).

There are several common reasons for memory leaks in C programs. One among the main reasons is forgetting to free memory. That is, after using malloc() or calloc(), developers sometimes forget to call free(). This makes leaving memory stuck and unavailable for other uses (Q. Gao et al., 2015). Another reason for memory leaks is when a pointer is given a new value without first freeing the memory it was already using. If the original memory is not freed, the program cannot access it anymore, which may lead to a memory leak. Mismanaging dynamic data structures is also a reason for memory leak. Structures like dynamic arrays, linked lists or trees normally needs to be carefully handled. Every piece of memory allocated for nodes or elements must be freed properly. Otherwise, leaks can accumulate over time. As C gives the programmer full control over memory, ownership and lifecycle management, these scenarios are extremely important (Zhiqiang Liu et al., 2015b).

In many real-world C projects, developers use external libraries that manage memory internally (Younan et al., 2010). GLib and GStreamer are those type of libraries. They introduce their own memory allocation, ownership and cleanup conventions that developers must understand and follow. Managing memory safely in these environments sometimes requires manual allocations tracking and proper usage of the library-provided functions to avoid memory leaks. Every piece of allocated memory must have a clearly defined "owner". That is some part of the code, which is responsible for freeing it. Failure to properly manage memory ownership leads to leaks, dangling pointers and even crashes in long running or embedded applications (Q. Gao et al., 2015). Good memory management practices are essential to ensure that C programs remain stable, efficient and reliable over time, especially in systems where resources are limited.

2.2 Traditional Memory Leak Detection Tools

Detecting memory leaks early is extremely important for maintaining the stability and efficiency of C programs (Jain et al., 2020). This is especially applicable for those programs which are using complex frameworks like GLib and GStreamer. There were existing tools which developers had used to find leaks and memory management problems before they cause major issues. Traditional methods for memory leak detection are mainly divided into two categories, static analysis and dynamic analysis (Gangwar & Katal, 2021). Both approaches have strengths and weaknesses depending on the nature of the application and the stage of development.

Static analysis of memory leaks uses tools such as the Clang Static Analyzer and Cppcheck (Sui et al., 2012) (Xu et al., 2010). They have been widely adopted for early detection of memory-related errors during development. These tools examine the source code without executing it. They normally analyze control flow and function interactions to identify paths where memory might be allocated but not properly freed (Aslanyan et al., 2024). They offer fast feedback and integrate easily into CI and development pipelines. However, static tools often face limitations when dealing with stack-specific

behaviors. In systems built on top of GLib and GStreamer, where ownership transfer can happen through complex mechanisms like signal handlers or reference counting, static analyzers may miss small leaks or conversely generate FPs. Static tools generally assume straightforward allocation and deallocation patterns, making them unsuited for interpreting advanced memory models that vary based on context. As a result, their usefulness is low in real-world, framework-heavy applications (Aslanyan et al., 2024) (Sui et al., 2012).

Dynamic analysis tools like Valgrind offer a better alternative for the static tools (Yu et al., 2021). They track actual memory usage at runtime. These tools use a shadow memory system to track which parts of memory are properly used. This will help them to find accurate invalid reads, writes and memory leaks (Nethercote & Seward, 2007). Even though these tools are very powerful, it makes programs run much slower. This runtime overhead makes dynamic tools unsuitable for use during normal development cycles or in performance-critical production environments (Yu et al., 2021). Dynamic analysis can detect the existence of leaks as mentioned but, it often lacks the semantic understanding needed to explain why a leak occurred, particularly when framework-specific memory patterns are involved.

Advanced static tools like LeakGuard have tried to close some of these gaps. They started using pointer escape analysis, path-sensitive summaries and function call graph exploration. This helped to find memory leaks more accurately, even in large and modular codebases (Liang et al., 2025). But this extra accuracy also makes things more complicated. LeakGuard is deeply tied to Low Level Virtual Machine (LLVM) and depends on heavy symbolic modeling. This makes it hard to use in fast-changing projects where it is a requirement to constantly recompile and reanalyze. This makes the process too slow and expensive. Along with this, LeakGuard is precise, and it usually relies on fixed ownership rules. So, it has trouble in adapting to real-world C projects, where memory management often changes or is not well documented. Other approaches, such as graph-based runtime detectors like the Memory Leak Detector (MLD) construct memory object graphs (Jain et al., 2020). They identify unreachable objects post-execution. MLD is conceptually simple and explainable. But MLD and similar systems require manual registration of root structures and they cannot easily track behaviors managed through callbacks or event-driven models, which are common in GLib and GStreamer.

ML approaches have also been used for identifying memory leaks (Yuan et al., 2022)(Jung et al., 2014b). This mainly focus on training classifiers based on runtime features like object age, frequency and lifetime to predict whether an allocation site is leak prone. This method is adapted from Java's generational garbage collection ideas (Andrzejak et al., 2017). They have achieved high accuracy on synthetic benchmarks. But these models depend heavily on hand-crafted feature engineering and runtime instrumentation. They also lack semantic understanding of code intent. This makes them poorly suited for detecting more lighter leak patterns arising from indirect memory ownership or misuse of framework internals (Yuan et al., 2022) (Andrzejak et al., 2017). So, all these mentioned tools and methods have clear limitations persist. Static tools offer speed but lack contextual reasoning. Dynamic tools provide accuracy but at an enormous performance cost. Graph-based or symbolic systems achieve precision but introduce high setup complexity and limited adaptability. ML classifiers show better performance in controlled environments but, when deals with real-world and framework-specific memory management, they were unable to identify the issues.

The tools and methods for memory leak detection have played an important role in software development. But their limitations are becoming more obvious as programs grow larger and more complex (Jain et al., 2020). This is especially true in systems built on frameworks like GLib and GStreamer. Their memory management involves complicated patterns such as delayed ownership transfer and shared references. Static and dynamic tools often fail to fully understand these advanced memory behaviors. They may miss important leaks which can mislead developers with FPs.

2.3 Large Language Models

The recent advances in AI, especially in LLMs have shown outstanding results in code understanding (Sarkar et al., 2024). They detect bugs and helps with programming tasks across a wide range of languages and projects. These models are good at recognizing general coding patterns and reasoning about program structures. This is particularly applicable in tasks involving code generation, completion, summarization and bug detection. Models such as OpenAI's Codex (Chen et al., 2021), CodeBERT (Feng et al., 2020), InCoder (Fried et al., 2023) and the recently released DeepSeek Coder have demonstrated strong capabilities in understanding code syntax and semantics across multiple programming languages. These models are trained on vast data of source code and associated documentation. It is typically taken from platforms like GitHub and Stack Overflow.

OpenAI's Codex, based on the GPT architecture, powers GitHub Copilot and is widely used for assisting with boilerplate code generation and interactive code completion (Chen et al., 2021). CodeBERT, developed by Microsoft, combines code and natural language pretraining. It has been widely used for code search, clone detection and summarization tasks (Feng et al., 2020). DeepSeek Coder, an open source LLM, excels in multilingual code understanding. It recently showed a better performance than several proprietary models on benchmarks such as HumanEval and CodeXGLUE.

However, these general-purpose models are typically not optimized for domain-specific behavior. This is especially applicable in system-level programming. Models like Codex sometimes successfully generate syntactically correct C code. But they often fail to understand the deeper memory semantics involved in frameworks like GLib or GStreamer. Such frameworks involve reference counting, ownership transfer and event-driven memory lifecycles. General LLMs are not trained to handle these scenarios effectively (Giagnorio et al., 2025).

Moreover, LLMs are known to hallucinate (Tvarožek & Haffner, 2025). That is, they may produce responses that looks perfect but incorrect into the question or scenarios. This is particularly when asked to reason about rare APIs or specialized libraries (Mohammed et al., 2024). This presents a significant challenge in domains like embedded or systems programming, where incorrect memory management can lead to performance degradation or security vulnerabilities (Le-Cong et al., 2024). They may not have enough knowledge about these libraries, and they can miss important details that are critical for reliable leak detection. Because of these challenges, it is important to adapt LLMs for better handling specialized tasks like memory leak detection in framework using C codebases. Simply using a large model as it is not going to serve the purpose. Instead, a task-specific strategies are needed to give the model access to more detailed and relevant information (Mohammed et al., 2024). Also, it must be fine-tuned its behavior without requiring enormous computing resources.

There are so many techniques available for these purposes. Among them two promising techniques suits best for this scenario. They are RAG and PEFT method, LoRA (Ramesh et al., 2024). RAG allows the model to bring in external knowledge such as documentation or examples. This can be used at the time of prediction, helping it reason about specific libraries even if they were not part of its original training (Wei et al., 2025). LoRA, on the other hand, provides a way to fine-tune large models on new tasks without retraining the entire model from scratch. This makes it much faster and more resource-efficient (Hu et al., 2021). Together, RAG and LoRA create a powerful way to make LLMs smarter and more specialized for detecting memory leaks in complex, real-world C projects.

These methods form the foundation of the system proposed in this thesis. This aims to bridge the gap between general code understanding and the specific needs of memory leak detection in specialized software stacks.

2.4 Parameter-Efficient Fine-Tuning (PEFT) and LoRA

Fine-tuning large pre-trained language models for new tasks traditionally requires updating all the model's parameters (Anisuzzaman et al., 2025). This approach normally requires a huge computational resource, large memory capacity and significant training time. This makes it impractical for many real-world applications, especially those with hardware constraints (Lv et al., 2024). PEFT is a method that overcomes this limitation (Gajulamandyam et al., 2025) (Zaken et al., 2022). This allows the model to adapt to new tasks by modifying only a small subset of parameters. Along with that, it keeps most of the original model weights frozen.

One of the most effective PEFT techniques is LoRA (Hu et al., 2021). In this method, instead of modifying the full weight matrices, LoRA inserts small, trainable low-rank matrices into certain layers of the model. This mainly comes within the attention mechanisms (Wang et al., 2024). During training, only these low-rank matrices are updated. The original pre-trained parameters of the models remain untouched. When the model is deployed, the low-rank updates can be merged into the frozen weights. This results in no additional inference latency compared to a fully fine-tuned model. This drastically reduces the number of parameters to be optimized. Sometimes optimization can reduce trainable parameters by factors as high as multiples of 10,000 while maintaining strong task performance (Pathak et al., 2023). LoRA has shown to be highly effective in adapting models to specific domains with this limited computational cost. It offers a scalable way to fine-tune only a small subset of model weights while preserving generalization (Biderman et al., 2023)(Hu et al., 2021).

Let $W \in R^{d \times k}$ be a weight matrix in a pre-trained transformer model. Figure 1 shows the weight merging in LoRA. Traditional fine-tuning updates the entire W, which is computationally expensive and parameter heavy. LoRA introduces a low-rank decomposition:

$$W' = W + \Delta W, \text{ where } \Delta W = BA$$

Here:

- $A \in R^{r \times k}$
- $B \in R^{d \times r}$
- and $r \ll \min(d,k)$ making the update efficient.

These trainable matrices A and B are randomly initialized and trained while keeping the original W frozen. During training, only A and B are updated. So, the total number of trainable parameters becomes:

$$\text{Trainable Parameters} = r \cdot (d+k)$$

This is significantly smaller than the full d·k matrix in standard fine-tuning. During inference, the full weight is approximated as:

$$W_{LoRA} = W + \alpha \cdot BA$$

Where α is a scaling factor (usually equal to r) that stabilizes training.

This formulation allows LoRA to maintain the base model's generalization while injecting task-specific adaptations efficiently (Hu et al., 2021).

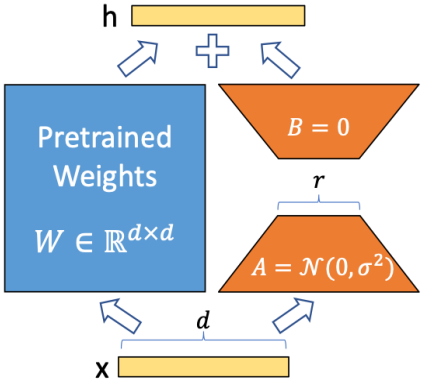

Figure 1: LoRA fine-tuning (Hu et al., 2021)

In the memory leak detection scenario, LoRA offers several key advantages. Firstly it enables fine-tuning an LLM to detect memory leaks specific to C codebases with GLib and GStreamer frameworks (Ramesh et al., 2024). This does not require access to massive computational infrastructure. Secondly it ensures that the general code understanding abilities of the base model are kept as such while selectively strengthening the model's attention to task-specific feature (Biderman et al., 2024). This includes cases such as ownership transfer, dynamic signal handling and lifecycle management in framework-heavy C systems. Recent studies have further reinforced the effectiveness of LoRA for domain adaptation (Gajulamandyam et al., 2025). It discusses that LoRA consistently outperformed other PEFT techniques when adapting LLMs to specialized domains requiring high recall and contextual understanding (Han et al., 2024). Similarly LoRA's advantages in improving task-specific performance while maintaining low resource overhead, making it a suitable choice for deployment on standard development hardware (Hu et al., 2021). In memory leak detection tasks for libraries such as GLib and GStreamer, the memory is managed through custom reference counting, signal-based ownership and callback-bound lifecycles. LoRA enables effective fine-tuning by allowing the model to specialize in such domain-specific patterns without sacrificing its general code understanding.

Customizations are further introduced in LoRA with advanced techniques such as Adaptive LoRA and Dynamic LoRA (Z. Liu et al., 2024). These methods further improves the performance by dynamically adjusting the strength or rank of the updates based on task complexity or model uncertainty (Wang et al., 2024). This dynamic adjustment is relevant in adapting to specific tasks like memory leak detection which normally depends on varying levels of semantic complexity. Adaptive LoRA enables stronger adaptation where uncertainty of the model is high. Also, this makes a lighter adaptation where the model is already confident. This helps in achieving an optimal balance between detection accuracy and computational efficiency.

Even though LoRA is having many advantages, this alone has certain limitations when applied to complex tasks such as memory leak detection in system-level C codebases. One key drawback is that LoRA does not introduce any new external knowledge into the model. That is, LoRA only fine-tunes existing parameters using the provided task data. This becomes a major issue in domains like GLib or GStreamer, where critical knowledge about reference counting, ownership rules and API usage is often not present in the original pretraining database (Mohammed et al., 2024). Additionally, LoRA's static fine-tuning does not adapt well to evolving or context-dependent scenarios. It includes incorporating updated library documentation or handling rare, dynamically discovered memory patterns. Since the fine-tuned parameters are fixed after training, the model cannot learn from new information unless re-trained (Le-Cong et al., 2024). This lack of flexibility limits its effectiveness in environments where dynamic, real-time reasoning is required. Therefore, to overcome these shortcomings, LoRA requires some additions for dynamic knowledge injection that reference up-to-date external knowledge at

inference time, resulting in a more accurate and context-aware solution. In short, PEFT method LoRA can offer an ideal mechanism to specialize large models for memory leak detection in C codebases only when it is able to include the external knowledge also.

2.5 Retrieval-Augmented Generation (RAG)

RAG is a hybrid approach that improves the capabilities of LLMs. In the context of improving accuracy and grounding responses in verifiable knowledge, it has emerged as a complementary approach. This has been achieved by combining two powerful components, that is retrieval and generation. In this method, the system is not depending solely on the model's internal memory. Instead, it allows models to consult an external knowledge base during inference. This can dynamically bring in relevant information to guide their responses (Lewis et al., 2021). This significantly improve the ability to handle domain-specific or infrequently encountered knowledge (Lewis et al., 2020). RAG is especially useful in scenarios where LLMs must reason using up-to-date or highly detailed documentation.

In a typical RAG system, when a query is given, the model first retrieves relevant documents or snippets from an external Knowledge Base (KB). These retrieved pieces of information from this KB are then used along with the original query as inputs to the model. This two-step process enhances the model's ability to produce better response, that is factual, contextually grounded and domain-specific. (Tural et al., 2024). RAG changes the LLM from a static knowledge holder into an active "reader," capable of dynamically accessing up-to-date or specialized information without needing expensive retraining. Figure 2 shows the working of a RAG system.

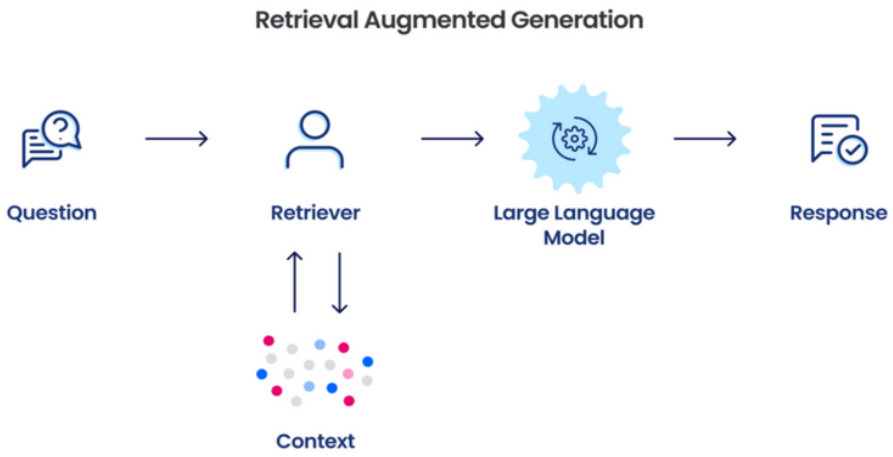

Figure 2: RAG System

This feature is particularly crucial when LLMs are applied to technical tasks where domain knowledge is very important and it is detailed, complex and constantly evolving. In programming contexts like C development with GLib and GStreamer frameworks, memory management practices come with very specific patterns. It includes reference counting, ownership transfers through callbacks, dynamic memory allocation for signals and event-driven behaviors. General LLMs may not be having that much specific details. So, without external assistance, an LLM might hallucinate incorrect memory handling suggestions, leading to unreliable results (Y. Gao et al., 2025). RAG directly addresses this challenge. This is done by allowing the model to query external documentation repositories. The repositories include GLib's memory management guidelines, GStreamer's object lifecycle rules or known bug reports. The LLM can access accurate technical information exactly when needed. Instead of

memorizing every possible corner case during training, the model can look up the relevant practice dynamically (Wei et al., 2025). This improves the model's ability to reason about framework-specific code scenarios, such as detecting missing g_object_unref() calls in GStreamer pipelines or mismanaged GList memory in GLib structures.

Recent research further highlights the advantage of RAG for specialized domains. RAG performs better than a full fine-tuning approach. This is significant when dealing with rare or long-tail knowledge that would be less used in standard model pretraining (Soudani et al., 2024). While RAG enables LLM to access external knowledge during inference, it lacks the ability to internalize task-specific patterns through parameter updates. In memory leak detection tasks, especially those involving frameworks like GLib and GStreamer, consistent understanding of even minor memory management rules is important. It includes areas such as ownership transfer or signal-based lifecycles which requires deeper model adaptation. RAG may retrieve relevant documentation, but without fine-tuning, the model might still fail to apply that knowledge consistently across varied contexts (Tvarožek & Haffner, 2025). This limitation makes RAG alone insufficient, highlighting the need to combine it with PEFT like LoRA for reliable and domain-aware performance.

The experiments demonstrated that combining retrieval with PEFT techniques, like LoRA, preserved reasoning quality. Along with that it reduces training and inference costs. Similarly, it demonstrated the success of integrating RAG with LoRA in educational and technical Question Answering (QA), where domain knowledge had to be dynamically retrieved across multiple documents (Alawwad et al., 2025). RAG divide different query types into classes such as explicit facts, interpretable rationales and hidden rationales. In these types, it was proven that technical tasks often demand higher levels of external knowledge integration (Zhao et al., 2024). Identifying a memory leak in C code typically falls under the interpretable rationale category. So, it is not just about retrieving facts, but about understanding underlying lifecycle rules across code snippets and documentation.

Retrieval is not just useful, but it becomes essential. In traditional fine-tuned models without RAG, hallucinations may occur when the model overconfidently generates information based on incomplete or incorrect internal patterns. Meantime a RAG system can pull in correct descriptions like the ownership semantics of g_list_free_full() in GLib and ground its response in verifiable external facts. This in turn minimize hallucination risks and improving developer trust (Arslan et al., 2024). Our model builds upon these insights by combining RAG with an uncertainty-driven LoRA adaptation mechanism. At inference time, a user code snippet is used for retrieval of relevant GLib/GStreamer documentation using sentence similarity search and filtering. The retrieved context is then presented to the model along with the query. LoRA dynamically adjusts its influence based on the model's confidence. This hybrid strategy ensures that the model not only remembers general programming semantics but also reasons over real-world, domain-specific practices efficiently.

2.6 Related Work on LLMs for Code Understanding

LLMs can deal with code understanding tasks across different programming languages. Models such as Codex, CodeBERT, StarCoder, and CodeLlama have been widely applied in bug detection, code summarization and static analysis. These models can identify common patterns and produce semantically valid responses. However, their performance in understanding low-level code such as C, especially in tasks involving memory management, is still limited. Many of these models tend to hallucinate when dealing with complex systems programming concepts like pointer aliasing, ownership transfer and dynamic memory behavior.

Fang et al. (2024) conducted an experiment on how well LLMs perform in software security-related code understanding. Their study showed that although models like GPT-4 and Claude performed better than others in general reasoning tasks, they still fail when asked to interpret complicated C functions or

system-level memory behaviors. The study concluded that LLMs often lacks the precision required for detailed code reasoning for detecting unsafe or leak-prone behaviors in security-critical applications. Liu et al. (2023) further explored the correctness of LLM-generated code through the introduction of EvalPlus. Their experiments demonstrated that even the most advanced models could generate functionally incorrect or unsafe C code. This reveals a major risk when using LLMs for automated code generation or static analysis for safety-sensitive tasks like memory leak detection. Jung et al. (2024) presented a detailed study to examine whether LLMs can understand program syntax and semantics. Their evaluation included tasks such as Abstract Syntax Tree (AST) generation, dataflow reasoning and pointer alias analysis. Results showed that while LLMs were good at syntax-level tasks, their performance dropped significantly in semantic tasks. That is particularly in pointer handling and taint analysis. This weakness is critical in the context of detecting memory leaks in real-world C codebases.

LLM-Assisted Memory Detection (LAMeD) Shemetova et al. (2025) proposed a method where LLMs can be used to automatically generate annotations to help static analyzers detect memory leaks. This method improved the performance of traditional tools like Infer and CodeQL when applied to projects like libxml2 and curl. However, the approach still relied on external analyzers and rule-based backends for final leak detection and it lacked dynamic model adaptation or integration of framework-specific context during inference.

Even though these methods have made notable contributions in code understanding and general bug detection, none of them address the full spectrum of memory safety issues found in GLib or GStreamer. These libraries involve reference counting, signal-based lifecycle management and delayed ownership transfers, which cannot be easily interpreted by general-purpose LLMs. Existing solutions either depend on symbolic engines, external rule-based analyzers or only partial integration of LLM reasoning.

2.7 Related Works on Combining RAG and Fine-Tuning (LoRA)

Recent works demonstrate the effectiveness of combining RAG with fine-tuning techniques like LoRA. For instance, Alawwad et al. (2025) introduced a fine-tuned RAG system using quantized models for low-resource deployment. On the other hand, Soudani et al. (2024) evaluated reliability optimization in rank-adaptive settings. These works support the integration of dynamic context retrieval with LoRA to build task-aware and explainable AI systems.

RAFT (Zhang et al., 2024) is another significant study that combines RAG with LoRA for educational reasoning. It shows strong results on complex QA tasks. Their architecture, which integrates adaptive prompt construction with document-grounded reasoning, offers a promising foundation for technical domains like code analysis. Another related development, RoRA ((J. Liu et al., 2025), introduces a reliability-optimized version of LoRA, emphasizing stable task performance even under noisy inputs. Such advancements are directly relevant to memory leak detection.

Ibrahim et al. (2024) proposed a multilingual QA system using LoRA-tuned LLMs enhanced by a hybrid dual-retrieval RAG module. Their contributions include optimizing inference performance using a two-stage retriever. It shows similar level of performance with full fine-tuning on LLMs. The system focused on multi-language general QA. But the architecture highlights the advantages of combining lightweight parameter-efficient tuning with contextual augmentation for better responses.

Kim et al. (2024) presented a scalable architecture that combines the retrieval-augmented instruction tuning with PEFT strategies. Their work demonstrates that low-resource fine-tuned models can be paired with adaptive retrieval schemes. The resulted architecture can achieve comparable reasoning depth to much larger systems. Such findings reinforce the thesis design choice of making a combined domain-specific LoRA-tuned weights with targeted memory management documentation to maintain accurate results even with limited training data.

Han et al. (2024) explored LoRA and RAG for biomedical applications. It demonstrates improvements in the correctness of the response and reduction in hallucinations. This is achieved through multi-level retrieval and fine-grained control over prompt structure. Even though the target domain is different, their emphasis on contextual specificity and retrieval-guided inference supports the thesis strategy of adding structured GLib/GStreamer documentation at inference time.

Even though these methods are introduced for combining the RAG and LoRA, the main drawback of those methods where high computational cost and requirement of large datasets. Also, none of these innovations, from this relatively few combined RAG + LoRA approaches points to static code analysis or memory leak detection. This thesis addresses that gap by developing a hybrid LLM-based detection system created to framework-heavy C codebases. It uses LoRA for efficient domain adaptation and RAG for injecting context-aware memory management documentation at inference time without any retraining. The system further integrates an uncertainty scoring mechanism to dynamically scale LoRA influence, enabling more accurate predictions across varied inference conditions.

By combining these techniques, the proposed model advances the State-of-the-Art (SOTA) in explainable, adaptable and resource-efficient memory leak detection.

2.8 Summary of Research Gaps

From the previous discussions, it becomes clear that significant limitations remain even after many approaches have been proposed for memory leak detection and code understanding. This comes particularly in the context of low-level programming with domain-specific libraries such as GLib and GStreamer.

Traditional memory analysis tools in both static and dynamic, are useful but are not well-suited for complex, modular C codebases. Static tools like Clang Static Analyzer and Cppcheck struggle with customized control flows and advanced memory ownership conventions. This often produces FPs or missing subtle leaks. Dynamic tools such as Valgrind and Address Sanitizer offer high accuracy but impose huge runtime overhead. This makes them impractical for use in CI pipelines or performance-sensitive systems. ML and DL based classifiers have also been applied to this problem. They consider memory leak detection as a binary classification task. However, these models rely heavily on hand-crafted features. They often fail to capture deeper semantic behaviors, such as pointer aliasing or resource lifecycles across multiple functions. Moreover, they lack flexibility and require retraining when applied to new projects, stacks or memory models.

LLMs like Codex, CodeBERT and DeepSeek Coder have brought significant improvements in general code intelligence tasks such as code completion, bug detection and documentation generation. However, they are not inherently capable of interpreting domain-specific memory management behaviors. These models are powerful, but they often hallucinate or overlook important patterns related to stack-specific APIs like those found in GLib and GStreamer. Some efforts have attempted to improve LLMs through fine-tuning. But the problem was that the full fine-tuning is computationally expensive and difficult to scale. More recently, lightweight fine-tuning techniques such as LoRA have shown promise in making LLMs task-aware without retraining the entire model. Similarly. RAG helps models pull in external knowledge at inference time. This reduced the dependency on static pre-training. Despite their individual strengths, these techniques are often used in isolation and do not adapt dynamically to the complexity or uncertainty of the input.

These insights reveal a clear set of research gaps. The first one is the lack of systems that can perform accurate memory leak detection in domain-specific, low-level C programs using libraries such as GLib and GStreamer. Secondly existing models and tools either lack contextual awareness, rely on rigid rule sets or require extensive retraining to adapt to new use cases. The third one is that general-purpose LLMs does not effectively handle framework-specific memory semantics and often generate incorrect or

unexplainable suggestions. The last one is that most current solutions do not integrate knowledge retrieval and fine-tuning in a way that adapts dynamically to prediction uncertainty or task complexity.

This thesis aims to address these limitations by introducing a hybrid system that combines RAG with PEFT method LoRA. The system is guided by an uncertainty scoring mechanism. This adjusts the model's behavior based on confidence. Also, this allows model to operate more reliably across a range of inference scenarios. By using predictions in external documentation and dynamically adapting the influence of fine-tuning, the proposed approach seeks to offer a lightweight, explainable and context-aware solution to memory leak detection in real-world systems programming.

3. Methodology

This section discusses the methodology adopted for designing, implementing and evaluating the proposed memory leak detection system. The methodology follows an experimental research approach. It aims the goal of constructing a hybrid system that uses LLMs, domain-specific knowledge retrieval, and adaptive LoRA inference mechanisms to detect memory leaks in C codebases that uses GLib and GStreamer. The methodology is divided into five key sections: research design, dataset development, system implementation and evaluation approach.

3.1 Research Design

This study adopts an experimental research methodology to design (Sjoeberg et al., 2005), implement, and evaluate a hybrid system for memory leak detection in C programs that use GLib and GStreamer libraries. The approach is structured around a comparative evaluation of five model configurations under controlled conditions to assess the effectiveness of combining PEFT (LoRA), RAG and uncertainty-based inference modulation.

The proposed system addresses a well-defined problem. That is the limitations of existing static and dynamic analysis tools in detecting memory leaks in programs using complex C libraries. Instead of relying solely on design iteration, this research focuses on systematic experimentation to compare model variants and isolate the contribution of each architectural component.

The methodology is based on a strictly controlled experimental setup, wherein a fixed model architecture, standardized test environment and a common evaluation dataset are consistently used across all experiments. This ensures that observed differences in performance can be attributed to model design rather than external factors. Furthermore, all model variants are evaluated using consistent performance metrics to enable fair and objective comparison. Standardized prompt formatting is applied uniformly to eliminate formatting bias and maintain evaluation integrity across different configurations. This design enables reproducible results and a rigorous comparison of performance across model variants.

3.2 Experimental Design

The experimental design aims to evaluate the performance of the proposed hybrid system by comparing it with several baseline and partial variants. The core focus is to isolate and assess the impact of LoRA fine-tuning, RAG-based knowledge augmentation and uncertainty-aware inference on the task of memory leak detection.

3.2.1 Variables

- **Independent Variable**: The independent variable is the model configuration, with five variants tested

    1. Base model: DeepSeek Coder 1.3B Instruct without any adaptation
    2. LoRA-only model: Fine-tuned on the memory leak dataset, without retrieval
    3. RAG- Uses external documentation retrieval without fine-tuning.
    4. LoRA + RAG: Combines fine-tuning and retrieval, but no uncertainty modulation.
    5. Hybrid model Integrates LoRA, RAG and uncertainty-based inference.
- **Dependent Variables**: The dependent variables are the classification performance metrics, including:

- Accuracy
- Precision
- Recall
- F1 score

These metrics evaluate the correctness and reliability of each model in classifying C code as memory-leaky or non-leaky.

- **Controlled Variables**: To maintain consistency, the following aspects are kept constant across all experiments

    - Model architecture: *DeepSeek Coder 1.3B Instruct*
    - Inference hardware: *Local machine with 25GB* RAM
    - Prompt formatting: *Standardized for all models*
    - Test dataset: *Same 100 balanced samples for all variants*

These controls ensure that observed differences in performance arise solely from the variations in model architecture.

3.2.2 Dataset Construction

Two datasets were developed to support the training and inference workflows of the proposed system. This includes a labeled code sample dataset for fine-tuning with evaluation and a structured documentation corpus for retrieval.

Fine-tuning and Evaluation Dataset:

This dataset contains 2,500 labeled C code samples, equally distributed between memory-leaky and non-leaky examples. Each sample is associated with:

- Code snippet
- Binary label (leak / non_leak)
- Explanation for the label
- Fixed version (if applicable)
- Expected output
- Retrieved documentation segment

The dataset includes both synthetic and real-world patterns focused on GLib and GStreamer functions such as g_malloc, g_object_unref, g_list_append, gst_element_set_state and g_array_free. This dataset is used for both LoRA fine-tuning and model evaluation.

RAG Knowledge Base:

This is a structured, version-aware KB built from official GLib and GStreamer documentation. It contains over 2,000 entries, each with focuses on functions with explicit memory ownership or deallocation behavior. Each entry includes function name and memory behavior descriptions. The description includes Parameter semantics, memory behavior patterns, ownership and deallocation details. Each entry is embedded using SentenceTransformers and indexed using Facebook AI Similarity Search (FAISS). During inference, relevant documentation is retrieved based on the functions detected in the input code and appended to the prompt to enable contextual reasoning by the model. Together, these datasets ensure a comprehensive setup for both training and evaluation, allowing the model to

learn from diverse memory usage patterns and reason with grounded domain knowledge during inference.

3.2.3 Experimental Protocol

Data Splitting: The fine-tuning data set of 2,500-samples were split into 80% for training and 20% for validation.

Training Procedure: LoRA adapters were trained for 2 epochs with batch size of 4 and learning rate of 2e-4. PEFT is used to maintain computational efficiency on low-resource hardware.

Testing: All model variants were evaluated on a common, unseen test set of 100 samples. This set was balanced and included diverse memory management patterns.

Analysis: Evaluation metrics were recorded and compared across all variants. Model outputs were analyzed using a simple rule-based text classifier: if the output contains phrases like *"This code has a memory leak..."*, it is classified as leak; if it says, *"No memory leak..."*, it is classified as non_leak. This uniform approach was applied across all models to ensure fairness.

Explanation Evaluation: In addition to classification accuracy, explanation quality was manually inspected to verify if the justification segments aligned with the correct reasoning. However, justification quality was not scored quantitatively and is discussed qualitatively in the discussion chapter.

3.3 System Implementation

The proposed memory leak detection system is composed of three integrated components designed to work in a balanced way to address the research objectives.

1. LoRA Fine-tuning: The base model (DeepSeek Coder 1.3B Instruct) is fine-tuned using LoRA. The goal is to enable the model to detect memory leaks in C code, particularly involving GLib and GStreamer APIs. This fine-tuning allows task-specific learning while maintaining computational efficiency.
2. RAG: A semantic retrieval pipeline is built using a KB constructed from GLib and GStreamer documentation. Each function is stored as a structured record in JavaScript Object Notation (JSON) format and embedded using SentenceTransformers. These embeddings are indexed with FAISS to support real-time retrieval of memory-related information based on the input code.
3. Uncertainty-Based Adaptive Inference: This component serves as the core mechanism that integrates LoRA fine-tuning with the RAG module. For each input, the system computes an uncertainty score which is based on cosine similarity and function coverage. This score reflects the reliability of retrieved context and modulates the influence of the LoRA-adapted model during inference. When uncertainty is low, that is the retrieval is accurate and relevant, the model relies more on RAG context. Conversely, when uncertainty is high, the system shifts more weight to LoRA's learned knowledge. This dynamic balancing enables the model to combine the strengths of both fine-tuning and retrieval, making the integration accurate and context aware.

3.4. Evaluation Approach

The proposed method integrates LoRA-based fine-tuning, RAG through an uncertainty-aware inference into a hybrid architecture for memory leak detection in C programs. This integration provides a direct answer to RQ1 by demonstrating how LoRA and RAG can be effectively combined through adaptive inference control. The evaluation in this section focuses on RQ2 **:** How do accuracy, recall and false positive rates vary when applying a task-specific LLM to memory leak detection in GLib and GStreamer-based C code, particularly one enhanced through the combination of knowledge augmentation and

PEFT?. The results provide both quantitative and qualitative evidence to assess the effectiveness and interpretability of the proposed method. This study adopts a comparative experimental design to assess how different model configurations perform in classifying C code snippets as memory-leaky or non-leaky. The evaluation isolates the impact of each system component by testing five distinct model variants under controlled conditions:

- Base Model: The unmodified DeepSeek Coder 1.3B Instruct.
- LoRA-Only: Fine-tuned with the labeled memory leak dataset; no external retrieval.
- RAG-Only: Uses GLib/GStreamer documentation retrieval without task-specific fine-tuning.
- LoRA + RAG (Non-Adaptive): Combines fine-tuning and retrieval, but no dynamic inference modulation.
- Hybrid Model: Integrates LoRA, RAG and adaptive inference modulation via uncertainty scoring.

All models were evaluated using the same unseen test dataset of 100 code samples. This dataset was curated to maintain class balance and include a diverse set of GLib and GStreamer function usage patterns. The sample size reflects the constraints of token limits in LLMs and available hardware resources, while ensuring representative coverage of real-world memory management scenarios.

Evaluation Metrics and Procedure

Each model was assessed using four standard classification metrics commonly used in binary classification tasks (Sokolova & Lapalme, 2009).

- Accuracy: Overall proportion of correct predictions.
- Precision: Proportion of predicted leaks that are actual leaks.
- Recall: Proportion of actual leaks that were correctly identified.
- F1 Score: Harmonic mean of precision and recall.

In addition to these metrics, support is reported for each class. Support refers to the number of actual instances of each class (leak and non-leak) in the test dataset and helps contextualize the evaluation metrics, particularly in cases of class imbalance.

To maintain evaluation consistency, a uniform output parsing rule was applied across all model outputs. The LLM-generated responses were classified based on the presence of specific phrases. Phrases such as "This code has a memory leak..." were interpreted as leak. Phrases such as "No memory leak is found..." were interpreted as non_leak. This rule-based text classification method ensured that subjective interpretation was minimized and that model outputs were mapped to binary predictions in a repeatable, fair manner.

Explanation Evaluation

In addition to classification performance, the system was also evaluated for its explanatory capability, a key feature for developer-facing tools. Each model output includes a short, natural language justification describing why the code was considered leaky or not. These explanations were manually reviewed for consistency, correctness and alignment with the retrieved context and ground truth. Although no quantitative scoring was applied, this qualitative inspection confirmed that the hybrid model generated explanations that were not only accurate but also, grounded in GLib/GStreamer memory semantics. This additional layer of validation reinforces the claim that the system offers interpretable and reliable outputs, supporting practical use in software development workflows.

4. The System

4.1. Overview of Proposed System

This thesis proposes a hybrid memory leak detection system that uses the strengths of LLMs and addressing their limitations in domain-specific reasoning. The system is designed to analyze C code statically and predict whether a given segment contains memory leaks. It focusses on applications that use GLib and GStreamer libraries. The architecture combines two advanced techniques: PEFT using LoRA and RAG, all coordinated through an uncertainty-aware adaptation mechanism. As mentioned in the background section for the memory leak detection in GLib and Gstreamer, the LLM is used for the leak detections as it can analyze the code semantics and patterns to identify the leak. For the general purpose LLM to be enabled for this task, two challenges must be addressed. One is, the LLM must be aware about the GLib and Gstreamer libraries, their memory usage and all the information related to memory management. The other one is, LLM must be aware about the memory leakage patterns and must be trained on the leak and non-leak classification. A single technique is not able to cover both these challenges together. So, the advantage of RAG and LoRA in a combined way can solve this issue.

The system starts by receiving a static C code snippet as input. This code is analyzed in two key stages. First, the relevant GLib or GStreamer function names are extracted. Then it is used to query a pre-built external knowledge base containing structured documentation entries. This step is handled using RAG. Here the retrieved information about the usage, ownership and lifecycle of these functions is appended to the query before it is passed to the LLM. This helps the model to make decisions based on actual library behavior rather than relying purely on internal memorization (Wei et al., 2025).

The LLM used in the system is fine-tuned using LoRA. So, it modifies only a small subset of its parameters. This makes the system efficient to train and deploy even on hardware with limited memory resources (Hu et al., 2021). The fine-tuning data consists of labeled examples of memory-leaky and non-leaky C code. This task-specific training allows the model to better recognize common patterns of misuse in dynamic allocation, pointer handling and library-specific cleanup functions, such as g_object_unref() or g_list_free_full().

When these two methods are combined, one of the main challenges will be contribution of each module. If the information retrieved by RAG is not accurate or irrelevant, it will adversely affect the output response. That is even if the LoRA fine-tuned LLM can make correct decision based on the training given, an incorrect or irrelevant information from RAG can make the system to hallucinate. So, confidence of the information retrieved from RAG is very important. Therefore, after retrieving context and preparing the prompt, the model calculates a confidence score based on the similarity and relevance between the input query and the retrieved documentation. This score is used to determine how strongly the LoRA layers should influence the final prediction. If confidence of the RAG information is high, minimal adaptation is applied. That is relying more on the base model. If confidence is low, that shows ambiguity or unfamiliar code. This makes the influence of the LoRA layers to be increased to bring in stronger task-specific reasoning. This dynamic tuning approach draws inspiration from similar techniques in confidence-aware retrieval systems (Soudani et al., 2024).

Figure 3 illustrates the overall flow of the system, which can be summarized in the following steps:

1. Input Handling**:** A C code snippet is submitted for analysis.
2. Function Name Extraction**:** The system identifies relevant GLib or GStreamer functions.
3. RAG Retrieval: Using the extracted names, it fetches related documentation using semantic similarity.

4. Prompt Construction: The retrieved context and the input code are combined into a structured prompt.
5. Uncertainty Scoring: The system computes how confident the model is in its prediction based on cosine similarity metrics.
6. Adaptive Inference: Based on uncertainty, the model adjusts how much the LoRA layers influence the output.
7. Prediction Output: The model returns whether the code is memory-leaky or safe, along with an explanation.

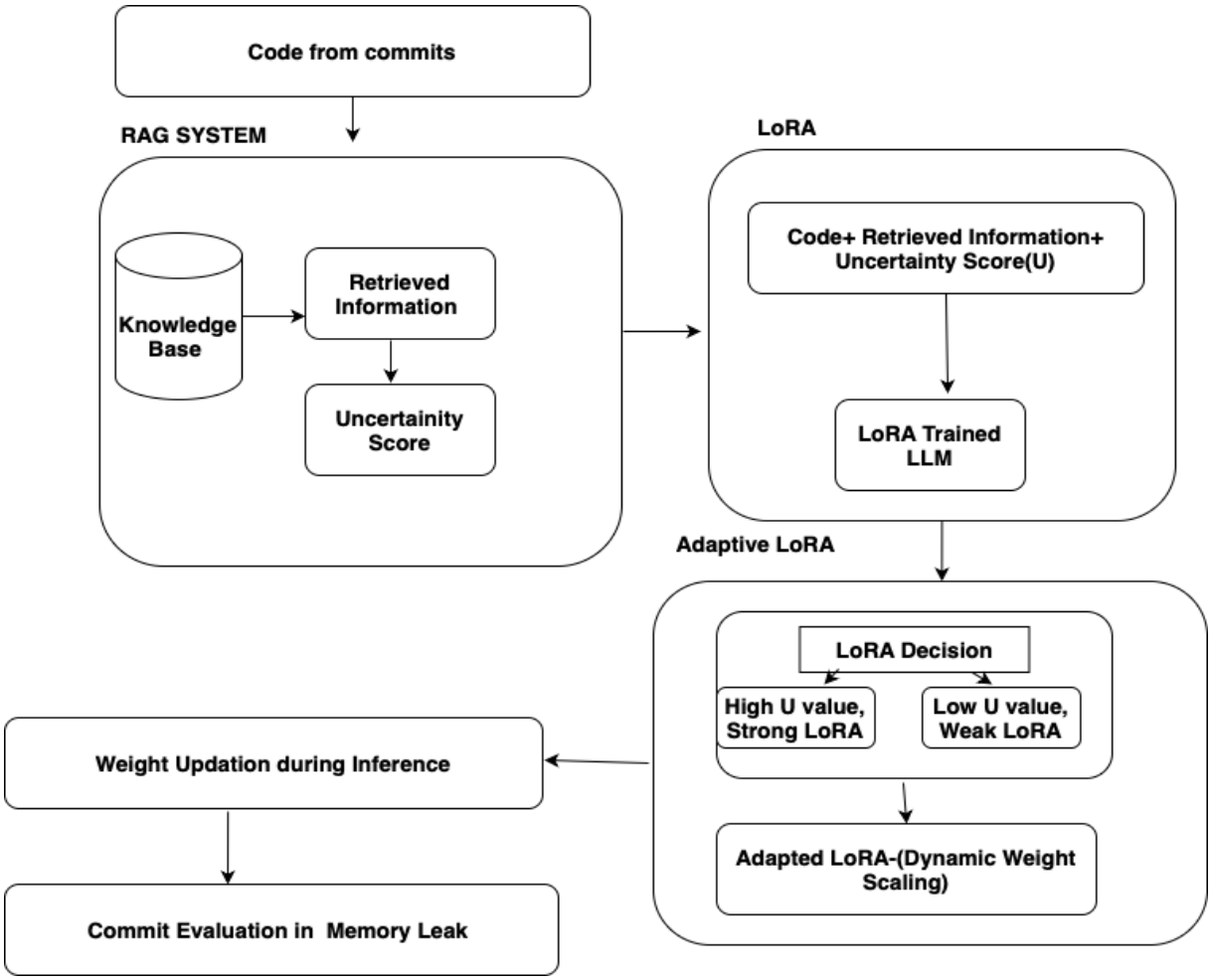

Figure 3: Proposed Architecture of Combining RAG with LoRA

This modular architecture is designed to be efficient, scalable and explainable. It avoids the performance costs of dynamic analysis tools like Valgrind. Sametime, it addresses the context-awareness issues found in traditional static tools. Furthermore, here the external knowledge is integrated only when needed. So, this system reduces hallucination risks and improves trust in predictions. This is the crucial factor in developer-facing tools (Alawwad et al., 2025)(Arslan et al., 2024).

4.2 LoRA Fine-Tuning

This section explains the process of fine-tuning the base language model using LoRA. It was selected because it enables large models to be adapted to specific tasks without the need to retrain all their internal parameters. Along with that, it saves both time and computational resources (Hu et al., 2021).

In this scenario, LoRA is used to fine-tune the model to detect memory leaks in C programs, especially those using frameworks like GLib and GStreamer.

For this work, we selected the DeepSeek Coder 1.3B Instruct model as the base LLM due to its lightweight architecture, open-source availability and strong performance on code-related benchmarks. Its instruction-following capabilities and smaller parameter size (1.3B) make it suitable for fine-tuning using LoRA on limited hardware.

4.2.1 Dataset Design and Structure

The model is trained on a task specific dataset of labeled C code examples. The dataset consists of a mix of synthetically created samples and real-world C code extracted from currently deployed company-internal programs. These samples were curated specifically for this study and are not publicly available. Each example is annotated with information needed to help the model to understand how a memory leak looks like and how it can be corrected. The dataset consists of both leaky and non-leaky C code samples. All of them involve dynamic memory handling through GLib/GStreamer functions such as g_array_new, g_list_append, g_object_unref etc.

Each training instance is structured with the following fields:

- Prompt_Code: The original C code snippet, usually a function or a small program segment.
- retrieved_context: Documentation or technical notes retrieved from GLib or GStreamer manuals that explain proper memory handling for the functions used.
- label: The classification target—either "leak" or "non_leak".
- explanation: A plain-text description that explains why the code is considered leaky or safe.
- Expected_output: An expected output response from the LLM
- fixed_code: A corrected version of the code that resolves any detected memory leak.


```
{
  "id": 1,
  "Prompt_Code": "#include <stdlib.h>\n#include <stdio.h>\n\nvoid leak() {\n
int *ptr = (int *)malloc(100);\n    // Memory is allocated but never
freed\n}\n\nint main() {\n    leak();\n    return 0;\n}",
  "retrieved_context": "Memory allocated using malloc() should be freed using
free().",
  "label": "leak",
  "explanation": "The function leak() allocates memory but does not free it,
leading to a memory leak.",
  "expected_output": "To fix this, free the allocated memory before exiting
the function.",
  "fixed_code": "#include <stdlib.h>\n#include <stdio.h>\n\nvoid leak() {\n
int *ptr = (int *)malloc(100);\n    free(ptr); // ✅ Fix: Freeing allocated
memory\n}\n\nint main() {\n    leak();\n    return 0;\n}"
},
```


Figure 4: Training Instance sample

Figure 4 shows an example of training document sample. This structure allows the model to learn the syntax along with the semantics for the safe memory usage. This makes the model suitable for real-world leak detection scenarios.

4.2.2 Tokenization and Prompt Construction

To prepare the data for training, each example is tokenized using the tokenizer. A prompt template is applied to every sample to give the model a consistent format. This helps the model to better understand the task. The prompt includes:

f"### Memory Leak Detection Task:\n"
f"### C Code:\n{p}\n"
f"### Retrieved Context:\n{ctx}\n"
f"### Explanation:\n{exp}\n"
f"### Fixed Code:\n{fix}\n"

This design helps the model to see each code sample in context, along with explanations and expected solutions. Token sequences are padded and truncated to a fixed length of 256 tokens for training stability. Special attention is given to masking padded tokens to ensure they do not affect the loss computation.

4.2.3 Model Selection

The base model used in this thesis is DeepSeek Coder 1.3B, a transformer-based LLM. The model was selected due to its optimal trade-off between model size, performance and practical deployability (Guo et al., 2024). With 1.3 billion parameters, the model provides strong code understanding and generation capabilities. Also, it remains lightweight enough to fine-tune and run efficiently on local hardware with limited memory. Its open-source availability, instruction-following architecture and proven performance on coding benchmarks made it a suitable base for PEFT using LoRA. Furthermore, its compact design aligns well with the system's aim of enabling resource-constrained static analysis tools without relying on large-scale cloud infrastructure.

In addition to being practical and efficient, DeepSeek Coder 1.3B was a great fit for this study. It understands memory-related behavior in C code and follow structured prompts to make clear leak or non-leak decisions. Since it is trained to follow instructions, the model can reason over both code and retrieved documentation, which works well with the RAG-based setup in this system. Its smaller size also makes it possible to apply adaptive LoRA scaling without needing high-end hardware. This helps the system stay fast and easy to interpret, even when the retrieved information is incomplete. While larger models might perform slightly better, DeepSeek offers a good balance between accuracy, flexibility and resource efficiency.

4.3 Retrieval-Augmented Generation

The proposed system enhances the model with RAG to introduce domain-specific knowledge during inference. This is essential because the LoRA fine-tuned LLM may not have in-depth understanding of memory management details specific to GLib and GStreamer. RAG addresses this by dynamically injecting relevant documentation about memory ownership, allocation and deallocation into the model's prompt at inference time.

4.3.1 Knowledge Base Preparation and Chunking

The RAG KB used in this system consists of structured documentation entries. Each sample containing a GLib or GStreamer function name and its corresponding description. This includes how the function manages memory, its ownership semantics and best practices for usage. It is not splitting the documents arbitrarily, but a function-level chunking strategy is used. Each function and its related documentation are stored as a single document unit to maintain clarity and retrieval precision.

The documents are wrapped using LangChain's Document class and embedded using the microsoft/codebert-base SentenceTransformer model. This makes sure that the embedding captures both the technical semantics and functional purpose of each entry. These embeddings are stored in a FAISS index for fast retrieval during inference.

4.3.2 Semantic Querying and Prompt Construction

When a C code snippet is submitted, the system first extracts GLib or GStreamer function calls using regular expressions. These extracted functions serve as semantic queries. For each function name, the top-k most similar documentation entries are retrieved from the FAISS index. If no function names are identified, the full code snippet is used as the query instead. This approach ensures that retrieval is highly targeted. Also, this reduces noise by focusing only on functions relevant to memory handling. The retrieved entries form the contextual knowledge used to help the model to reason about code behavior during inference. This retrieved context is combined with the input C code to form a structured prompt. This prompt instructs the model to analyze memory usage and provide a binary leak or non-leak decision, along with justification. The sample prompt structure is:

prompt = f""" Can you analize the following code change, list all glib and gstreamer related functions and check whether returned values must be freed or not. Are there any memory leaks related to them?

== This is the more relevant information about GStreamer related functions ==
{combined_context}

== This is the C Code change i wish you analyze ==
```c
{code_snippet}
```"""

4.3.3 Integration with Cosine Similarity and FAISS

The quality of each retrieved document is evaluated. It is calculated by the cosine similarity between the embedding of the input code snippet and the embeddings of the retrieved documents. The highest similarity score reflects how well the retrieved information aligns with the input. This similarity score is then combined with two additional metrics. That is missing_from_retrieved and missing_from_KB. This is used to compute an overall uncertainty score. The final uncertainty value is a weighted sum of these three components. This determines the model's confidence in the relevance and completeness of the retrieved context. The FAISS vector store helps for efficient similarity search. It works well even across many documentation chunks. The system combines semantic search with task-specific prompting. Then the RAG system provides in-time knowledge addition. This design enables the LLM to analyze more accurately about domain-specific code without retraining or modifying the base model weights.

4.4 Adaptive Integration via Uncertainty Scoring

One of the unique components of the proposed system is its ability to adaptively control the influence of the LoRA fine-tuning module during inference. This is achieved by an uncertainty scoring mechanism. It evaluates the relevance and completeness of the documentation retrieved by RAG, is in the context of the given C code snippet. In this method, the retrieved information and the fine-tuned model are not equally treated in every scenario. The system dynamically adjusts how much it should trust each source of knowledge based on confidence. This mechanism allows the system to handle the uncertainty more effectively. If the retrieved documentation is incomplete or mismatched with the input code, the system can depend more on what it has learned through fine-tuning. When the context retrieved is strong and

aligns well with the code, the model can rely more on the external documentation. This in turn reduce the risk of overfitting or hallucination.

4.4.1 Motivation for Adaptivity

For the LLM to be task specific, LoRA provides pattern adaptation and RAG introduces external domain knowledge. But most important thing is, how their combination is going to work effectively. For better adaption and response, both must be carefully balanced. If RAG retrieves irrelevant documentation and relying on it too heavily can confuse the model. So even if LoRA has already learned the correct memory management pattern, it will work according to the retrieved information. On the other hand, when RAG provides strong evidence aligned with the code, fine-tuning should step back and let the model make use of it. To handle this trade-off, we introduce a scalar uncertainty score $U \in [0,1]$. This score acts as a knob that controls how strongly the LoRA layers influence the final prediction. A low uncertainty means the system is confident in the retrieved information and LoRA's influence can be reduced. A high uncertainty means the system is unsure about the retrieved knowledge and should rely more on its task-specific training via LoRA.

4.4.2 Components of the Uncertainty Score

The uncertainty score is computed using three measurable components:

- Similarity score ($1-\text{max\_similarity}$): This shows the similarity and semantic alignment of the code snippet with the retrieved documentation. Cosine similarity is used between the embedding of the input code and each retrieved chunk. A low similarity implies that the retrieved content might not be relevant.
- Missing from retrieved fraction: This computes the ratio of GLib or GStreamer functions found in the code that are not present in the retrieved context. If many used functions are missing, the retrieved information is incomplete.
- Missing from knowledge base fraction: These measures whether the functions used in the input code exist at all in the entire RAG KB. If not, it indicates a critical gap in available information.

Each of these components is weighted to reflect their relative importance. The final uncertainty score U is calculated as:

$$U = \alpha \cdot (1 - \text{max\_similarity}) + \beta \cdot \text{missing\_retrieved\_fraction} + \gamma \cdot \text{missing\_kb\_fraction}$$

Where the weights are defined as:

- $\alpha = 0.2$ - contextual mismatch
- $\beta = 0.35$ - impact of incomplete retrieval
- $\gamma = 0.45$ - whether the function exists at all in the KB

These values were determined based on observed behavior during experiments. More weight was given to the KB completeness. This indicate a complete absence of a function, indicating that the system is working completely in unfamiliar scenarios.

4.4.3 Integration with LoRA Inference

Once the uncertainty score U is computed, it is used to scale the influence of LoRA layers dynamically during inference. This is implemented using forward hooks on the LoRA adapter layers, which scale the low-rank updates proportionally to U.

Let:

- h be the hidden state input to a transformer block
- W be the frozen base weight
- ΔW=BA be the LoRA low-rank weight update
- U∈ [0,1] be the adaptive uncertainty-based scaling factor
- α be the original LoRA scaling (typically r)

The adapted output becomes:

$$h' = (W + U \cdot \alpha \, .BA) \cdot h$$

This formula shows that LoRA's influence is softly blended into the output. It does not fully turn on or off. The behavior varies depending on the context:

- If U≈0 → minimal LoRA influence; rely on retrieved knowledge and base model
- If U≈1→ full LoRA adaptation; trust task-specific training more than retrieval

This dynamic modulation makes sure that the model can self-correct when faced with unfamiliar, low-confidence contexts and use external help when it is relevant and reliable.

4.4.4 Benefits of the Adaptive Approach

This adaptive brings several practical advantages. It improves generalization. So, the model does not overfit to noisy or sparse RAG context. Another advantage is reduced hallucination risk**.** The weak or irrelevant documentation does not overpower the learned LoRA patterns. The explainability and efficiency is also a significant advantage of this approach. The system's behavior is traceable through uncertainty metrics and no retraining is required.

This uncertainty-aware fusion of RAG and LoRA enables a highly flexible, context-sensitive architecture suitable for static code analysis tasks like memory leak detection. This is useful especially in environments like GLib and GStreamer where both semantic reasoning and documentation grounding are essential.

4.5 Prompt Structuring and Model Inference

The performance of the proposed memory leak detection system depends mainly on how information is presented to the language model during inference. Prompt format plays a key role in shaping the model's understanding of the code context. It guides toward producing structured, accurate and explainable outputs.

The model receives both the input C code and relevant documentation from the RAG retriever. The way these are combined into a prompt must be consistent, informative and aligned with the model's capabilities. The prompts used in this system are carefully designed to make proper instructions and it includes all necessary technical context for the model to decide.

4.5.1 Prompt Structure

Each prompt follows a structured, instructional format. It includes:

- A clear task-definition
- The input C code
- The retrieved documentation from GLib/GStreamer

- A specific question asking whether a memory leak exists
- Additional guidelines to help the model focus on relevant memory handling rules.

A typical prompt used during inference looks like figure 5.

```
### Memory Leak Detection Task:

### C Code:
<Insert C code snippet here>

### Retrieved GLib Context:
<Insert relevant documentation for GLib/GStreamer functions>

### Question:
Does this C program have any memory leak?

### Instructions:
- Analyze the use of dynamic memory functions.
- Use the functions mentioned in the Retrieved Context to evaluate the code.
- If the code dynamically allocates memory (e.g., malloc, g_new, g_array_new), check if the memory is freed properly.
- If freeing functions like free(), g_array_free(), or g_object_unref() are missing, classify it as a leak.
- If all allocations are paired with proper deallocations, classify it as non_leak.
- Provide a YES or NO followed by a brief justification.

### Format:
Leak_Status: <YES or NO>
Justification: <Brief explanation with function names and memory logic>
```

Figure 5: Prompt Structure

4.5.2 Justification-Based Output Interpretation

During inference, the model is expected to produce two things:

1. A clear leak classification (YES or NO), and
2. A short justification explaining the reason behind the decision.

This format improves explainability as shown in figure 6. It allows the predictions to be easily evaluated against the ground truth and used in a developer-facing tool for feedback.

```
🧠 Model response:
Leak_Status: YES
Justification: The consumer thread frees the popped data before calling g_async_queue_unref(), leading to a memory leak. The data is not free
d before being removed from the queue, leading to a memory leak.
### Retrieved Contexts
```

Figure 6: Output format

4.5.3 Model Execution Workflow

At runtime, the system performs the following inference steps:

1. Function Extraction**:**
   GLib or GStreamer function calls are extracted from the C code using pattern matching
2. Document Retrieval**:**
   For each function, related documentation is retrieved from the FAISS index using semantic similarity based on SentenceTransformer embeddings.
3. Uncertainty Score Calculation**:**
   The code snippet and retrieved documents are embedded and compared. Missing function checks are performed, and the uncertainty score is calculated
4. LoRA Scaling**:**
   The LoRA module's influence is scaled based on the uncertainty score. This is done via runtime hooks applied to the model's adapter layers.

5. Prompt Generation**:**
   A prompt is constructed using the input code, retrieved documentation and instructions.
6. Model Inference**:**
   The final prompt is fed into the LoRA-enhanced LLM. The model generates output tokens, which are parsed to extract the leak classification and justification.

This full pipeline ensures that each prediction is grounded in both task-specific training and up-to-date external documentation, while adaptively handling ambiguity through uncertainty scoring.

5.System Implementation Details

5.1 Environment and Tools

The implementation of the proposed memory leak detection system is made using a modular pipeline. It integrates fine-tuned language modeling, document retrieval and uncertainty-guided reasoning. All development and experimentation were conducted on a local machine equipped with 25 GB of system RAM and Apple's M1 chip. This ensures that the pipeline could be deployed in low-resource settings without dependence on large-scale cloud infrastructure.

To implement the core functionality, the following tools and frameworks were used:

- Base Model: deepseek-ai/deepseek-coder-1.3B-instruct is the base model used. It is a compact but powerful transformer-based language model. Also, it is specifically optimized for code understanding and instruction following.
- Tokenizer: The Hugging Face AutoTokenizer corresponding to the base model was used to preprocess inputs and format prompt tokens appropriately.
- LoRA Fine-tuning Framework: LoRA fine-tuning was implemented using the PEFT library. It is provided by Hugging Face.
- Training Framework: The Hugging Face Trainer API was employed to simplify training loops and manage evaluation routines during fine-tuning.
- Document Embeddings: To embed technical documentation, the microsoft/codebert-base model from the sentence-transformers package was used. This allowed the system to generate vector representations of function descriptions, and it aligns with the semantic of the source code.
- Retrieval Layer: The FAISS library was used to build a vector index of documentation entries. It is used to retrieve the top-k closest entries for a given code query using cosine similarity.
- RAG Pipeline: LangChain was utilized to wrap document representations. It handles the interaction between code snippets, retrieval queries and embeddings.
- Uncertainty Control and Prompt Routing: Custom runtime hooks were added to the transformer's LoRA adapter layers. These hooks allow dynamic scaling of task-specific LoRA influence during inference. This is based on the uncertainty score.
- Evaluation: Metrics such as accuracy, precision and recall were computed using sklearn.metrics to evaluate and compare model variants.

The entire system was implemented in Python (version 3.10).

5.1.1. Training Procedure

Fine-tuning was conducted using the HuggingFace Trainer API on a task-specific dataset. It consists of 2,500 labeled samples of C code. This dataset was designed to cover both memory-leaky and non-leaky examples. The samples mainly focus on usage patterns in GLib and GStreamer. This size keeps a balance between model specialization and training efficiency on available hardware. The model was trained for two epochs. This was sufficient for the model to converge, given the size and diversity of the dataset. A batch size of 4 was used. This is combined with gradient accumulation steps of 2 and it results in an effective batch size of 8. This setting helped manage memory usage and it maintains a stable updates to the model parameters.

The AdamW optimizer was used for optimization. It is having a learning rate of 2e-4, which is commonly recommended for fine-tuning transformer models. A cosine learning rate scheduler was used to gradually decrease the learning rate during training. This helps the model to settle into an optimal configuration. A warm-up ratio of 0.02 was applied. This allows the learning rate to gradually increase

during the initial training steps and it will improve training stability. These hyperparameters were selected based on empirical best practices for fine-tuning language models using PEFT techniques like LoRA.
All these experiments were conducted on a local machine with 25GB RAM, showcasing the practicality of LoRA in low-resource environments. The small memory proves that the LoRA is feasible to fine-tune a billion-parameter model without needing access to high-end Graphics Processing Unit (GPUs) or cloud infrastructure.

5.2. LoRA Configuration

LoRA fine-tuning is applied to the base model DeepSeek Coder 1.3B. During fine-tuning instead of updating the entire model, LoRA works by inserting low-rank matrices into specific layers of the LLM. This is primarily the self-attention projections (q_proj, k_proj, v_proj, o_proj) and Multi-Layer Perceptron (MLP) layers (gate_proj, up_proj, down_proj).

The following hyperparameters were used for LoRA configuration:

- Rank (r): 32
- Scaling factor (α): 32
- Dropout: 5%
- Bias: None
- Task type: Causal Language Modeling

In this, RANK determines the dimensionality of LoRA matrix. A higher rank allows the model to capture more complex adaptation patterns. A value of 32 provides a good trade-off between task performance and parameter efficiency (Hu et al., 2021). The scaling factor controls the magnitude of the LoRA update. It helps to make sure that the learned updates in the LLM do not dominate the base model's activations. Using a scaling factor equal to the rank (r = α = 32) is a common practice and is used to maintain stability during training(Shuttleworth et al., 2024b). Dropout factor introduces regularization during training. This prevents overfitting to the small task-specific dataset. A moderate dropout (e.g., 0.05) helps the model to generalize better without significantly reducing learning capacity. The bias setting to 'none' disables fine-tuning of biases in the model. This reduces the number of trainable parameters of LoRA. Causal Language Modeling specifies that the model is being adapted for autoregressive generation. This aligns with the memory leak explanation and code generation task.

5.3 Knowledge Base: GLib & GStreamer JSON Structure

The RAG component of the system is based on the KB constructed from official documentation of the GLib and GStreamer libraries. This external knowledge source provides the necessary information during inference. This enables the model to analyze about memory ownership and function-specific behavior. So even if such patterns were not encountered during fine-tuning, the model could recognize the leaks.

5.3.1 Structure of the Knowledge Base

The knowledge base is stored in JSON format. In the KB, each entry corresponds to a specific memory-related function from either GLib or GStreamer. Each function is documented with key information relevant to safe usage and memory handling. The JSON structure follows a flat, dictionary-based format.

Each record contains:

{
"name": "g_array_free",
"description": "Frees the memory allocated for a GArray. The 'free_segment' parameter determines whether the actual array data should also be freed. If not freed, memory must be manually managed to prevent leaks."
}

Here the 'name' field is the exact function name like g_object_unref, g_list_free_full, etc. 'Description' is something like a human-readable summary of how the function behaves in terms of memory allocation, deallocation and reference counting. It also includes common mistakes that could lead to memory leaks.

5.3.2 Coverage

The KB was constructed to include a wide range of over 1,000 functions from both GLib and GStreamer official documentation. It is structured to support memory-related information. It comes with function categories used in systems-level C programming. The entries include both basic GLib functions and higher-level media-centric GStreamer operations. Major categories include:

GLib

- Dynamic memory allocation
  g_malloc, g_realloc, g_slice_new, g_array_new, etc.
- Container and data structure management
  g_list_append, g_hash_table_insert, g_queue_push_tail, g_tree_insert, etc.
- Object lifecycle and reference counting
  g_object_ref, g_object_unref, g_object_weak_ref, etc.
- Deallocation and cleanup routines
  g_list_free_full, g_array_free, g_free, etc.
- Threading and concurrent utilities
  g_async_queue_push, g_thread_pool_free, g_mutex_lock, etc.

GStreamer

- Buffer and memory management
  gst_buffer_new, gst_buffer_ref, gst_buffer_unref, gst_memory_new_wrapped, etc.
- Caps and metadata structures
  gst_caps_new_simple, gst_caps_unref, gst_structure_new, etc.
- Object management and reference control
  gst_object_ref, gst_object_unref, gst_element_set_state, etc.
- Element and pad lifecycle
  gst_element_factory_make, gst_element_link, gst_pad_new, gst_pad_unlink, etc.
- Pipeline cleanup and streaming control
  gst_bin_add, gst_pipeline_new, gst_element_set_state, gst_object_unref, etc.

The KB is designed with list functions which consist of usage guidance, memory ownership semantics and cleanup expectations. All these information is essential for correctly evaluating memory leak risks. This makes it highly applicable to real-world C applications built with GLib and GStreamer.

5.4 Chunk Preprocessing

In RAG, the raw documentation for GLib and GStreamer functions are preprocessed into semantically meaningful chunks. This enables effective retrieval in the RAG component of the system. This step makes sure that during inference, the system can retrieve the most relevant, function-specific information rather than arbitrary or diluted text segments.

5.4.1 Function-Level Chunking Strategy

In traditional document splitting, it breaks text at fixed token or character limits. They often cut across sentences or functions. This leads to loss of semantic similarity. This reduces retrieval quality during inference. This system adopts a function-level chunking strategy. Each entry in the KB corresponds to exactly one documented function, along with a structured description of its memory behavior. Each chunk includes function name, memory ownership semantics, usage guidelines and cleanup requirements, common pitfalls or leak scenarios. This structured chunking provides high semantic similarity and ensures that every retrieved chunk is directly relevant to a memory leak detection task.

5.4.2 Integration with RecursiveCharacterTextSplitter

The system supports integration with LangChain's RecursiveCharacterTextSplitter. This helps to maintain formatting consistency and simplify future KB extensions. Although this chunking method (like character-based splitting) is not the main approach used, it is still available as a backup tool for breaking down large raw documentation files when preparing the KB in bulk. The splitter helps to make sure that there are no mid-sentence breaks keeping semantic clarity. This dual approach of combining strict function-level chunking with token-based preprocessing allows the system to maintain both semantic clarity and retrieval efficiency.

5.4.3 Output Format for Indexing

The final output of the preprocessing step is a flat list of JSON-like dictionary records. Each entry contains:

{
"name": "g_list_free_full",

"description": "Frees all the memory used by a GList, including each element using the provided free_func. Useful for preventing memory leaks in list structures."}

These entries are then embedded using Sentence Transformers and stored in the FAISS index for similarity-based retrieval at inference time.

5.5 FAISS Search + Cosine Similarity

The system integrates FAISS for vector-based search. This enables fast and accurate retrieval of relevant documentation during inference. FAISS is a highly optimized library for efficient similarity search and clustering of dense vectors. This makes it suitable for handling large-scale embeddings such as those from GLib and GStreamer documentation.

5.5.1 Embedding Strategy

Each document chunk in the KB is embedded using the microsoft/codebert-base SentenceTransformer. It converts the text of each function entry, that is the name and description into a dense vector. These embeddings are then stored in a FAISS index. This allows the system to measure semantic similarity between C code snippets and documentation entries. During inference, the function names or the entire code snippet is embedded in the same vector space and compared against the FAISS index. The retrieval

is based on cosine similarity. This measures how close the embedded query is to the documentation vectors in high-dimensional space.

5.5.2 Query Processing

When a C code snippet is submitted, the system extracts relevant GLib/GStreamer function calls using regex. If matches are found, then the embeddings of the function names are used to query the FAISS index. If no matches are found, the entire code snippet is embedded and used as the query. The top-k closest documentation chunks are retrieved based on cosine similarity scores. This dual querying helps for function-specific retrieval.

5.5.3 Cosine Similarity Usage

Cosine similarity is calculated by

$$\text{Sim}(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

Here the rank of the retrieved documentation is calculated based on relevance. This contributes to the uncertainty score computation also. It is done by taking 1−max(similarity). It indicates the semantic mismatch between input and context. This similarity-driven approach helps to make sure that the retrieved knowledge is both contextually aligned and semantically relevant. This is critical for accurate memory leak detection.

5.6 Prompt Formatting Example

A proper prompt is something that ensures the way the LLM processes both the input code and the retrieved context effectively. It must be carefully structured to guide the model toward a clear and accurate analysis. The prompts in this system are designed to serve dual roles. One is to provide the model with enough context to make a well-informed decision. The other one makes sure the consistency in how outputs are generated across different inference runs.

5.6.1 Prompt Construction Format

Each prompt follows a structured format containing that includes a high-level instruction, retrieved GLib/GStreamer function documentation, the C code snippet to be analyzed and a task-specific question regarding memory leak detection. The figure 7 shows a typical example of how the prompt is structured during inference.

```
Can you analyze the following code change, list all GLib and GStreamer related functions and check whether returned values must be freed
or not? Are there any memory leaks related to them?

== This is the most relevant information about GLib/GStreamer related functions ==
<RETRIEVED CONTEXT>

== This is the C Code I wish you to analyze ==
```c
<INPUT CODE SNIPPET>

== Please provide the result ==

Memory Leak Detected? (YES/NO)
Justification:
```

Figure 7: Structured Prompt
```

5.6.2 Example Prompt Instance

Assume the input code is:

```
GList *list = NULL;
list = g_list_append(list, g_strdup("example"));
// list is not freed
```

And the retrieved documentation includes:

- *g_list_append: Adds a new element to the end of the list. Does not transfer ownership of elements.*
- *g_list_free_full: Frees the list and calls a provided function to free each element.*
- *g_strdup: Returns a newly allocated string. Must be freed using g_free().*

The complete prompt would be:

Can you analyze the following code change, list all GLib and GStreamer related functions and check whether returned values must be freed or not? Are there any memory leaks related to them?

== This is the most relevant information about GLib/GStreamer related functions ==

g_list_append: Adds a new element to the end of the list. Does not transfer ownership of elements.

g_list_free_full: Frees the list and calls a provided function to free each element.

g_strdup: Returns a newly allocated string. Must be freed using g_free().

== This is the C Code I wish you to analyze ==

```c

*GList *list = NULL;*

list = g_list_append(list, g_strdup("example"));

// list is not freed

== Please provide the result ==

- *Memory Leak Detected? (YES/NO)*
- *Justification:*

5.7 Uncertainty Score Normalization

In the proposed memory leak detection system, the uncertainty score plays a central role. It decides the modulation behavior of the model during inference. It responsible for how strongly the LoRA fine-tuned layers influence the final prediction. This is done by analyzing how confident the system is in the retrieved documentation. However, to ensure that this score remains interpretable and compatible, it must be normalized within a consistent range. That is specifically between 0 and 1.

5.7.1 Components Used in Uncertainty Calculation

As described in Section 3.4, the uncertainty score is derived from a weighted combination of three key components. First one, is cosine similarity-based mismatch between the input code and the most relevant retrieved documentation. This is calculated with (1−max_similarity). The second one is what is missing from retrieved functions. That is the fraction of GLib/GStreamer functions used in the code that were not covered by the retrieved context. The final one is missing from KB. This is calculated by the fraction of GLib/GStreamer functions used in the code that are not found in the entire KB. Each of these components is scaled with pre-defined weights α, β, γ. These are set to emphasize the significance of documentation absence.

$$U=\alpha\cdot(1-\text{max\_similarity}) + \beta\cdot\text{missing\_retrieved\_fraction}+\gamma\cdot\text{missing\_kb\_fraction}$$

With:

$\alpha=0.2$, $\beta=0.35$, and $\gamma=0.45$

5.7.2 Normalization Strategy

The resulting uncertainty score U is designed to be in the range [0, 1]. For this the cosine similarity is naturally bounded between 0 and 1. Also both missing_retrieved_fraction and missing_kb_fraction are computed as ratios over total detected function calls, hence also lie in [0, 1]. Weights $\alpha+\beta+\gamma=1$, preserving the normalization of the overall score. This ensures that the final uncertainty score is bounded and interpretable, enabling it to be directly used as a scaling factor for the LoRA weight contribution during inference.

5.7.3 Role in Dynamic Adaptation

The normalized uncertainty score is used during inference to dynamically control the influence of LoRA updates. When uncertainty is very low (e.g., 0.1–0.3), the model trusts the retrieved context and applies minimal LoRA scaling. When uncertainty is high (e.g., 0.7–1.0), the system relies more on task-specific patterns learned through LoRA. This compensates for unreliable or missing documentation. This control allows the system for better generalization and fewer hallucinations.

6. Evaluation and Results

6.1 Overview

This section presents a detailed evaluation of the proposed hybrid memory leak detection system. The results are measured across multiple model configurations to highlight the contribution of each component. That is base model, LoRA, RAG, LoRA and RAG and uncertainty-aware integration. We compare metrics such as accuracy, precision, recall and FP rate to demonstrate the overall utility and reliability of the system.

6.2 Evaluation Metrics

To assess the classification performance, the following standard metrics are used:

- Accuracy: Proportion of total correct identifications.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

- Precision: Proportion of predicted "leak" instances that were actually leaks.

$$\text{Precision} = \frac{TP}{TP+FP}$$

- Recall: Proportion of actual leaks that were correctly identified.

$$\text{Recall} = \frac{TP}{TP+FN}$$

- F1 Score: Harmonic mean of precision and recall.

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Where:

- TP = True Positives (correctly identified leaks)
- TN = True Negatives (correctly identified non-leaks)
- FP = False Positives (non-leaks wrongly labeled as leaks)
- FN = False Negatives (leaks missed by the model)

Support is specified along with the metrics evaluation values. Support is the number of true instances of each class in the evaluation set used to compute these class-specific metrics.

6.2.1 Justification Evaluation

In addition to evaluating the system's classification accuracy, precision, recall and F1 score, this study also assesses the quality of the natural language justifications provided by the model for each memory

leak prediction. As the system is designed for use in developer-facing environments, the interpretability and reliability of these explanations are essential for practical adoption.

Since there is no existing benchmark for memory leak explanations in GLib/GStreamer code, a manual review process was adopted for the explanation evaluation. Explanations were considered valid if they referenced the correct functions, aligned with memory behavior and supported the final prediction. Although no automatic scoring metrics were used, this qualitative assessment confirmed that most justifications were logically consistent and helped interpret the model's decision, especially for developer-facing use cases.

6.3 Results

The proposed combined approach of RAG and LoRA-based fine-tuning was evaluated on both the test dataset and selected real-world code snippets. The objective was to assess the system effectiveness in identifying memory leaks within C code that makes use of the GLib and GStreamer libraries. The model successfully detected memory leaks in various code commits, demonstrating its ability to use both components of the architecture in a balanced and context-aware manner.

The figure 8 shows the code real world code section that comes with Glib and Gstreamer. The code is given to the RAG-LoRA adaptive system to get the leak details.

```c
{
  GstElement *element;
  GstElement *backchannel_bin;
  RTSPMediaStream *stream;

  g_return_if_fail (media != NULL);

  g_object_get (G_OBJECT (media), "element", &element, NULL);
  g_assert (element != NULL);

  backchannel_bin = gst_bin_get_by_name (GST_BIN (element), "onvif-backchannel");
  gst_object_unref (element);

  if (!backchannel_bin) {
    GST_DEBUG ("no backchannel bin, not collecting");
    return;
  }
```

Figure 8: C code snippet

```
### Memory Management Analysis

1. **g_object_get**:
   - This function does not return a GStreamer object that needs to be unreferenced.

2. **gst_bin_get_by_name**:
   - The `backchannel_bin` is retrieved using `gst_bin_get_by_name`.
   - The reference count of `backchannel_bin` is incremented, so it must be unreferenced when no longer needed.
   - In the provided code, `backchannel_bin` is not unreferenced after use, which could lead to a memory leak.

3. **gst_object_unref**:
   - The `element` is correctly unreferenced after retrieving the `backchannel_bin`.

### Potential Memory Leak

The code snippet does not unref the `backchannel_bin` after it is no longer needed. This could lead to a memory leak because the reference co
unt of `backchannel_bin` was incremented by `gst_bin_get_by_name` and was not decremented.

### Corrected Code

To fix the potential memory leak, you should unref the `backchannel_bin` after it is no longer needed. Here is the corrected code:

```c
{
  GstElement *element;
  GstElement *backchannel_bin;
  RTSPMediaStream *stream;

  g_return_if_fail (media != NULL);

  g_object_get (G_OBJECT (media), "element", &element, NULL);
  g_assert (element != NULL);

  backchannel_bin = gst_bin_get_by_name (GST_BIN (element), "onvif-backchannel");
  gst_object_unref (element);

  if (!backchannel_bin) {
    GST_DEBUG ("no backchannel bin, not collecting");
    return;
  }

  // Use backchannel_bin as needed

  gst_object_unref (backchannel_bin);  // Unref backchannel_bin when done
}
```

By adding `gst_object_unref (backchannel_bin);` at the end of the code block, you ensure that the reference count of `backchannel_bin` is pro
perly decremented, preventing a memory leak.
```

Figure 9: Memory leak detection response

Figure 9 illustrates the output generated by the adaptive LLM. The system first identifies all GLib and GStreamer functions present in the input code, retrieves the corresponding documentation using the RAG component and then analyzes the code for potential memory leaks based on the retrieved context.

The test data set is given to the model to check the performance metrics. Recall, precision and accuracy are calculated based on the test data set that contains balanced count of leak non leak samples. Figure 10 shows the performance metrics. The classification results summaries that system is effective in memory leak detection by correctly classifying 71% of the code samples as either leak or non-leak. Leak Class: showed a high recall of 0.77, meaning it successfully detected most memory leak instances. The precision is 0.69, suggestion that most of the predicted leaks were indeed actual leaks. The F1-score of 0.73 reflects a well-balanced performance for leak detection. In the non-leak class, the model achieved a precision of 0.73 and a recall of 0.64, leading to an F1-score of 0.68. This shows that while non-leak predictions were mostly correct, a few actual non-leaky samples were misclassified as leaks.

	precision	recall	f1-score	support
leak	0.69	0.77	0.73	51
non_leak	0.73	0.64	0.68	49
accuracy			0.71	100
macro avg	0.71	0.70	0.70	100
weighted avg	0.71	0.71	0.70	100

Figure 10: Performance evaluation

6.4 Model Variants Compared

To systematically evaluate the system, we compare the following four model configurations as shown in Table 1.

Model Configuration	Description
Base Model	DeepSeek Coder 1.3B with no fine-tuning and no RAG.
LoRA-Only	Fine-tuned using memory-leak classification dataset but without any retrieval
RAG-Only	Uses external documentation retrieval but no LoRA training.
LoRA + RAG	RAG combined with LoRA in pipeline without uncertainty factor calculation
LoRA + RAG + Adaptive system	Combines LoRA fine-tuning, RAG retrieval and dynamic inference modulation based on uncertainty scoring.

Table 1: Variants for evaluation

	precision	recall	f1-score	support
leak	0.50	0.57	0.53	51
non_leak	0.48	0.41	0.44	49
accuracy			0.49	100
macro avg	0.49	0.49	0.49	100
weighted avg	0.49	0.49	0.49	100

Figure 11: Base Model

Classification Report:

	precision	recall	f1-score	support
leak	0.53	1.00	0.69	51
non_leak	1.00	0.08	0.15	49
accuracy			0.55	100
macro avg	0.77	0.54	0.42	100
weighted avg	0.76	0.55	0.43	100

Figure 12: LoRA fine Tuned Model

	precision	recall	f1-score	support
leak	0.53	0.82	0.64	51
non_leak	0.55	0.22	0.32	49
accuracy			0.53	100
macro avg	0.54	0.52	0.48	100
weighted avg	0.54	0.53	0.48	100

Figure 13: RAG model

	precision	recall	f1-score	support
leak	0.55	0.78	0.64	23
non_leak	0.58	0.32	0.41	22
accuracy			0.56	45
macro avg	0.56	0.55	0.53	45
weighted avg	0.56	0.56	0.53	45

Figure 14: RAG and LoRA

When comparing all the model variants, the adaptive system, which combines LoRA fine-tuning, RAG-based retrieval and uncertainty-guided inference, clearly gives the best performance. As show in figure 10, it achieves the highest accuracy of **0.71** and maintains a good balance between both classes than any other method that shows in figure 11, Figure 12, figure 13 or figure 14. For memory leak cases, it reaches a precision of **0.69** and a recall of **0.77**, which means it not only detects leaks well but does so with fewer false positives. At the same time, it also keeps non-leak predictions reliable, with **0.73** precision and **0.64** recall. This strong balance between both classes leads to the best overall F1-scores and a macro F1 of **0.70**, showing that the system performs well across different types of inputs.

In contrast, the base model struggles with an accuracy of only **0.49** and low recall and precision for both classes. The LoRA-only and RAG-only models tend to focus too much on leak detection and often misclassify non-leaky code, which results in higher FPs. Even when LoRA and RAG are combined without uncertainty control, the performance improvement is limited. Figure 11,12,13 and 14 shows the evaluation metrics. By adding the uncertainty-aware adjustment in the adaptive model, the system becomes smarter in deciding when to rely on retrieved knowledge and when to trust the fine-tuned model. This makes it more accurate and practical for real-world memory leak detection in GLib and GStreamer-based C programs.

6.5 Effectiveness of the Combined Approach

These comparative results provide a clear and direct answer to RQ1 *"How the combined approach be applied to knowledge augmentation and parameter-efficient fine-tuning to make an LLM task-specific for memory leak detection in GLib and GStreamer- base C code?"*

The evaluations demonstrates that the proposed hybrid architecture, combining LoRA-based fine-tuning, RAG and adaptive uncertainty-based inference offers the best overall performance across all tested configurations (Figure 10). The system consistently achieved the highest scores in terms of accuracy (71%), leak recall (0.77) and macro F1 score (0.70), outperforming the base model as well as the LoRA-only and RAG-only variants.

These improvements validate that the LoRA component enables the model to generalize well on task-specific patterns related to memory management. Also, the RAG component supplies domain-specific information critical for understanding GLib and GStreamer semantics during inference. Most importantly, the uncertainty-aware modulation mechanism dynamically balances these two sources giving importance to the retrieval when relevant and falling back to fine-tuned behavior when the context is weak or missing. This leads to more context-sensitive, reliable and explainable memory leak detection.

In short, this result confirms that the integrated approach not only enhances raw classification performance but also addresses the limitations of each individual component when used in isolation. The balance between learned task knowledge (LoRA), external documentation (RAG), and adaptive inference control (uncertainty scoring) is key to enabling accurate static analysis of memory leaks in real-world C codebases.

6.6. Justification Evaluation samples

The memory leak detection system was also evaluated for the quality of its textual justifications. Samples of both leaky and non-leaky code explanations generated by the model was manually reviewed.

Sample 1

```
{
  +    GstRTSPStatusCode code)
+{
+  GSList *errors = NULL;
+  GSList *errors_iter = NULL;
+  GstMessage *error = NULL;
+  GstObject *sender = NULL;
+  RTSPMedia *media = NULL;
+  GstRTSPStatusCode adjusted_code;
+  GstElement *rtsp_bin = NULL;
+
+  g_return_val_if_fail (client != NULL, code);
+  g_return_val_if_fail (ctx != NULL, code);
+
+  adjusted_code = code;
+  media = RTSP_MEDIA (ctx->media);
+  if (media != NULL) {
+    errors = rtsp_media_get_pipeline_errors (media);
+    rtsp_bin = gst_rtsp_media_get_element (ctx->media);
+    for (errors_iter = errors; errors_iter != NULL;
+        errors_iter = g_slist_next (errors_iter)) {
+      error = errors_iter->data;
+      sender = GST_MESSAGE_SRC (error);
+      if (gst_object_get_parent (sender) == GST_OBJECT (rtsp_bin)) {
+        adjusted_code = GST_RTSP_STS_BAD_REQUEST;
+        GST_DEBUG_OBJECT (client,
+            "got error from %s, adjusting error response code from %d to %d",
+            gst_element_get_name (sender), code, adjusted_code);
+        break;
+      }
+    }
+
+    g_slist_free_full (errors, (GDestroyNotify)gst_message_unref);
+  }
+  return adjusted_code;
+}
```

Figure 15: Sample 1

Figure 15 presents a real example of a memory leak scenario detected by the hybrid model. The code involves a series of operations using GStreamer and GLib functions, including gst_rtsps_media_get_element, gst_element_get_name, and gst_object_get_parent. These functions typically return objects with increased reference counts, requiring explicit cleanup using gst_object_unref and similar mechanism. In this case, the model correctly predicts a leak and provides a natural language explanation indicating:

“There were potential memory leaks related to gst_rtsps_media_get_element, gst_element_get_name, and gst_object_get_parent.
”

```
{
  GstRTSPStatusCode code)
{
  GSList *errors = NULL;
  GSList *errors_iter = NULL;
  GstMessage *error = NULL;
  GstObject *sender = NULL;
  RTSPMedia *media = NULL;
  GstRTSPStatusCode adjusted_code;
  GstElement *rtsp_bin = NULL;

  g_return_val_if_fail (client != NULL, code);
  g_return_val_if_fail (ctx != NULL, code);

  adjusted_code = code;
  media = RTSP_MEDIA (ctx->media);
  if (media != NULL) {
    errors = rtsp_media_get_pipeline_errors (media);
    rtsp_bin = gst_rtsp_media_get_element (ctx->media);
    for (errors_iter = errors; errors_iter != NULL;
        errors_iter = g_slist_next (errors_iter)) {
      error = errors_iter->data;
      sender = GST_MESSAGE_SRC (error);
      GstObject *parent = gst_object_get_parent (sender);
      if (parent == GST_OBJECT (rtsp_bin)) {
        adjusted_code = GST_RTSP_STS_BAD_REQUEST;
        char *name = gst_element_get_name (sender);
        GST_DEBUG_OBJECT (client,
            "got error from %s, adjusting error response code from %d to %d",
            name, code, adjusted_code);
        g_free(name);
        gst_object_unref(parent);
        break;
      }
      gst_object_unref(parent);
    }

    g_slist_free_full (errors, (GDestroyNotify)gst_message_unref);
    gst_object_unref(rtsp_bin);
  }
  return adjusted_code;
}
```

Summary

- **Memory Leaks**: There were potential memory leaks related to `gst_rtsp_media_get_element`, `gst_element_get_name`, and `gst_object_get_parent`.
- **Fixes**: Added necessary `gst_object_unref` and `g_free` calls to avoid memory leaks.

Figure 16: Sample 1 response

Figure 16, shows that justification clearly identifies all relevant function calls associated with dynamic memory ownership, demonstrating the model’s awareness of GStreamer’s memory management patterns. The explanation is not only correct but also aligns well with the retrieved documentation context. Additionally, the model suggests appropriate fixes, including:

- gst_object_unref(parent) to release the parent object returned by gst_object_get_parent
- g_list_free_full(errors, (GDestroyNotify)gst_message_unref) to clean up the dynamically created message list

These additions are syntactically and semantically valid and reflect standard GStreamer memory safety practices. The explanation is interpretable and provides actionable reasoning, supporting the model's decision

Sample 2

Figure 17 demonstrates a non-leaky scenario in which the model accurately identifies that the code manages memory responsibly. The C snippet uses several GStreamer and GLib functions—including gst_rtsps_media_get_element, gst_object_get_parent, gst_element_get_name and g_list_free_full—each of which typically requires explicit deallocation or reference management.

```
{
 GstRTSPStatusCode
adjust_status_code (GstRTSPClient *client, RTSPContext *ctx, GstRTSPStatusCode code)
{
  GSList *errors = NULL;
  GSList *errors_iter = NULL;
  GstMessage *error = NULL;
  GstObject *sender = NULL;
  GstRTSPStatusCode adjusted_code;
  GstElement *rtsp_bin = NULL;

  g_return_val_if_fail (client != NULL, code);
  g_return_val_if_fail (ctx != NULL, code);

  adjusted_code = code;
  rtsp_bin = gst_rtsp_media_get_element (ctx->media);

  if (ctx->media != NULL) {
    errors = rtsp_media_get_pipeline_errors (RTSP_MEDIA (ctx->media));

    for (errors_iter = errors; errors_iter != NULL; errors_iter = g_slist_next (errors_iter)) {
      error = errors_iter->data;
      sender = GST_MESSAGE_SRC (error);
      GstObject *parent = gst_object_get_parent (sender);

      if (parent && parent == GST_OBJECT (rtsp_bin)) {
        char *name = gst_element_get_name (GST_ELEMENT (sender));
        GST_DEBUG_OBJECT (client, "Error from %s, changing status to BAD_REQUEST", name);
        g_free (name);
        adjusted_code = GST_RTSP_STS_BAD_REQUEST;
        gst_object_unref (parent);
        break;
      }

      if (parent)
        gst_object_unref (parent);
    }

    g_slist_free_full (errors, (GDestroyNotify) gst_message_unref);
    if (rtsp_bin)
      gst_object_unref (rtsp_bin);
  }

  return adjusted_code;
}
```

Figure 17: Sample 2

```
### Analysis of Memory Management

1. **`gst_rtsp_media_get_element`**
   - **Return Value**: The internal `GstElement` used to construct a `GstRTSPMedia` instance.
   - **Ownership**: You own the returned element and must call `gst_object_unref()` when done.
   - **Memory Management**: The code correctly calls `gst_object_unref(rtsp_bin)` at the end of the function.

2. **`gst_object_get_parent`**
   - **Return Value**: The parent of a given `GstObject` with its reference count incremented.
   - **Ownership**: You own the returned object and must call `gst_object_unref()` when done.
   - **Memory Management**: The code correctly calls `gst_object_unref(parent)` both inside and outside the loop.
                            Collapse Output
3. **`gst_element_get_name`**
   - **Return Value**: A newly allocated string containing the name of the given `GstElement`.
   - **Ownership**: You must free the returned string with `g_free()` after use.
   - **Memory Management**: The code correctly calls `g_free(name)` after using the name.

4. **`gst_object_unref`**
   - **Usage**: This function is used to decrement the reference count of a `GstObject`.
   - **Memory Management**: The code correctly uses `gst_object_unref` to manage the reference counts of `rtsp_bin` and `parent`.

5. **`gst_message_unref`**
   - **Usage**: This function is used to decrement the reference count of a `GstMessage`.
   - **Memory Management**: The code correctly uses `g_slist_free_full(errors, (GDestroyNotify) gst_message_unref)` to free the list of messa
ges.

### Potential Memory Leaks

The code appears to correctly manage the memory for all GStreamer-related functions. Here are the key points:

- The `rtsp_bin` element is correctly unreferenced at the end of the function.
- The `parent` object is correctly unreferenced both inside and outside the loop.
- The `name` string is correctly freed after use.
- The list of errors is correctly freed using `g_slist_free_full` with `gst_message_unref`.

### Conclusion

The provided code does not have any apparent memory leaks related to the GStreamer functions. The memory management is correctly handled by e
nsuring that all returned values are properly freed or unreferenced.
```

Figure 18: Sample 2-response

Figure 18 shows that the model correctly predicts non-leak and provides a structured explanation confirming that all memory-sensitive operations are safely handled. Key points from the model's justification include:

- gst_object_unref(parent) is called both inside and outside the conditional block.
- g_list_free_full(errors, (GDestroyNotify)gst_message_unref) is used to properly free the list of messages.
- The g_free(name) function is applied after allocating the string from gst_element_get_name.

The explanation also includes a breakdown of memory ownership responsibilities for each function. This structured response clearly shows that the model understands not just the syntax, but the semantic memory behavior required by GLib and GStreamer APIs.

Sample 3

Figure 19 shows an example where a memory leak is introduced by commenting out two cleanup statements: g_error_free(err) and g_free(debug). These lines are responsible for releasing memory allocated by g_error and the debug string returned by gst_message_parse_error. By commenting them out, the application fails to deallocate memory, causing a leak.

The model correctly identifies the leak and explains:

"The code introduces a memory leak by not freeing the GError and debug string."

Figure 20 shows that the justification explicitly references the two variables involved and explains the cause of the leak. It also provides a corrected version of the code with the cleanup lines restored, along with a summary highlighting the key fix:

"The fix involves uncommenting and correctly calling g_error_free(err) and g_free(debug) to free the allocated memory."

This example is easy to interpret: the original issue is clearly annotated in the code and the fix precisely matches the identified problem. The explanation aligns directly with the code context, helping the user understand both what is wrong and how to resolve it.

```
switch (GST_MESSAGE_TYPE (message)) {
  case GST_MESSAGE_ERROR:
  {
    GError *err;
    gchar *debug;

    gst_message_parse_error (message, &err, &debug);

    // Simulated logging, using standard macros only
    GST_DEBUG ("%s: %s", GST_OBJECT_NAME (GST_MESSAGE_SRC (message)), err->message);
    GST_DEBUG ("bus debug message: '%s'", debug);

    // ❌ Memory leak introduced by commenting out cleanup:
    // g_error_free (err);
    // g_free (debug);

    break;
  }
  default:
    break;
}
```

Figure 19: Sample 3

````
### Memory Leak Identification

The code comments indicate that the cleanup lines are commented out:

```c
// g_error_free (err);
// g_free (debug);
```

This means that the `GError` and the debug string are not being freed, which will result in a memory leak.

### Fixing the Memory Leak

To fix the memory leak, you need to uncomment and properly free the allocated memory:

```c
switch (GST_MESSAGE_TYPE (message)) {
  case GST_MESSAGE_ERROR:
  {
    GError *err;
    gchar *debug;

    gst_message_parse_error (message, &err, &debug);

    // Simulated logging, using standard macros only
    GST_DEBUG ("%s: %s", GST_OBJECT_NAME (GST_MESSAGE_SRC (message)), err->message);
    GST_DEBUG ("bus debug message: '%s'", debug);

    // Fix memory leak by freeing the allocated memory
    g_error_free (err);
    g_free (debug);

    break;
  }
  default:
    break;
}
```

### Summary

- The original code introduces a memory leak by not freeing the `GError` and debug string.
- The fix involves uncommenting and correctly calling `g_error_free(err)` and `g_free(debug)` to free the allocated memory.

By ensuring that all dynamically allocated resources are properly freed, you can prevent memory leaks in your application.
````

Figure 20: Sample 3-response

These examples support the conclusion that the model can recognize well-managed code and can justify its leaky or non-leak prediction with detailed reasoning grounded in API-specific memory rules. Such interpretability is valuable for developers seeking not only automated detection but confidence in the reasoning behind it.

7. Discussion

7.1 Interpretation of Results

The outcomes of the proposed memory leak detection system evaluation results clearly shows that the proposed hybrid system performs better than the baseline configurations in detecting memory leaks in C programs. This is particularly visible in those involving GLib and GStreamer functions. The base model used is DeepSeek Coder 1.3B. It can understand code syntax but lacked domain-specific reasoning. This leads to inconsistent or incorrect predictions. LoRA fine-tuning improved this by learning task-specific memory leak patterns. But sometimes it over-predicted leaks and this results in high recall and lower precision.

When RAG was introduced, it enhanced the model's access to real-time, domain-aware documentation. But the retrieval alone lacked the memory allocation behavior learned through training. This makes the model to less understand leak patterns. It was only when both LoRA and RAG were combined that a significant balance was achieved. This helps the model to handle both recall patterns and analyzed with retrieved context. However, without adaptive control, it occasionally produced incorrect outputs when irrelevant or noisy context was retrieved.

The most notable improvement comes with the integration of uncertainty scoring. This allowed the system to dynamically control how much it should trust the retrieved documentation versus the fine-tuned parameters. This adaptive behavior improved classification accuracy. Also, this reduces FPs and enhanced explainability. In short, the system became capable of handling its inference strategy based on the relevance of the input and the context provided.

These findings directly address both RQs. It shows a good way of combining both RAG and LoRA for a better result. Also, it shows how the accuracy, recall and precision significantly improved in the task-specific model compared to the base model, especially when uncertainty-aware adaptation was applied.

7.2 Practicality on Resource-Constrained Devices

One of the major strengths of the system is its ability to operate on a resource-constrained device such as a Mac with 25GB RAM. The use of LoRA mainly makes this possible. This significantly reduces the number of trainable parameters during fine-tuning. Full model fine-tuning normally requires extensive GPU resources. But LoRA allows for adapting large models using minimal memory and processing power. All training, evaluation and inference steps were performed locally, without using on cloud GPUs or distributed compute environments. This demonstrates the feasibility of deploying similar models in embedded development environments.

Furthermore, the efficient vector search capabilities provided by FAISS, combined with SentenceTransformers for document embeddings. This ensures that the RAG component remained responsive and memory-efficient throughout.

7.3 Impact on Static Code Analysis and Real-Time Tooling

Traditional static analysis tools like Clang or Cppcheck normally operate on predefined rules. So, they lack flexibility when dealing with custom libraries or unfamiliar memory ownership semantics. But the proposed system adapts to new patterns by learning from labeled examples and referencing external documentation at runtime. This makes the model particularly effective in real-world scenarios where memory management practices vary across codebases, frameworks and developer styles. By integrating RAG, the system overcomes a major limitation of typical static tools. That is lack of stack-specific

understanding. It can analyze code not just syntactically but semantically. This helps to identifying leaks based on both observed patterns and documented details.

In addition, the system's structured prompts and explainable outputs allow easy integration into real-time development environments. For instance, it can be adapted as a plugin for Integrated Development Environment (IDEs) or code review pipelines. This provides developers with not just warnings but contextual explanations and even suggested fixes. This makes the system well-suited for early-stage bug detection during development and pre-commit checks for CI or Continuous Deployment (CD) pipelines. Also, this is a lightweight alternative to dynamic tools like Valgrind in performance-critical systems.

7.4 Challenges Faced

While the system achieved promising results, several challenges were faced during development. One of the most important things was the limited availability of high-quality, labeled memory leak datasets. It was less in samples as it is especially involving GLib and GStreamer. Manual annotation was required to create a balanced dataset of leaky and non-leaky C code examples. This limitation affected the diversity of training samples and generalizability to unseen code patterns. Another important thing was high sensitivity to the structure and clarity of prompts. Minor inconsistencies in prompt format sometimes led to unpredictable outputs. This was addressed by enforcing a standardized, instruction-following prompt template. But it remains a concern for real-world deployments where input can vary. Another notable issue was, although DeepSeek Coder 1.3B is relatively lightweight, inference still required careful memory management on a local machine. For larger-scale adoption or integration into toolchains, a more compact distilled version may be explored in future work. Last but not the least is the RAG database. The success of RAG heavily depends on the quality and completeness of the KB. If relevant function documentation is missing or poorly retrieved, the system's predictions may degrade. This was partially addressed through the uncertainty scoring mechanism, but the solution is not completely error free.

8. Conclusion and Future Work

8.1 Conclusion

Memory leak detection in low-level languages like C remains a critical challenge in modern software development. This is particularly critical in systems that rely on libraries such as GLib and GStreamer. Traditional static and dynamic analysis tools are useful to an extent. But they are practically less useful when applied to complex or framework-specific codebases. These tools either rely on hardcoded patterns that lack adaptability or impose significant runtime overhead. This makes them less practical for integration into real-time development workflows.

This thesis proposed and implemented an adaptive system for detecting memory leaks in C programs. The approach combines two techniques. One is LoRA for efficient fine-tuning and RAG for injecting domain-specific knowledge at inference time. The hybrid architecture was further enhanced through an uncertainty scoring mechanism that dynamically adjusts model behavior based on the confidence in retrieved documentation and code context alignment.

The experimental results demonstrate that this combined approach significantly improves recall precision and accuracy compared to baseline models. The LoRA module learned task-specific memory patterns efficiently without requiring full model retraining. The RAG pipeline provided real-time access to function-level documentation for GLib and GStreamer. This helps the model to reason through context-aware memory safety. The uncertainty-aware inference is used. This helps the system to avoid over dependency on either the training data or the retrieved knowledge. This leads to more reliable and explainable results.

Additionally, the system was successfully implemented and evaluated on a resource-constrained local machine with 25GB RAM. This confirms its practicality for real-world use without access to large-scale GPU infrastructure. This design allows the system to be adapted for integration into code review tools, development environments or CI pipelines. In short, this research contributes an interpretable and context aware solution to the memory leak problem in C. It shows the real-world applicability of LLMs when paired with efficient fine-tuning and retrieval techniques. This is particularly for static code analysis in safety-critical software environments.

8.2 Future Work

The current system has demonstrated promising results in detecting memory leaks in GLib and GStreamer-based C code. But several limitations remain. These limitations can be addressed to enhance the system's applicability, scalability, and depth of analysis. The following future directions are proposed based on the practical gaps observed during this research.

1. Extending the System to Detect Runtime Memory Issues

The present framework is designed for static code analysis only. This is effective in many scenarios. But certain memory-related issues are completely not handled by this. It includes use-after-free, double-free and memory corruption. It is not handled because it can only be confirmed at runtime. These are often influenced by program execution paths and dynamic memory state. Therefore, a major area for future enhancement is supporting dynamic memory error detection**.**

This can be approached by expanding the RAG KB to include documentation of detailed descriptions of runtime behaviors. That is logs, runtime bug reports, memory layout diagrams and instrumentation outputs from tools like Valgrind or AddressSanitizer. Integrating these documents into the retrieval

mechanism will allow the model to analyze the execution-phase behaviors also. This will make the system more capable of identifying memory leaks that depend on runtime conditions.

In parallel, a long-term vision could involve integrating lightweight runtime traces with the inference system. These could be pre-processed into semantic summaries (e.g., "buffer not released on loop exit") and included in the prompt. This can help the model to avoid the gap between static structure and runtime semantics.

2. Enhancing Scalability through Pre-Filtering of Large Codebases

A major practical limitation of the current system is its capacity to analyze only small code segments. That is 10 to 15 function codes or small commits. In large-scale industrial codebases, memory leaks may span across multiple files or occur in deeply nested functions. Therefore, a pre-filtering mechanism is proposed for future implementation.

This mechanism would scan larger codebases to locate and extract the most memory-relevant segments**.** This focus on function calls associated with memory allocation (e.g., gst_append, g_list_append) or suspicious patterns (e.g., missing deallocation). These filtered segments would then be passed to the memory leak detection pipeline.

Such a pre-filtering layer could be implemented using a small, task-specific LLM or lightweight classification model. That is the model that is trained to scan large codebases and flag potentially memory-relevant regions. Therefore, without analyzing the full code in-depth, this model would act as a first-stage filter that identifies functions, blocks or commits where memory management patterns are present. It includes areas such as allocation without matching deallocation and code sections that appear suspicious. Detecting unmatched usage of functions like g_malloc, g_new, or g_object_ref without subsequent g_free or g_object_unref also comes to this section. To keep the system lightweight and efficient, this pre-filtering step could use a smaller version of the main LLM that tell whether a piece of code is likely to involve memory-related behavior or not. It does not need to detect leaks directly. But they need to flag parts of the code where memory is allocated, freed or passed around. Thus, the system does not have to process the entire codebase in detail. On the same time, it can quickly scan large files and send only the memory-relevant parts to the main detection pipeline for deeper analysis.

3.*Structured Output via Multi-Level Prompt Engineering*

Currently, the model provides natural language responses indicating whether a leak is present. This comes along with a textual justification. Even though this is informative, sometimes these outputs can be inconsistent or difficult to include in automated workflows.

Future work will focus on advanced prompt engineering to produce structured and layered outputs. For example, the model could return answers in a JSON format with distinct fields such as:

```
{
 "leak_detected": "YES",
 "affected_function": "g_list_append",
 "issue": "Allocated data not freed",
 "recommendation": "Use g_list_free_full with proper free_func",
 "certainty_score": 0.84
}
```

This structured format will help for processing and integration into developer tools. Also, it can use even for visualization in IDEs. Moreover, by defining multi-level prompts like reasoning, then detection to recommendation, the system could be made more explainable and reliable. This can take next step to

integrating multi-turn inference, where the model first identifies memory issues and then answers follow-up questions such as "What is the fix?" or "Which line is problematic?"

Such structured and hierarchical prompting would significantly improve the system's usability especially in team-based or industrial software development settings.

4. Explore with Larger and More Capable Language Model

While the current system uses a lightweight 1.3B parameter model (DeepSeek Coder) to ensure compatibility with resource-constrained environments, the quality of memory leak detection can potentially be improved by leveraging larger, more capable LLMs. These larger models often show stronger reasoning abilities, deeper contextual understanding and better generalization across unseen code patterns. In scenarios where sufficient computational resources are available—such as GPU-enabled servers or cloud-based deployments—future extensions of the system could incorporate high-capacity models like DeepSeek Coder 6.7B, CodeLlama, Codestral-2501 or GPT-based variants. This shift would enable the model to better understand complex memory interactions, especially in large codebases, and reduce FPs or ambiguous explanations. Additionally, more powerful LLMs could support advanced features like multi-turn code analysis, line-level issue identification and improved natural language justifications.

5. *Generalizing to Other Libraries and Programming Contexts*

The system is specially designed to detect memory leaks in C programs that use the GLib and GStreamer libraries. But in real-world software development, many other C-based libraries also have complex ways of managing memory. It includes libraries like GTK for building graphical applications, OpenSSL for secure communication, or libav for audio/video processing.

For more generalizing the application of the system, the future work would be to expand the system so it can handle other libraries as well. This would involve collecting memory-related documentation and best practices for each new library and adding that to the knowledge base. Once the new information is included, we can fine-tune the model with a few new examples. This will allow it to learn how memory is managed in those contexts. Since the current system already uses RAG and has a flexible architecture, we would not need to change much to make this work. The same idea could even be extended beyond C to languages like C++, where memory issues like double-deletion or improper use of RAII are common.

By making the system more general, it could become a powerful tool for detecting memory problems in many kinds of projects—not just those that use GLib or GStreamer, but across a wider range of libraries and programming environments.

Acknowledgement

This master thesis was carried out in collaboration with the OS Streaming department of Axis Communications, Lund, Sweden, during the spring semester of 2024. We would like to express our sincere gratitude to all those who contributed to the successful completion of this work. We are especially thankful to Axis Communications for providing us with the opportunity, technical guidance, and supportive environment to conduct this research. The experience and resources provided by the team were instrumental in shaping the direction and outcome of this thesis. We would like to extend our heartfelt thanks to our examiner, Vladimir Tarasov and supervisor He Tan at Jönköping University, for their continuous support, insightful feedback, and encouragement throughout the thesis period. Their academic expertise and constructive suggestions significantly enriched our work. Lastly, we acknowledge the collaborative spirit and shared motivation of our fellow thesis students and colleagues at Axis, whose discussions and support added great value during this research.

REFERENCES

Alawwad, H. A., Alhothali, A., Naseem, U., Alkhathlan, A., & Jamal, A. (2025). Enhancing textual textbook question answering with large language models and retrieval augmented generation. *Pattern Recognition*, *162*, 111332. https://doi.org/10.1016/j.patcog.2024.111332

Andrzejak, A., Eichler, F., & Ghanavati, M. (2017). Detection of Memory Leaks in C/C++ Code via Machine Learning. *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 252–258. https://doi.org/10.1109/ISSREW.2017.72

Anisuzzaman, D. M., Malins, J. G., Friedman, P. A., & Attia, Z. I. (2025). Fine-Tuning Large Language Models for Specialized Use Cases. *Mayo Clinic Proceedings: Digital Health*, *3*(1), 100184. https://doi.org/10.1016/j.mcpdig.2024.11.005

Arslan, M., Ghanem, H., Munawar, S., & Cruz, C. (2024). A Survey on RAG with LLMs. *Procedia Computer Science*, *246*, 3781–3790. https://doi.org/10.1016/j.procs.2024.09.178

Aslanyan, H., Gevorgyan, Z., Mkoyan, R., Movsisyan, H., Sahakyan, V., & Sargsyan, S. (2022). Static Analysis Methods For Memory Leak Detection: A Survey. *2022 Ivannikov Memorial Workshop (IVMEM)*, 1–6. https://doi.org/10.1109/IVMEM57067.2022.9983955

Aslanyan, H., Movsisyan, H., Hovhannisyan, H., Gevorgyan, Z., Mkoyan, R., Avetisyan, A., & Sargsyan, S. (2024). Combining Static Analysis With Directed Symbolic Execution for Scalable and Accurate Memory Leak Detection. *IEEE Access*, *12*, 80128–80137. https://doi.org/10.1109/ACCESS.2024.3409838

Azhari, V., Bhamra, S., Ezzati-Jivan, N., & Tetreault, F. (2021). Efficient heap monitoring tool for memory leak detection and root-cause analysis. *2021 IEEE International*

Conference on Big Data (Big Data), 3020–3030. https://doi.org/10.1109/BigData52589.2021.9671473

Barnett, S., Kurniawan, S., Thudumu, S., Brannelly, Z., & Abdelrazek, M. (2024). *Seven Failure Points When Engineering a Retrieval Augmented Generation System* (No. arXiv:2401.05856). arXiv. https://doi.org/10.48550/arXiv.2401.05856

Biderman, D., Portes, J., Ortiz, J. J. G., Paul, M., Greengard, P., Jennings, C., King, D., Havens, S., Chiley, V., Frankle, J., Blakeney, C., & Cunningham, J. P. (2024). *LoRA Learns Less and Forgets Less* (No. arXiv:2405.09673). arXiv. https://doi.org/10.48550/arXiv.2405.09673

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., & Tao, C. (2022). MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. *Proceedings of the 44th International Conference on Software Engineering*, 1456–1468. https://doi.org/10.1145/3510003.3510219

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Wu, X., Tao, C., Zhang, T., & Liu, W. (2024). Learning to Detect Memory-related Vulnerabilities. *ACM Transactions on Software Engineering and Methodology*, *33*(2), 1–35. https://doi.org/10.1145/3624744

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). *Evaluating Large Language Models Trained on Code* (No. arXiv:2107.03374). arXiv. https://doi.org/10.48550/arXiv.2107.03374

Fan Lin-bo, Wu Ying-cheng, Gong Ming-qing, & Zhao Ming. (2009). Research on the transition between the software reliability and safety. *2009 2nd IEEE International Conference on Computer Science and Information Technology*, 372–376. https://doi.org/10.1109/ICCSIT.2009.5234521

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). *CodeBERT: A Pre-Trained Model for Programming and Natural Languages* (No. arXiv:2002.08155). arXiv. https://doi.org/10.48550/arXiv.2002.08155

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W., Zettlemoyer, L., & Lewis, M. (2023). *InCoder: A Generative Model for Code Infilling and Synthesis* (No. arXiv:2204.05999). arXiv. https://doi.org/10.48550/arXiv.2204.05999

Gajulamandyam, D. K., Veerla, S., Emami, Y., Lee, K., Li, Y., Mamillapalli, J. S., & Shim, S. (2025). Domain Specific Finetuning of LLMs Using PEFT Techniques. *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)*, 00484–00490. https://doi.org/10.1109/CCWC62904.2025.10903789

Gangwar, D. K., & Katal, A. (2021). Memory Leak Detection Tools: A Comparative Analysis. *2021 International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*, 315–320. https://doi.org/10.1109/RTEICT52294.2021.9574012

Gao, Q., Xiong, Y., Mi, Y., Zhang, L., Yang, W., Zhou, Z., Xie, B., & Mei, H. (2015). Safe Memory-Leak Fixing for C Programs. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 459–470. https://doi.org/10.1109/ICSE.2015.64

Gao, Y., Xiong, Y., Zhong, Y., Bi, Y., Xue, M., & Wang, H. (2025). *Synergizing RAG and Reasoning: A Systematic Review* (No. arXiv:2504.15909). arXiv. https://doi.org/10.48550/arXiv.2504.15909

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., & Liang, W. (2024). *DeepSeek-Coder: When the Large*

Language Model Meets Programming -- The Rise of Code Intelligence (No. arXiv:2401.14196). arXiv. https://doi.org/10.48550/arXiv.2401.14196

Han, Z., Gao, C., Liu, J., Zhang, J., & Zhang, S. Q. (2024). *Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey* (No. arXiv:2403.14608). arXiv. https://doi.org/10.48550/arXiv.2403.14608

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models* (No. arXiv:2106.09685). arXiv. https://doi.org/10.48550/arXiv.2106.09685

Jain, R., Agrawal, R., Gupta, R., Jain, R. K., Kapil, N., & Saxena, A. (2020). Detection of Memory Leaks in C/C++. *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*, 1–6. https://doi.org/10.1109/SCEECS48394.2020.32

Jung, C., Lee, S., Raman, E., & Pande, S. (2014a). Automated memory leak detection for production use. *Proceedings of the 36th International Conference on Software Engineering*, 825–836. https://doi.org/10.1145/2568225.2568311

Jung, C., Lee, S., Raman, E., & Pande, S. (2014b). Automated memory leak detection for production use. *Proceedings of the 36th International Conference on Software Engineering*, 825–836. https://doi.org/10.1145/2568225.2568311

Le-Cong, T., Le, B., & Murray, T. (2024). *Semantic-guided Search for Efficient Program Repair with Large Language Models* (No. arXiv:2410.16655). arXiv. https://doi.org/10.48550/arXiv.2410.16655

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2021). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* (No. arXiv:2005.11401). arXiv. https://doi.org/10.48550/arXiv.2005.11401

Liang, H., Yin, L., Wu, G., Li, Y., Yi, Q., & Wang, L. (2025). *LeakGuard: Detecting Memory Leaks Accurately and Scalably* (No. arXiv:2504.04422). arXiv. https://doi.org/10.48550/arXiv.2504.04422

Liu, J., Kong, Z., Dong, P., Shen, X., Zhao, P., Tang, H., Yuan, G., Niu, W., Zhang, W., Lin, X., Huang, D., & Wang, Y. (2025). RoRA: Efficient Fine-Tuning of LLM with Reliability Optimization for Rank Adaptation. *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1–5. https://doi.org/10.1109/ICASSP49660.2025.10889613

Liu, Z., Lyn, J., Zhu, W., & Tian, X. (2024). ALoRA: Allocating Low-Rank Adaptation for Fine-tuning Large Language Models. *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 622–641. https://doi.org/10.18653/v1/2024.naacl-long.35

Lv, K., Yang, Y., Liu, T., Gao, Q., Guo, Q., & Qiu, X. (2024). *Full Parameter Fine-tuning for Large Language Models with Limited Resources* (No. arXiv:2306.09782). arXiv. https://doi.org/10.48550/arXiv.2306.09782

Mohammed, N., Lal, A., Rastogi, A., Roy, S., & Sharma, R. (2024). *Enabling Memory Safety of C Programs using LLMs* (No. arXiv:2404.01096). arXiv. https://doi.org/10.48550/arXiv.2404.01096

Pathak, A., Shree, O., Agarwal, M., Sarkar, S. D., & Tiwary, A. (2023). Performance Analysis of LoRA Finetuning Llama-2. *2023 7th International Conference on Electronics, Materials Engineering & Nano-Technology (IEMENTech)*, 1–4. https://doi.org/10.1109/IEMENTech60402.2023.10423400

Poldrack, R. A., Lu, T., & Beguš, G. (2023). *AI-assisted coding: Experiments with GPT-4* (No. arXiv:2304.13187). arXiv. https://doi.org/10.48550/arXiv.2304.13187

Poornima, S. (2013). *Detection of Precise C/C++ Memory Leakage by diagnosing Heap dumps using Inter procedural Flow Analysis statistics*. *2*(10).

Ramesh, R., M, A. T. R., Reddy, H. V., & N, S. V. (2024). Fine-Tuning Large Language Models for Task Specific Data. *2024 2nd International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, 1–6. https://doi.org/10.1109/ICNEWS60873.2024.10730913

Sarkar, S., Kushwaha, S. P., Sharma, V., Mishra, N., & Alkhayyat, A. (2024). A Novel LLM enabled Code Snippet Generation Framework. *2024 International Conference on Intelligent & Innovative Practices in Engineering & Management (IIPEM)*, 1–5. https://doi.org/10.1109/IIPEM62726.2024.10925748

Shuttleworth, R., Andreas, J., Torralba, A., & Sharma, P. (2024a). *LoRA vs Full Fine-tuning: An Illusion of Equivalence* (No. arXiv:2410.21228). arXiv. https://doi.org/10.48550/arXiv.2410.21228

Shuttleworth, R., Andreas, J., Torralba, A., & Sharma, P. (2024b). *LoRA vs Full Fine-tuning: An Illusion of Equivalence* (No. arXiv:2410.21228). arXiv. https://doi.org/10.48550/arXiv.2410.21228

Sjoeberg, D. I. K., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, *31*(9), 733–753. https://doi.org/10.1109/TSE.2005.97

Sokolova, M., & Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, *45*(4), 427–437. https://doi.org/10.1016/j.ipm.2009.03.002

Soudani, H., Kanoulas, E., & Hasibi, F. (2024). Fine Tuning vs. Retrieval Augmented Generation for Less Popular Knowledge. *Proceedings of the 2024 Annual*

International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region, 12–22. https://doi.org/10.1145/3673791.3698415

Sui, Y., Ye, D., & Xue, J. (2012). Static memory leak detection using full-sparse value-flow analysis. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 254–264. https://doi.org/10.1145/2338965.2336784

Swamy Prasadarao Velaga. (2020). AI-ASSISTED CODE GENERATION AND OPTIMIZATION: LEVERAGING MACHINE LEARNING TO ENHANCE SOFTWARE DEVELOPMENT PROCESSES. *International Journal of Innovations in Engineering Research and Technology*, *7*(09), 177–186. https://doi.org/10.26662/ijiert.v7i09.pp177-186

Tural, B., Örpek, Z., & Destan, Z. (2024). Retrieval-Augmented Generation (RAG) and LLM Integration. *2024 8th International Symposium on Innovative Approaches in Smart Technologies (ISAS)*, 1–5. https://doi.org/10.1109/ISAS64331.2024.10845308

Tvarožek, R., & Haffner, O. (2025). What If LLM Doesn't Know. *2025 Cybernetics & Informatics (K&I)*, 1–6. https://doi.org/10.1109/KI64036.2025.10916478

Wang, H., Ping, B., Wang, S., Han, X., Chen, Y., Liu, Z., & Sun, M. (2024). *LoRA-Flow: Dynamic LoRA Fusion for Large Language Models in Generative Tasks* (No. arXiv:2402.11455). arXiv. https://doi.org/10.48550/arXiv.2402.11455

Wei, J., Wu, S., Liu, R., Ying, X., Shang, J., & Tao, F. (2025). *Tuning LLMs by RAG Principles: Towards LLM-native Memory* (No. arXiv:2503.16071). arXiv. https://doi.org/10.48550/arXiv.2503.16071

Xu, Z., Kremenek, T., & Zhang, J. (2010). A Memory Model for Static Analysis of C Programs. In T. Margaria & B. Steffen (Eds.), *Leveraging Applications of Formal Methods, Verification, and Validation* (Vol. 6415, pp. 535–548). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-16558-0_44

Yaozong, X., Xuebin, S., Shuhua, Z., Qiujun, Z., & Weinan, J. (2023). Static Analysis Method of C Code Based on Model Checking and Defect Pattern Matching. *2023 IEEE 5th International Conference on Power, Intelligent Computing and Systems (ICPICS)*, 567–573. https://doi.org/10.1109/ICPICS58376.2023.10235566

Younan, Y., Joosen, W., Piessens, F., & Van Den Eynden, H. (2010). Improving Memory Management Security for C and C++: *International Journal of Secure Software Engineering*, *1*(2), 57–82. https://doi.org/10.4018/jsse.2010040104

Yu, B., Tian, C., Zhang, N., Duan, Z., & Du, H. (2021). A dynamic approach to detecting, eliminating and fixing memory leaks. *Journal of Combinatorial Optimization*, *42*. https://doi.org/10.1007/s10878-019-00398-x

Yuan, L., Zhou, S., Pan, P., & Wang, Z. (2022). MLD: An Intelligent Memory Leak Detection Scheme Based on Defect Modes in Software. *Entropy*, *24*(7), 947. https://doi.org/10.3390/e24070947

Zaken, E. B., Ravfogel, S., & Goldberg, Y. (2022). *BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models* (No. arXiv:2106.10199). arXiv. https://doi.org/10.48550/arXiv.2106.10199

Zhang, T., Patil, S. G., Jain, N., Shen, S., Zaharia, M., Stoica, I., & Gonzalez, J. E. (2024). *RAFT: Adapting Language Model to Domain Specific RAG* (No. arXiv:2403.10131). arXiv. https://doi.org/10.48550/arXiv.2403.10131

Zhao, P., Zhang, H., Yu, Q., Wang, Z., Geng, Y., Fu, F., Yang, L., Zhang, W., Jiang, J., & Cui, B. (2024). *Retrieval-Augmented Generation for AI-Generated Content: A Survey* (No. arXiv:2402.19473). arXiv. https://doi.org/10.48550/arXiv.2402.19473

Zhiqiang Liu, Bo Xu, Dong Liang, Chang Liu, Zejun Jiang, & Chenglie Du. (2015a). Semantics-based memory leak detection for C programs. *2015 12th International*

Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2283–2287. https://doi.org/10.1109/FSKD.2015.7382308

Zhiqiang Liu, Bo Xu, Dong Liang, Chang Liu, Zejun Jiang, & Chenglie Du. (2015b). Semantics-based memory leak detection for C programs. *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2283–2287. https://doi.org/10.1109/FSKD.2015.7382308