الجمهورية الجزائرية الديمقراطية الشعبية
The People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research
جامعة محمد بوضياف بالمسيلة
University Mohamed Boudiaf of M'sila

كلية الرياضيات والإعلام الآلي
Faculty of Mathematics and Informatics

قسم الإعلام الآلي
Department of Computer Science

Domain: Mathematics and Computer Science

Thesis Presented to Fulfill the Partial Requirement
for **Master's Degree** in Computer Science

Specialty: Information System and Software Engineering

Prepared By: Amina MEKIDECHE, Amani DJAIDJA

Supervised By:

Hichem DEBBI

ENTITLED

Automated Program Repair using Large Language Models (LLMs)

Jury Members

Abdelbaki BOUGUERRA	President
Hichem DEBBI	Supervisor
Marouane KIHAL	Examiner

Academic Year 2024/2025

Dedications

في لحظةِ تأمُّلٍ بين عجزِ الإنسانِ وعظمةِ التدبيرِ الإلهيّ، لجأتُ إلى رُكنٍ لا يُخذَلُ، مستودِعةً أمري بين يديه... فكان من رحمتهِ أن ييسّر لي دربًا أجهله وهو يعلمه. فله الحمدُ أولًا وآخرًا.

إلى من آمنا بي، وساندانا في كلّ مسعى، إلى من أسندا قلبي حين مال، وشدّا على يديّ حين خارتا، وسقيا قلبي من فيض دعائهما جهرًا وغيبًا، وعلّمانا أن العلم ضياءٌ لا ينطفئ، وأن السعي في دروبه عبادة...

والدي، الذي كان ولا يزال نجمًا يهتدي به قلبي في ظلمات الطريق، قدوتي الأولى ومصدر عزيمتي، شكرًا لك بحجم السماء.

والدتي، نبع الحنان وسند الروح، دعاؤكِ كان الزاد في لحظات التعب، جزاكِ الله عني خيرًا، وأثقل ميزان حسناتكِ في الدنيا والآخرة.

إلى إخوتي وعائلتي الكبيرة، شركاء الرحلة، سندي في دروب الحياة، وجودكم أمانٌ لا يُقدَّر.

إلى كل من كان له في مسيرتي بصمة، وسندني بدعاءٍ أو بكلمة، شكرًا.

– آمنة مكيدش –

إلى فلسطين...

إلى الأرض التي تلد الشهداء وتُرضع أطفالها الكبرياء،

إلى الزيتون الذي لا ينكسر مهما تعاقبت عليه الفصول،

إلى جراحك التي نزفت نورًا، وإلى مقاومتك التي علّمتنا معنى أن نكون أحرارًا،

إليكِ يا قدس، يا غزّة، يا نابلس وجنين...

أهديكِ تخرّجي، لا كإنجاز شخصي، بل كبيعة وفاء، وعهد لا يخون.

وإلى من سلكوا درب الثبات،

إلى كل من ناصر القضية بقلبه، بصوته، بموقفه،

إلى أولئك الذين حملوا سلاح المقاطعة، ووقفوا على ثغر الوعي حين نام كثيرون،

إليكم أهدي هذا التخرج... لأنكم المعنى الحقيقي للثبات والكرامة.

ثم إلى عائلتي الحبيبة...

إلى أمي، نبع الصبر والرحمة،

إلى أبي، ضوئي ونوري أينما سعيت،

وإلى أحبتي، الذين شاركوني الدرب والنبض،

إلى كل من كانوا لي وطنًا وسندًا

أهديكم هذا التتويج، فهو منكم ولكم، كما هو مني.

في هذا التخرّج... لا أرفع قلمي فخرًا فقط،

بل أرفعه عهدًا بأن نبقى حيث يجب أن نكون... مع الحق، مع فلسطين،

حتى يعود الأذان حرًّا، وتعود الأرض لأحضان أهلها.

– أماني جعيجع –

Acknowledgments

First and foremost, we thank Allah, the Almighty, for granting us the strength and patience to complete this thesis.

We are deeply grateful to our supervisor, **Professor Hichem DEBBI**, who has been responsible for most of what we have learned during this process; guiding us, supporting us at all times, and doing so without hesitation. His dedication and encouragement were instrumental to our progress.

Our sincere thanks also go to the jury members for their time, thoughtful evaluation, and constructive feedback, which greatly enriched our work.

We extend our appreciation to all the professors and the Computer Science Department for their continuous support and for providing us with a strong academic foundation.

Finally, we would like to express our heartfelt thanks to our families, friends, and all those who contributed, directly or indirectly, to the completion of this work. Their encouragement and unwavering support were essential to our success.

Abstract

This work proposes an approach to APR by integrating a fine-tuned CodeLLaMA-7b model with a GraphRAG-based retrieval framework. We first applied LoRA to fine-tune the CodeLLaMA model using the domain-specific RepairLLaMA dataset. To enhance contextual awareness, we built a graph-based retriever that combines CodeBERT-generated embeddings with structural relationships between buggy and fixed code pairs from the Defects4J dataset. Our system prioritizes relevant repair examples and enables efficient retrieval of code context. Additionally, we developed a simple web interface to provide real-time bug fix suggestions, demonstrating the practical applicability of our pipeline.

Key words: Large Language Models, Automated Program Repair, GraphRAG, Fine-tuning.

الملخص

يقترح هذا العمل نهجًا متقدمًا لإصلاح البرامج التلقائي (APR) من خلال دمج نموذج CodeLLaMA-7b مضبوط بدقة مع إطار استرجاع قائم على GraphRAG. في البداية، قمنا بتطبيق تقنية LoRA لضبط نموذج CodeLLaMA باستخدام مجموعة بيانات RepairLLaMA المتخصصة في مجال إصلاح الأكواد. ولتعزيز الوعي بالسياق البرمجي، أنشأنا مسترجعًا قائمًا على الرسوم البيانية يجمع بين تمثيلات CodeBERT الشعاعية والعلاقات البنيوية بين أزواج الأكواد الخاطئة والمصححة المستخرجة من مجموعة بيانات Defects4J. يتيح نظامنا استرجاع أمثلة إصلاح ذات صلة بشكل فعّال، مع إعطاء أولوية للأكواد المصححة. بالإضافة إلى ذلك، قمنا بتطوير واجهة ويب بسيطة لتوفير اقتراحات إصلاح الأخطاء البرمجية في الزمن الحقيقي، مما يبرز قابلية تطبيق هذا النظام عمليًا.

الكلمات المفتاحية : نماذج اللغة الكبيرة، الإصلاح التلقائي للبرمجيات، GraphRAG، الضبط الدقيق.

Résumé

Ce travail propose une approche de réparation automatique de code en combinant un modèle CodeLLaMA-7b affiné avec LoRA et un système de récupération GraphRAG. En utilisant les données de RepairLLaMA et Defects4J, nous intégrons des embeddings CodeBERT et des relations entre codes erronés et corrigés. Une interface web simple permet de suggérer des correctifs en temps réel, montrant l'efficacité pratique de notre méthode.

Mots Clés : Grand modèle de langage, Réparation Automatique des Programmes, GraphRAG, Réglage fin.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence

ML Machine Learning

DL Deep Learning

NN Neural Network

NLP Natural Language Processing

LLMs Large Language Models

RAG Retrieval-Augmented Generation

APR Automated Program Repair

LoRA Low-Rank Adaptation

OSS Open-Source Software

General Introduction

Software has become an integral part of modern life. From mobile phones, computers and smart devices, programs run silently in the background to help us work, connect, and live more efficiently. However, as software systems grow in complexity, ensuring code correctness becomes increasingly difficult. These bugs can cause software to crash and behave unexpectedly.

An essential component of software development is debugging, which is a crucial part of software development. However, it is often a slow and challenging task that requires deep expertise and patience. While many tools exist to support developers in this process, they often come with limitations: some are difficult to use, others are too slow, and many fail to provide intelligent insights into what is wrong or how to fix it.

Artificial intelligence developments in recent years, especially the creation of Large Language Models, have brought promising new approaches to these problems. LLMs are strong artificial intelligence systems that can comprehend and generate natural language and programming code. They have shown remarkable potential in reading code, identifying errors, and even suggesting corrections, which can greatly assist developers in their debugging tasks.

Objective

This thesis investigates the use of Large Language Models for automated program repair, aiming to detect and correct coding errors without human intervention. To enhance this process, it incorporates Graph-RAG. By leveraging these contextual references, Graph-RAG supports the LLM in better understanding the underlying issues and generating more accurate and context-aware repair suggestions.

Manuscript Organisation

This thesis is organized into four chapters:

Chapter 1 introduces the foundational concepts of Deep Learning and Large Language Models, setting the technical groundwork.

Chapter 2 provides an overview of software bugs and the traditional approaches to debugging, including their current limitations.

Chapter 3 presents the use of LLMs for automated program repair, explaining how these models can assist in the debugging process.

Chapter 4 details the implementation of our proposed system, combining LLMs with Graph-RAG, and discusses the results of testing our tool on real-world code.

CHAPTER 1

Deep Learning and Large Language Models

1. Introduction

Deep learning revolutionized Artificial Intelligence, especially Natural Language Processing, with Large Language Models. Inspired by biological neural networks and driven by architectures like Transformers, the models can be pre-trained on massive datasets and fine-tuned for a specific task. This chapter explores deep learning, architectures of NLP neural networks, the Transformer model, and the emergence of LLMs, focusing on mechanisms that make them a vital part of modern AI.

2. Deep Learning

Deep Learning, a subfield of Machine Learning and Artificial Intelligence, uses artificial neural networks to autonomously learn patterns and make predictions or decisions without explicit rule-based programming. Inspired by the brain's neural architecture, DL adjusts internal parameters to improve accuracy over time. It is applied across domains such as natural language processing, computer vision, healthcare, and cybersecurity.

2.1. Intersection between Deep Learning and Artificial Intelligence

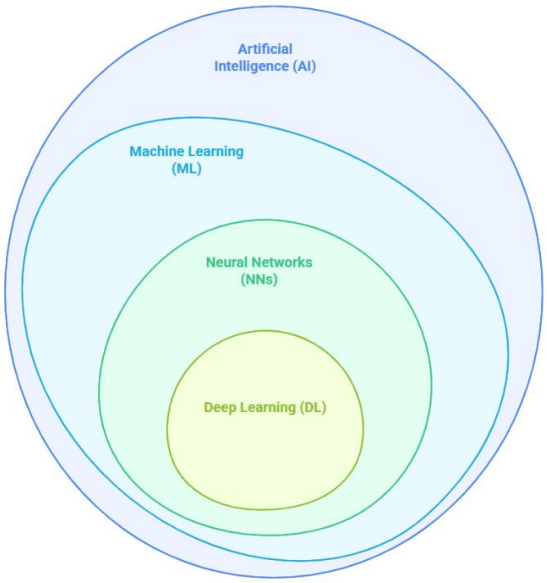

Figure 1.1: Relationship between AI, ML, NN and DL

Deep Learning is a subset of Machine Learning, which falls under the broader field of Artificial Intelligence. AI refers to systems that mimic human cognitive functions, including reasoning, problem-solving, and learning. Where ML enables these systems to improve through data-driven prediction and error reduction, without relying on explicitly programmed rules. DL advances this by employing neural networks with multiple layers, enabling the extraction of complex patterns from large datasets. These deep architectures form the core of modern DL methods.

DL represents a major advancement in AI, enhancing its ability to model and automate tasks with human-like adaptability and intelligence [1].

2.2. Difference between Deep Learning and Neural Networks

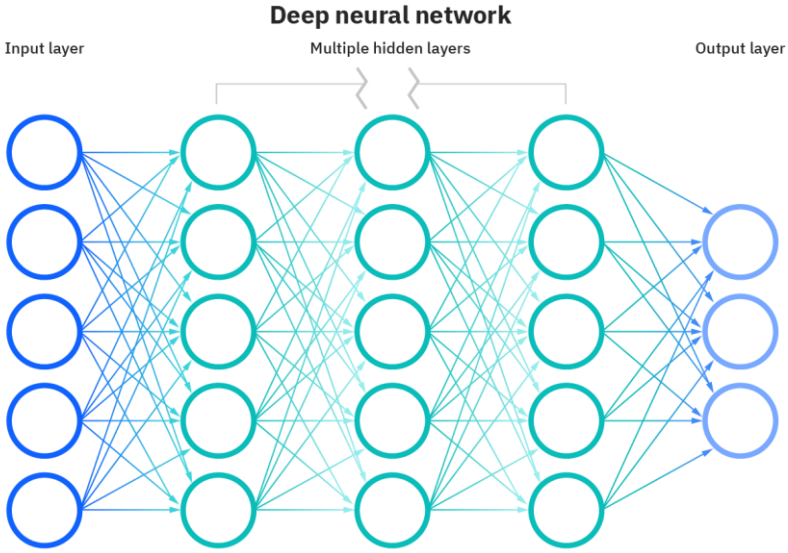

Figure 1.2: Typical Architecture of Deep Learning Neural Network

Neural networks, or Artificial Neural Networks are a subset of ML that originated from the study of the human brain. ANNs attempt to accomplish this biological effectiveness through simulating connected processing units (neurons) that adapt and learn from experience. These models are structured in a hierarchical manner, consisting of an input layer, one or multiple hidden layers, and an output layer.

In contrast, DL is a specialized subset of neural networks. The “deep” in DL refers to the number of hidden layers that contribute to the model [2], which enables more effective modeling of complex and hierarchical data representations.

2.3. Neural Network Architectures for Natural Language Processing (NLP)

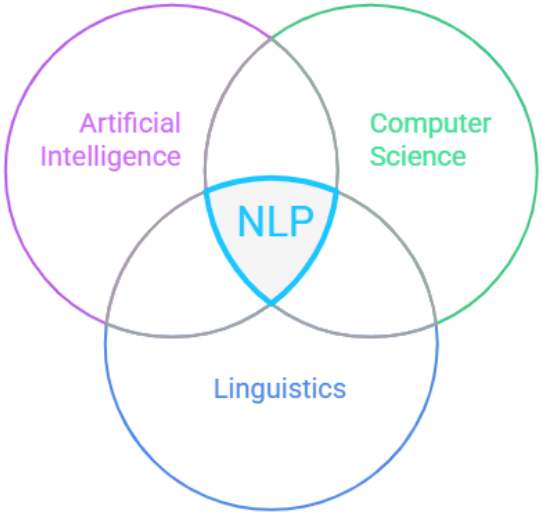

Figure 1.3: Intersection between NLP, AI, Computer Science and Linguistics

Natural Language Processing is a subfield of artificial intelligence, computer science, and linguistics that focuses on interactions between computers and human language [3].It enables computers to recognize, understand, and generate text and speech. NLP combines rule-based language modeling with statistical methods, machine learning, and deep learning.

Various Neural Network architectures are used in NLP, each serving different purposes. Some of the most commonly used architectures include:

2.3.1. Convolutional Neural Networks (CNNs)

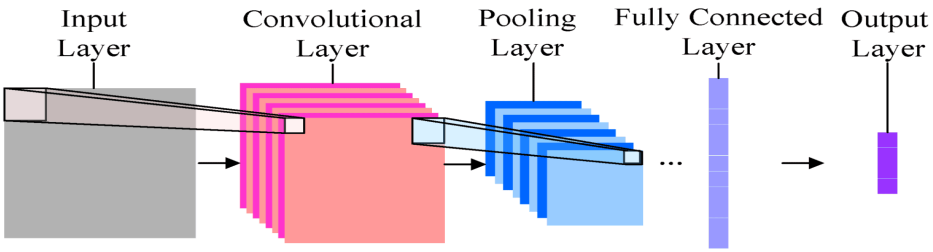

Figure 1.4: Convolutional Neural Networks Layers Architecture

CNNs, originally developed for image processing tasks, have been adapted to implement NLP applications like text classification and sentiment analysis. Their architecture includes convolutional layers that extract local features from input sequences, followed by pooling layers that reduce dimensionality. The resulting features are then passed through one or more fully connected layers for classification [4].

2.3.2. Recurrent Neural Networks (RNNs)

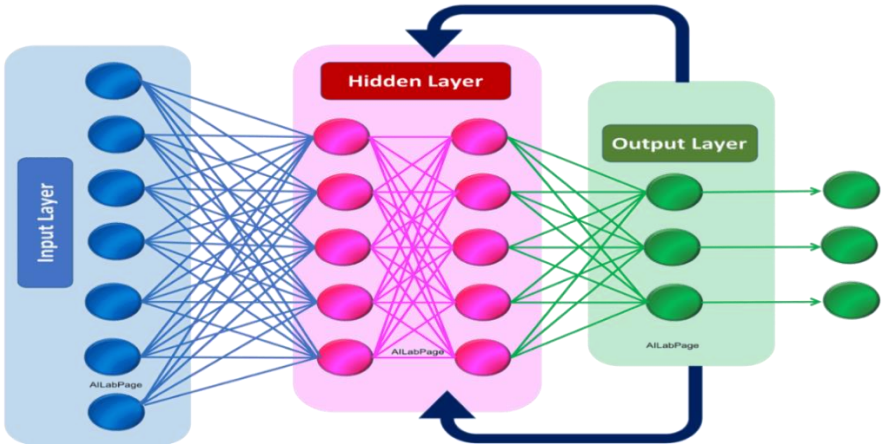

Figure 1.5: Recurrent Neural Networks Architecture

RNNs are another type of neural network architecture commonly used in NLP tasks such as language modeling, machine translation, and sentiment analysis. They use recurrent neurons to retain information across time steps, enabling a form of memory [5]. Variants include the basic RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU), which differ in how they update hidden states and regulate information flow.

2.3.3. Transformer Networks

Transformers are a neural network architecture that has significantly improved performance across a wide range of NLP tasks. Introduced in "*Attention Is All You Need*" by Vaswani et al. [6], they rely on a self-attention mechanism, that enables the model to weigh different parts of the input sequence when processing each token.

A key advantage of transformers is their capacity for parallelized training on large datasets, making them highly scalable.

2.3.4. Graph Neural Networks (GNNs)

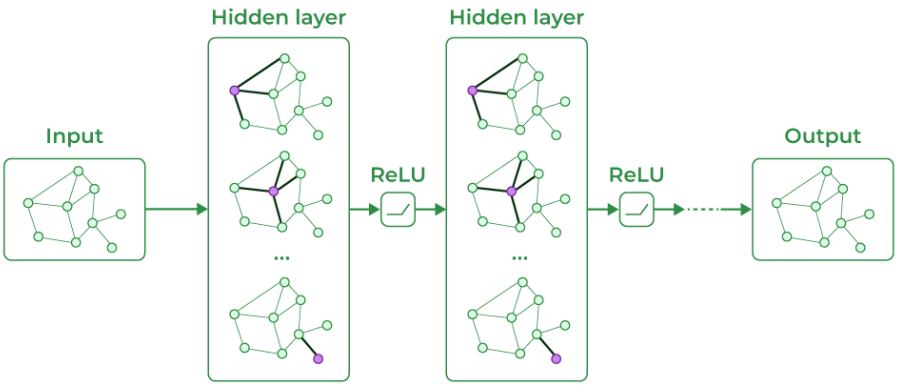

Figure 1.6: Flow Chart of Graph Neural Network

GNNs are a special type of neural network useful for solving challenging problems using graphs that represent complex data structures [7]. Developed in response to the need for models that handle unstructured and relational data, GNNs extend traditional neural networks by incorporating the topology and features of graphs. They are effective in capturing dependencies and patterns within complex structures.

3. Transformer Architecture

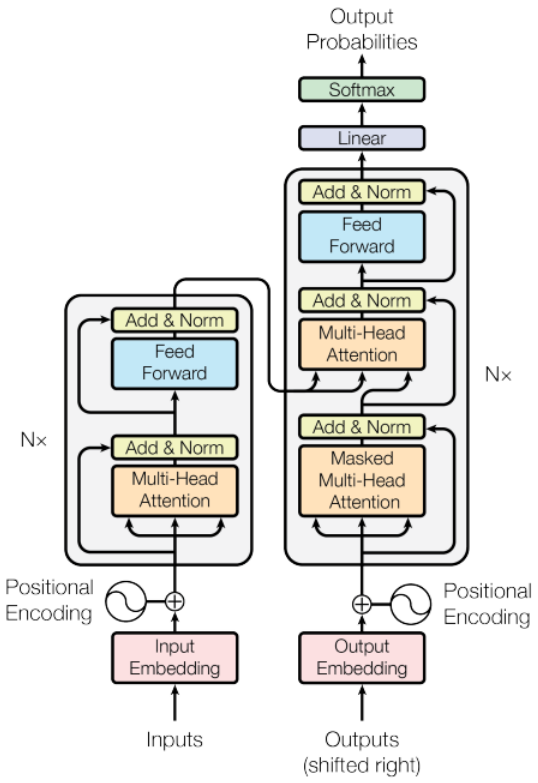

Figure 1.7: Transformer Architecture [6]

The Transformer architecture consists of two main parts as indicated in Figure 1.7, each comprising multiple layers of self-attention and feedforward networks. Key elements include the input layer, encoder and decoder stacks, attention mechanisms, position-wise feedforward networks, embeddings, softmax layer, and positional encoding [6], which allow the model to learn word relations and process information efficiently.

3.1. Input Layer

- **Tokenization:** Tokenization is the process of converting input data into corresponding tokens, which machines can understand. In Natural Language Processing, it typically means splitting sentences into words or subwords [8]. These tokens are then mapped to numerical IDs using a lookup table, forming the input for Transformer models.

- **Padding and Truncation:** They are essential steps in preparing text for transformer models. Since models process inputs in batches, all sequences must be the same length. Padding adds special tokens (e.g., [PAD]) to shorter sequences to match the longest one [9], while truncation shortens sequences that exceed a predefined limit. These techniques ensure uniform input size, enabling efficient computation and consistent model training.

3.2. Encoder and Decoder Stacks

- **Encoder:** The encoder designed to understand the entire sentence all at once, instead of processing each token separately like older models did. This holistic understanding helps it capture the relationships between tokens, resulting in deeper and more accurate representations. That's what makes it powerful for tasks like translation and text generation.

- **Decoder:** The decoder's main job is to generate text sequences in a logical and fluent manner, based on what the encoder understood. Its design includes two multi-head attention layers; one to track what's being written and another to focus on the encoder's output. With additional layers for refinement and stabilization, the decoder ensures the generated text is coherent and contextually accurate.

3.3. Attention Mechanism

Attention is a fundamental component of the Transformer model, enabling it to decide which parts of a sentence to focus on while performing tasks like translation or summarization. Instead of treating all words equally, attention allows the model to understand which words matter more in the current context. This mechanism leads to smarter, more context-aware decisions.

- **Self-Attention:** Self-attention is a key feature where each token in a sentence understands its meaning by looking at all other tokens. Each token is turned into an embedding, the model generates Query (Q), Key (K), and Value (V) vectors for each token to compute relevance scores. These scores weight the values, producing context-aware representations for each token.

- **Multi-Head Attention:** Multi-head attention takes the idea of self-attention further by allowing the model to look at the sentence from different perspectives at once. It splits each token embedding into parts and processes them in parallel through multiple attention heads. Each head captures distinct contextual patterns, enhancing the model's overall understanding.

3.4. Position-wise Feed-Forward Networks (FFN)

Each layer in the encoder and decoder includes a position-wise feed-forward network. The FFN fine-tunes each token independently by applying two linear transformations with a ReLU activation in between. The FFN expands the input from 512 to 2048 dimensions, then reduces it back, enabling deeper feature extraction with layer-specific parameters.

3.5. Embeddings

Before a Transformer can understand language, it needs to turn words into numbers it can work with. This is where embeddings come in. Each token is converted into a vector of fixed dimension that captures its meaning in context. Similar words receive similar vectors, aiding pattern recognition. The model learns these embeddings during training and uses the same ones for both input and output tokens to reduce the number of parameters and improve efficiency.

3.6. Softmax Layer

At the end of the Transformer's processing, it needs to assigns scores to all vocabulary words. The model first generates a score for every word in the vocabulary. Then, the softmax function converts these scores into probabilities, selecting the most likely next word.

Weight sharing between input embeddings, output embeddings, and the pre-softmax layer reduces parameters and enhances efficiency.

3.7. Positional Encoding

Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique vector. These vectors, added to word embeddings, enable the model to distinguish word order.

A common approach uses sine and cosine functions based on position, allowing the model to capture relative positions and generalize to longer sequences without recurrence.

4. Large Language Models (LLMs)

Large Language Models are a type of generative AI systems that can understand and generate human-like text, they are based on transformer architectures. These models are pre-trained on massive amounts of text-based resources, allowing them to understand complex language structures, contextual relationships, and syntactic patterns [10].

Because of this, LLMs are capable of performing various NLP tasks such as text generation, summarization, translation. With billions of parameters, these models can reason, generate content, and adapt to different styles and topics.

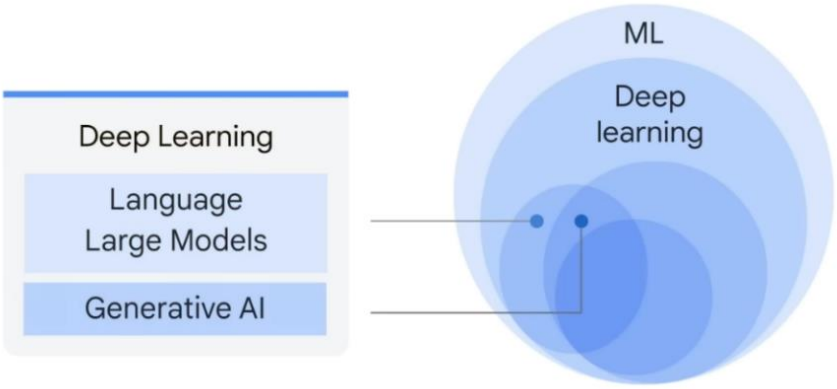

Figure 1.8: Relationship between ML, DL, Generative AI and LLMs

5. LLMs Adaptation Stages

The main adaptation stages of LLMs include pre-training, fine-tuning for specific downstream tasks, and practical utilization. Additionally, alignment-tuning is highlighted as the process of adapting the model to align with human preferences. This process is illustrated in the accompanying figure 1.9.

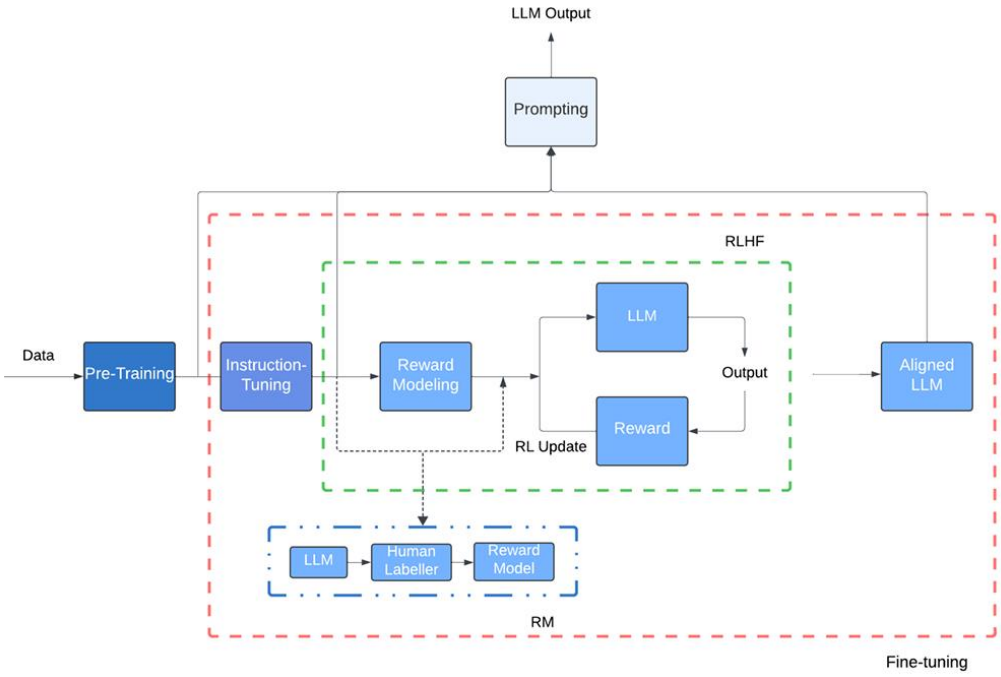

Figure 1.9: Basic Flow Diagram present the Stages of LLMs From Pre-Training to Prompting [11]

5.1. Pre-Training

In the very first stage, the model is trained in a self-supervised manner on a large corpus to predict the next tokens given the input [11] on vast amounts of data. This stage equips the model with general language understanding by exposing it to diverse data types. It involves large-scale distributed training, adaptive learning rates, and modern optimizers.

5.2. Fine-Tuning

Fine-tuning is the process of taking a pre-trained model and further training it on a domain-specific dataset [12]. This approach reduces computational expenses and the ability to leverage cutting-edge models without the necessity of building one from the ground up. It enables the creation of models that are more accurate and context-aware for specific tasks.

5.2.1. Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning is methodology that has emerged as a solution to the significant challenges posed by adapting LLMs with billions of parameters to specific downstream tasks [13]. The core idea of PEFT is to train only a small number of additional task-specific parameters while freezing the majority of the pre-trained model's architecture. This makes it possible to adapt large models quickly and efficiently, without the heavy computational cost usually required. Thanks to this approach, PEFT significantly cuts down on memory usage, training time, and storage needs, while still delivering strong performance.

5.2.2. Parameter-Efficient Fine-Tuning Techniques

There are various PEFT techniques and algorithms available, each with its relative advantages and specializations, including:

- **Adapters:** Adapters are one of the first PEFT techniques to be applied to NLP models. These methods aim to adapt pre-trained models to specific tasks by adding new trainable parameters.

- **LoRA (Low-Rank Adaptation):**LoRA is a fine-tuning method that adapts large pre-trained models efficiently by freezing the pre-trained model weights and injecting trainable rank decomposition matrices into each layer of the Transformer architecture. This approach significantly reduces the number of trainable parameters required for downstream tasks[14] and lowers GPU memory usage, making fine-tuning feasible even on a single GPU. LoRA enables modularity, allowing many small LoRA modules to be built for different tasks, which can be swapped without retraining the full model. During deployment, the trainable matrices can be merged with the frozen weights, ensuring no added inference latency.

- **QLoRA (Quantized Low-Rank Adaptation):** An advanced version of LoRA that aims to significantly reduce memory requirements enough to fine-tune large-parameter models on a single GPU with limited memory capacity, while maintaining the performance of full 16-bit fine-tuning [15]. It achieves this by integrating quantization into the pre-trained model before applying LoRA.

5.3. Prompting

Prompting is a method to query trained LLMs for generating responses [11]. Instead of retraining or modifying the model, prompting works by tapping into the model's existing knowledge and capabilities through well-crafted input.

6. Types of LLMs

6.1. Encoder-Based Models

Encoder-only models are transformer-based architectures optimized for understanding and representing input text rather than generating new content. Notable examples include BERT (Bidirectional Encoder Representations from Transformers) [16] and RoBERTa [17]. These models are especially effective for natural language understanding (NLU) tasks that require deep comprehension of input text. Tasks such as text classification, named entity recognition, and question answering benefit from the models' ability to capture context-rich representations.

6.2. Decoder-Based Models

Decoder-only models optimized for autoregressive text generation. They predict the next token based on previous ones, using masked self-attention to preserve the left-to-right sequence makes them especially effective for tasks such as generating content, completing text, or even writing code. Notable models such as the GPT (Generative Pre-trained Transformer) models, including GPT-2, GPT-3 [18], and the more recent GPT-4, and LLaMA [19] have been trained on massive amounts of data, allowing them to perform a variety of language tasks with minimal instruction.

6.3. Encoder-Decoder Models

These models combine both approaches, input encoding and output decoding mechanisms, making them well-suited for sequence-to-sequence tasks such as summarization, machine translation, and question answering. Newer models like BART [20] and T5 (Text-to-Text Transfer Transformer) [21] build on this approach with smarter training methods like cleaning up noisy text or treating every task as a text-to-text problem.

7. Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation is a novel approach introduced in the paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" [22]. It enhances the capabilities of LLMs by combining retrieval and generation. In RAG, a retriever component searches a large database or knowledge base to find relevant information based on the input query.

The retrieved data is then fed into a generative model, allowing it to produce more accurate, specific, and contextually relevant responses. By supplementing the model's parametric memory with external knowledge, RAG significantly improves performance on knowledge-intensive tasks and laid the groundwork for future research in integrating retrieval mechanisms with generative AI.

8. Core Stages in RAG

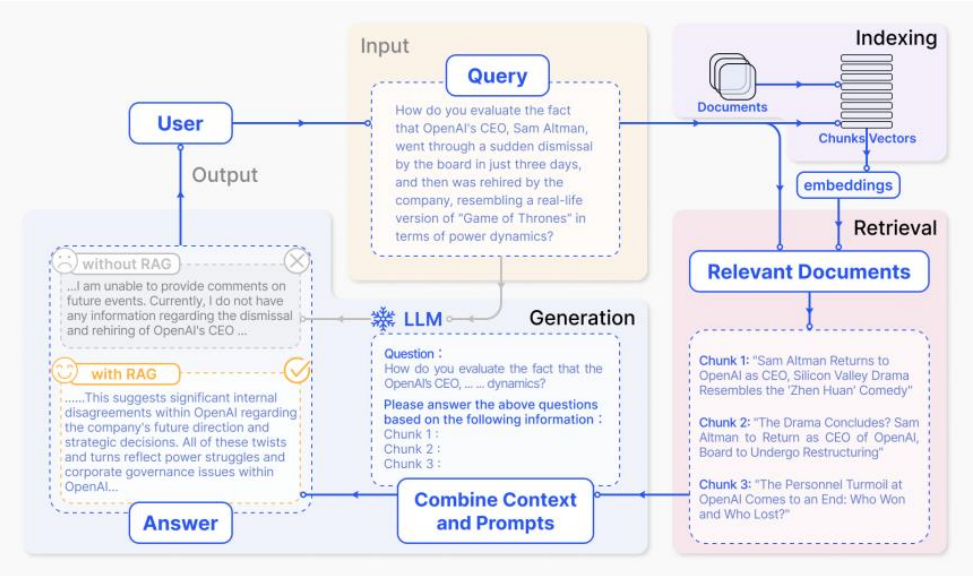

Figure 1.10: A Representative Instance of RAG Process Applied to Question Answering [23]

Instead of making a language model guess answers from memory, RAG assists it by providing real information from external sources. RAG operates through three main steps [23], as shown in Figure 1.10, including:

8.1. Indexing

The indexing phase begins by extracts and cleans data from diverse sources such as PDFs, HTML pages, Word documents, standardizing all content into plain text format. Given the context length limitations of language models, the text is segmented into smaller, coherent chunks. Each chunk is encoded into a dense vector using an embedding model and stored in a vector database for efficient similarity-based retrieval.

8.2. Retrieval

Upon receiving a query, the system encodes it into a dense vector using the same embedding model utilized during the indexing phase. A similarity search retrieves the top-k chunks from the vector database based on semantic alignment. These chunks supplement the language

model's internal memory with external, context-specific information, guiding accurate and informed generation.

8.3. Generation

The user query and the retrieved context chunks are concatenated into a structured prompt and input into the language model. The LLM integrates external knowledge with its internal memory to generate a coherent and contextually grounded response. This generation phase combines retrieval-based context with model reasoning to produce accurate, high-quality outputs.

9. Business applications of LLMs

9.1. Customer Service Applications

LLMs have transformed customer service by enabling automated and intelligent interactions, AI-powered chatbots now handle tasks such as answering FAQs, processing transactions, and delivering personalized support [24]. By employing LLMs, companies enhance efficiency, reduce operational costs, and improve customer satisfaction while easing the workload on human agents.

9.2. Content Creation

LLMs enable the generation of informative, and relevant content at scale in various formats, including blog posts, articles, product descriptions, and social media updates. saving companies time and resources. LLMs can also recommend trending topics and pertinent keywords by examining audience interests and existing content, which helps to improve visibility and optimize content for search engines.

9.3. Software Development

The implementation of LLMs in software development has resulted in a major overhaul of business applications within the field, enhancing their performance and capabilities including code generation and vulnerability detection [10].

LLM-based agents now perform actions and make decisions autonomously, continuously learning and adapting. These agents enhance traditional LLM functions, boosting productivity and fostering innovation in complex development tasks. Their use accelerates workflows and supports agile adaptation to evolving technological demands.

10. Overview of Popular LLMs

10.1. General-Purpose Large Language Models

These models are built to handle a variety of tasks, including natural language interpretation, reasoning, summarization, translation, and the generation of creative content. They are trained on a vast range of internet-scale data. They can be fine-tuned or prompted to carry out different tasks and are not restricted to any one domain.

10.1.1. Overview of General-Purpose LLMs

Model	Institution	Architecture	Context Window	Parameters	Tokens	Release	Open Source
GPT-3	OpenAI	Decoder only	128k	1.76T	13T	2023	No
Gemini	Google DeepMind	Encoder-Decoder	2M	1.8B, 3.25B	Not officially disclosed	2023	No
Claude 3	Anthropic	Decoder only	200k	Not officially disclosed	Not officially disclosed	2024	No
LLaMA 3	Meta AI	Decoder only	128K	8B, 70B,	+15T	2024	Yes
DeepSeek	DeepSeek	Decoder only	32K	1.3B, 6.7B, 33B	2T	2024	Yes

Table 1.1: Overview of General-Purpose Large Language Models

Table 1.1 provides a comparative analysis overview of general-purpose LLMs, focusing on institutional origin, architecture of the model, context window size, parameter count, volume of training tokens, release date, and open-source availability. The information compiled in this table is primarily based on the comprehensive survey by Li et al. (2024) [25].

10.2. Code-Specific Large Language Models

To perform well in specialized tasks, such as code generation, bugs fixing and mathematical reasoning, these models are fine-tuned or pre-trained using domain-specific datasets. In their limited scope, they usually perform better than general-purpose models.

10.2.1. Overview of Code-Specific LLMs

Model	Institution	Architectures	Size	Vocabulary	Context Window	Open Source
CodeGPT	Microsoft	Decoder-only	124M	50K	1024	Yes

Codex	OpenAI	Decoder-only	12M, 25M, 42M, 85M, 300M,679M, 2.5B, 12B	Not officially disclosed	4096	No
CodeT5	Salesforce	Encoder-Decoder	60M, 220M, 770M	32K	512+256	Yes
Code Llama	Meta	Decoder-only	7B, 13B, 4B	32K	16,384	Yes
CodeGemma	Google	Decoder-only	2B, 7B	25.6k	8192	Yes
AlphaCode	DeepMind	Encoder-Decoder	284M, 1.1B, 2.8B,8.7B, 41.1B	8K	1536+768	No
Code-Qwen	Qwen Group	Decoder-only	7B	92K	65,536	Yes
DeepSeek-Coder	DeepSeek	Decoder-only	1.3B, 6.7B, 33B	32k	16,384	Yes

Table 1.2: Overview of Code-Specific Large Language Models

Table 1.2 provides a comparative overview of several prominent code-specific LLMs, detailing their architectures, parameter sizes, vocabulary sizes, context window lengths, and open-source availability. This information is primarily based on the comprehensive survey presented in "*A Survey on Large Language Models for Code Generation*" [26], which consolidates key specifications from various code-oriented LLMs developed.

11. Conclusion

In summary, deep learning and Transformer-based models have transformed NLP and AI, making LLMs essential tools for both research and practical applications. Techniques like pre-training, fine-tuning, and prompting empower LLMs to handle a wide range of tasks. The chapter emphasized how innovations and Retrieval-Augmented Generation (RAG) continue to push the limits of what intelligent systems can achieve.

CHAPTER 2
Debugging in Software Development

1. Introduction

Debugging is a critical process in software development, aimed at identifying and resolving faults to ensure correct program behavior. As software grows in complexity, debugging plays a vital role in maintaining quality, improving reliability, and reducing long-term costs. This chapter explores the importance of debugging, common bug types, practical strategies, and tools used by developers. It also introduces modern advancements such as automated program repair, highlighting how these innovations are shaping the future of software reliability.

Debugging is a methodical process of finding and fixing the number of bugs, or defects, in a computer program [27]. It involves isolating the root cause of errors, understanding their impact, and implementing corrective measures. Debugging is a critical phase in software development, quality assurance, and maintenance.

2. Importance of Debugging in Software Development

Bugs and errors occur in computer programming because it is an abstract and conceptual activity. Computers manipulate data in the form of electronic signals. Programming languages abstract this information to allow humans to interact more effectively with computers. Any type of software involves multiple layers of abstraction, with different components communicating to enable an application to function properly. When errors occur, it can be challenging to identify and resolve the issue.

Debugging tools and strategies enable faster problem-solving and enhance developer productivity. Consequently, software quality and the end-user experience are improved.

3. Relationship between Debugging and Software Quality

One of the important steps in software development that directly affects the final product's quality is debugging. It entails systematically locating, identifying and fixing program bugs. The following are some of the main impacts of debugging on software quality:

3.1. Impact on Software Reliability

A key element that contributes to a reliable programming system is the assurance that the software will perform satisfactorily according to both its functional and nonfunctional

specifications within expected deployment environments. Effective debugging plays a vital role in achieving this by reducing the number of defects, thereby increasing the system's dependability.

3.2. Enhancing Maintainability

Maintainability is another key aspect of software quality that debugging directly influences. When developers engage in structured debugging, they improve their understanding of the system's architecture and behavior. This, in turn, leads to better documentation, cleaner code, and easier future modifications [28].

3.3. Reducing Development and Maintenance Costs

Debugging is a costly and time-consuming aspect of software development that frequently consumes 50–75% of development resources [29]. Debugging tools help developers identify and fix bugs more efficiently, which lowers development costs right away and helps keep maintenance costs down over time.

3.4. Ensuring Performance Optimization

Some bugs, such as memory leaks or inefficient algorithms, lead to performance degradation. Debugging helps in identifying performance bottlenecks (specific points in the system where limitations in resources such as CPU, memory [30] and optimizing the system for better efficiency.

3.5. Preventing Recurring Defects

Debugging also contributes to software quality by preventing the recurrence of similar defects. Developers who analyze bug patterns and maintain logs of debugging sessions enhance their ability to anticipate and avoid future errors. Additionally, debugging complements software testing by offering deeper insights into the causes of failures, leading to more effective test case design.

4. Common Programming Bugs

Software bugs can arise from a variety of sources and present distinct challenges in both detection and resolution. Three key types of bugs; branch errors, assignment errors, and code-missing errors [31].

4.1. Branch Errors

Branch errors, particularly those affecting conditional statements, are naturally addressed by

the proposed debugging methodology. This is because the approach is based on control flow and thus excels at detecting bugs that alter the execution path. By focusing on how the execution path deviates due to faulty conditions.

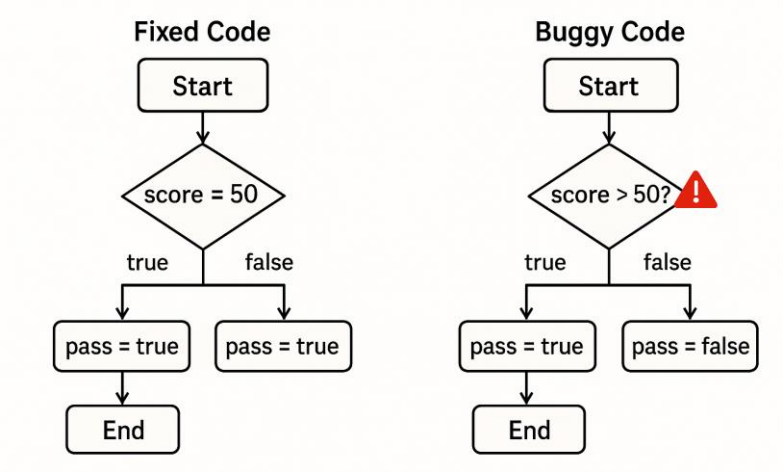

Figure 2.1: An Example of Detecting a Branch Bugs Using Control Flow Comparison

Figure 2.1 shows how a branch error, caused by a faulty condition (score > 50?), changes the program's execution path compared to the correct version, helping highlight the bug through control flow comparison.

4.2. Assignment Errors

Assignment errors occur when a variable is given an incorrect value, leading to faulty output. They are more difficult to identify using control-flow-based techniques because, in contrast to branch faults, they do not alter the program's control flow. These errors include assigning wrong constants, computing incorrect expressions, or updating the wrong variable.

4.3. Code-missing Errors

When segments of code are unintentionally left out of a newer version of a program, this is known as a code-missing error. These omissions result in fewer execution paths in the updated version compared to the original. By comparing execution traces and identifying diverging paths, such errors can be detected, enabling localization without modifying the core debugging methodology.

5. Debugging Strategies

The approach to debugging is influenced by the nature of the bug, the complexity of the code, and the available debugging tools. Various strategies assist in streamlining the process, facilitating the efficient identification and resolution of bugs. The common debugging strategies

include:

5.1. Forward Search Debugging

This approach entails methodically tracking the program's execution from its starting state and following it step-by-step to locate bugs. Because it enables developers to compare the expected and actual results at each level of execution, it works especially effectively when the program's expected behavior is well understood [32].

5.2. Backward Search Debugging

This debugging technique entails locating the bug's onset and tracking the execution flow backward to uncover the underlying root cause. It is particularly useful when an unexpected outcome is observed, but the exact location of the fault within the code is unknown.

By analyzing the most recent operations that modified the relevant data, developers can systematically eliminate possible causes and narrow down the source of the problem [33].

5.3. Hypothesis-Based Debugging

The Hypothesis-Test strategy uses code evidence or runtime behavior to generate and test hypotheses regarding the root cause of a defect. This strategy includes developing hypotheses about the cause of the error and testing them by collecting evidence from the code or runtime behavior [32].

Expert developers frequently base their reasoning more on output data than input data, which makes this approach particularly useful for complicated or challenging-to-reproduce problems.

5.4. Test-Driven Debugging (TDD)

TDD is a software development strategy that integrates debugging into the development lifecycle by requiring test cases to be written before the actual implementation. This approach ensures early defect detection and provides a structured method for identifying and resolving bugs as the code evolves [33].

By translating functional requirements into test cases, TDD helps developers break down complex problems into smaller, manageable sub-problems derived from test case requirements.

6. Debugging Tools

Based on studies, Spinellis [34] observes that debugging is a difficult and costly activity, in some cases accounting spend 20% to 40% of development time. Consequently, tools that assist in debugging are extremely valuable by reducing the time required to identify and fix bugs. These tools providing multiple solutions for various programming environments. These tools

include:

6.1. Traditional Debugging Tools

Examine the state of a program while it is running and manage its flow of execution. Among the most crucial tools are the following:

- **GNU Debugger (GDB):** A source-level and machine-level debugger. It, however, supports more languages, like C, C++, D, Fortran, Go, Objective-C, Pascal, Rust, and some others [35].

- **LLVM Debugger (LLDB):** Is part of the LLVM tools collection [36], supporting C, C++, and Objective-C, with strong integration with the Clang compiler.
- **Python Debugger (PDB):** A command-line debugging tool designed for Python, helping developers trace errors during execution that comes built into Python [37].
- **Java Debugger (JDB):** It enables developers to manage program execution, set breakpoints, examine variables, and examine stack traces through the use of a command-line tool for debugging Java applications [38].

6.2. Advanced Debugging Tools

For effective bug detection and resolution, modern software development depends on advanced debugging tool. These tools are categorized into:

- **Dynamic analysis tools:** These tools are essential for identifying software issues that only emerge during runtime, such as memory leaks, runtime errors, and concurrency problems [29], they analyze program's behavior while it is running.

Tools like Valgrind and AddressSanitizer are widely recognized for their ability to detect out-of-bounds access, use-after-free, and invalid memory deallocations, particularly in C/C++ programs [34].

- **Static analysis tools:** These tools enable developers to identify potential issues early in the development process by analyzing source code without actually running it. For example, the FindBugs inspects Java bytecode to detect possible bugs, such as null pointer references and infinite loops, before the program runs [34].

6.3. Web Debugging Tools

The complicated challenges of modern web development, which is by its very nature asynchronous, event-driven, and distributed across several platforms and browsers, need the use of web debugging tools that requires specialized strategies due to the dynamic nature of the

DOM and the widespread use of JavaScript. Browser-integrated tools like Chrome DevTools and Firefox Developer Tools offer powerful features such as *"Inspect", "Watch", "Call Stack" and "Scope"* that enable developers to monitor element behavior, trace execution paths, and analyze variable scopes in real time. These properties are used by techniques like DOM-debugging and DevTools-debugging to identify problems like improper DOM manipulation or attribute changes [32].

6.4. Integrated Development Environments (IDEs) Debugger

By offering integrated debugging tools that assist developers in identifying and fixing problems while coding, IDEs increase the efficiency of development. Real-time syntax checking and auto-completion to avoid structural errors are made possible by these tools' ability to detect language-specific faults without the need for code compilation.

Additionally, IDEs support instant execution and debugging, enabling developers to test and fix code issues during development. These environments, such as Visual Studio, Eclipse, IntelliJ IDEA, and PyCharm enable developers to set breakpoints, step through code, and inspect variables and data structures in real time [29].

The graphical user interfaces in these IDEs often display source code alongside execution markers and breakpoint indicators, making it easier to track program flow and internal state.

7. Challenges in Debugging

Despite advancements in debugging methodologies and tools, modern software development still faces numerous challenges, such as:

7.1. Large-Scale Systems and Distributed Systems

Debugging across several services, databases, and related components becomes increasingly difficult as software architectures grow more complex. Large-scale distributed systems pose additional challenges such as heterogeneity, concurrency, distributed state, and partial failures.

7.2. Limited Access to Code and or Debugging Tools

Developers have significant difficulties when they don't have access to source code or debugging tools, particularly in production settings because of the lack of complete visibility or control, as occurs when working with built binaries, third-party APIs, or client-side components.

7.3. Error Localization Complexity

One of the most difficult and time-consuming aspects of debugging is error localization.

Developers frequently invest hours in determining the hidden causes of apparent symptoms. It is a crucial but challenging stage in guaranteeing software stability since it requires patience, deep code understanding, and careful analysis.

7.4. Lack of Clear Debugging Strategies

Developers frequently rely on guesswork and trial-and-error when debugging instead of using explicit, defined methodologies [34]. This absence of clear instructions slows down problem-solving and makes debugging more difficult.

7.5. Variations in Developer Expertise

Developers have varying levels of experience, which affects their ability to troubleshoot efficiently. While novice developers may struggle with fundamental debugging practices, experienced professionals often face more intricate system-level challenges.

7.6. Influence of Environmental and Contextual Factors

The complexity of the codebase, the type of bug, team dynamics, the debugging tools available, and project limitations are some of the variables that influence how effective debugging is.

8. Modern Advancements in Debugging

Debugging has consistently been an essential component of software development, necessitating robust problem-solving abilities and appropriate tools. As systems grow more complex, this has led to the emergence of modern advancements, including:

8.1. Large Language Models (LLMs) for Debugging

In automated debugging, LLMs, especially those designed for code, have become effective tools. By verifying intermediate variable states and control flows, LLMs enable step-by-step examination of runtime execution and refinement of generated code.

This approach, which mimics human debugging techniques, allows LLMs to identify faults more precisely [39].

8.2. Automated Fault Localization (AFL)

Fault localization aims to automatically identify buggy lines of code, serving as a crucial first step in many manual and automated debugging tasks Retrospective fault localization improves bug detection by reusing the results of failed patch attempts, without adding much extra computing cost [40].

8.3. Self-healing Software Systems

Self-healing software systems automatically detect, diagnose, and fix failures at runtime to maintain functionality without human help. They use AI models and observability tools to enhance software resilience [41].

9. Automated Program Repair (APR)

Automated Program Repair is an emerging field in software engineering that aims to automatically fix bugs and security vulnerabilities (software defects) in software systems without human intervention [42].

The primary goal of APR is to automatically generate source-level patches, similar to those written by programmers when addressing issues in their code. APR techniques typically take as input a buggy program, often identified through a failing test case, and produce a patch that corrects the fault while preserving the program's intended functionalities. This automation aims to reduce the manual debugging effort and contribute to maintaining high software quality.

10. Typical Workflow of APR

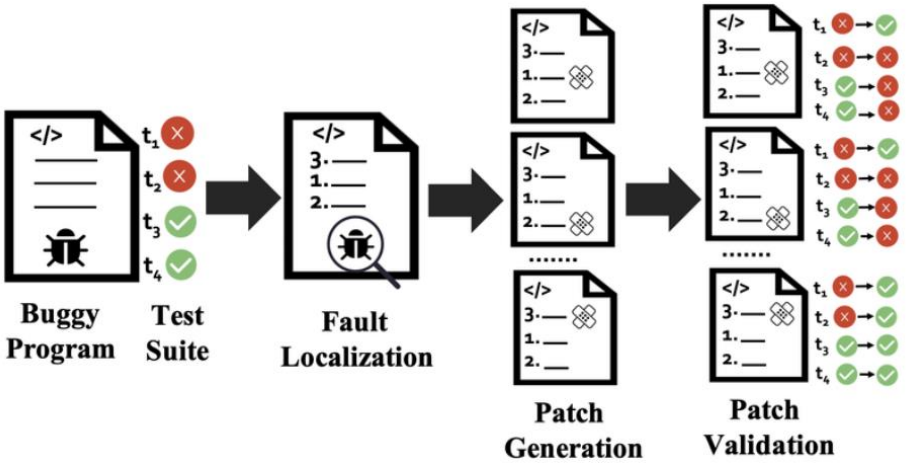

Figure 2.2: Workflow of Automated Program Repair

APR typically follow a structured process to detect and fix bugs in software. The most commonly adopted workflow is known as the generate-and-validate paradigm [43], which composed of three main stages:

10.1. Fault Localization

Identifying the faulty parts of the source code is the first phase. The goal of this stage is to find questionable code elements that are probably bugged. It is built on existing fault

localization methods, which typically utilize information from test executions especially failed test cases to rank code statements based on their likelihood of being faulty.

10.2. Patch Generation

Once the buggy code regions have been identified, the APR system advances to the patch generation stage. In this phase, the system creates candidate patches by applying predefined transformation rules to the suspicious code segments.

These rules specify how particular lines or blocks of code can be modified, deleted, or replaced in order to address the underlying bug.

10.3. Patch Validation

Patch validation is the last step in the APR procedure, during which the system assesses the produced patches using the test suite already in place for the program. These tests assist in determining whether a patch fixes the bug without creating additional problems.

A patch is regarded as credible if it passes every test. The correct patch is one that not only satisfies the test suite but also maintains the intended behavior of the original program across a broader range of cases, much like a fix written manually by an experienced developer.

11. Overview of APR Techniques

There are several techniques for APR, each employing different strategies for generating and validating patches [44]. This section provides a brief overview of the main APR techniques, including:

11.1. Search-Based Repair

The core idea of search-based techniques is to search for the correct patch in a predefined patch space. The techniques first look in the search space to identify the most likely locations of buggy code in the program, usually with the help of heuristics to generate candidate fixes, and then search for suitable fix patches by means of mutation-selection, test execution, traversal strategies.

These techniques rely heavily on how effectively the search space is explored. By using intelligent heuristics and adaptive exploration strategies, they increase the chances of locating valid patches that pass all tests. This approach is particularly useful when dealing with large and complex codebases, where manually identifying the root cause of bugs becomes impractical.

11.2. Constraint-Based Repair

Constraint-based program repair techniques are among the most effective approaches, relying on program behavior analysis and the derivation of solutions based on formal specifications. These techniques transform the program repair problem into a constraint solver problem and use formal specifications to quickly prune infeasible parts to find expression-level variations that satisfy constraints collected from tests by program analysis, thus facilitating patch generation.

This approach is fundamentally grounded in the idea of constructing a repair constraint that the patched code should satisfy. The patch code to be generated is treated as a function to be synthesized. Solutions to the repair constraint can be obtained by constraint solving or other search techniques. In these approaches, formulation of the repair constraint is the key, not the mechanism for solving it [44].

11.3. Template-Based Repair

Template-based repair techniques emerged to address some of the shortcomings of constraint-based approaches, particularly the difficulty in formulating suitable specifications for each bug type and the limitations imposed by heavy reliance on constraints. In this context, this approach relies on predefined program fix templates (also known as fix patterns or transformation schemas) to generate repair patches. These templates can be manually extracted or automatically mined.

11.4. Learning-Based Repair

Learning-based repair techniques offering a shift from predefined rules to data-driven solutions. This technique is also known as data-driven repair solutions, where APR tools can automate the learning of empirical knowledge from a large number of defect repair samples that can be used to guide defect repair [44].

These approaches are classified into three main categories: those that learn models of correct code to rank patches, those that infer transformation templates from commit histories, and end-to-end models that predict the fixed code directly from the buggy version without requiring test suites or constraint solvers.

12. Evaluation of APR

The evaluation of APR is essential for measuring the effectiveness and practicality of suggested methods. Whether an APR technique can consistently correct defects in real-world software or if it simply works in specific, controlled scenarios is hard to determine without

thorough evaluation.

Most APR workflows follow a generate-and-validate pipeline, where after detecting the bug and generating candidate patches.

This validation step is crucial to ensure that the proposed fix does not just make the test cases pass, but actually resolves the intended bug without introducing new issues [45].

12.1. Benchmarking APR

The advancement of research in APR depends on benchmarking because it offers standardized resources for the systematic evaluation of tools and techniques in the field of software engineering. By comparing different APR approaches objectively using the same datasets, researchers can avoid inconsistencies that may arise from different experimental setups.

Benchmarks capture real-world code modifications made to fix bugs, aligning program behavior with intended specifications.

Community-curated datasets such as Defects4J, CodeFlaws, and ManyBugs support reproducible, equitable evaluation, exposing common challenges and guiding advancements in APR and fault localization.

12.2. Benchmarking Datasets for APR

Benchmark	Language	Bugs	Project Type	Real-World/Artificial
Defects4	Java	835	Open-source projects	Real-World
GitBug-Java	Java	199	GitHub Repositories/Issues	Real-World
HumanEval-Java	Java	164	Multiple OSS projects	Real-World
ManyBugs	C/C++	185	OSS Projects	Real-World
IntroClass	C/C++	+998	Student Assignments	Semi-artificial
CodeFlaws	C/C++	3,902	Programming Contests	Artificial
BugsInPy	Python	160	OSS Projects	Real-World
PyBugs	Python	+ 200	OSS + Scripts	Real-World

Table 2.1: Overview of Benchmark Datasets for APR

Table 2.1 provides an overview of widely used benchmark datasets in the evaluation of APR systems across different programming languages. The information is derived from the recent paper *" Software Development Life Cycle Perspective: A Survey of Benchmarks for Code Large Language Models and Agents"* [46], to ensure data reliability and up-to-date coverage of benchmarks.

The size, provenance, and complexity of these datasets vary, offering a varied testing environment.

Defects4J, GitBug-Java, and HumanEval-Java are popular benchmarks for Java that provide a realistic environment for tool evaluation by using real-world bugs gathered from open-source projects.

ManyBugs and IntroClass are commonly used in the C/C++ sector; IntroClass is semi-artificial and instructional. A sizable artificial dataset from competitive programming is provided by CodeFlaws.

Real bugs for Python are captured by PyBugs and BugsInPy from various open-source repositories. Standardizing experimental conditions and facilitating equitable comparisons between APR systems are made possible by these standards.

13. Conclusion

This chapter outlined the importance of debugging in enhancing software quality, optimizing performance, and reducing maintenance costs. It examined common bug types such as branch, assignment, and code-missing errors, along with effective debugging strategies like forward search, backward search, and test-driven debugging. Tools ranging from traditional debuggers to modern IDEs and web-based platforms were reviewed. Key challenges in debugging large and complex systems were also addressed.

Finally, recent advancements such as LLMs and APR techniques highlight the shift toward more intelligent and efficient debugging processes in modern software development.

CHAPTER 3

Large Language Models on Automated Program Repair

1. Introduction

The domain of APR is no longer the same with the emergence of LLMs. The former are trained with massive volumes of code as well as natural language and have showcased remarkable abilities in comprehending, generating, and repairing source code with minimal supervision. The chapter describes how the landscape of APR changed with the impact of LLMs by comparing the conventional techniques with the LLM-based techniques. It also reviews different architecture paradigms prompt-based, fine-tuned, and hybrid along with state-of-the-art systems utilizing LLMs for efficient, scalable, and smart program repair.

2. Comparison between Traditional and LLM-Based APR Techniques

Feature	Traditional APR	LLM-Based APR
Approach	Search / Heuristic	Data-driven / Generative
Speed	Faster at inference	Slower without optimization
Accuracy	Medium	High
Architecture	Classical DL models	Transformer-based LLMs
Data Requirement	Large repair datasets	Few-shot / Zero-shot possible
Patch Evaluation	Manual or test-based	Automatic / Self-repair possible
Integration	Mostly standalone	Flexible, easy to integrate

Table 3.1: Key Differences between Traditional APR and LLM-Based APR Approaches

Table 3.1 lists the main differences between LLM-Based APR and Traditional APR techniques across several features. In terms of approach, Traditional APR uses search-based or heuristic techniques to find and apply code fixes, where LLM-Based APR uses a data-driven, generative approach using pretrained language models. When it comes to speed, LLM-based systems can be slower, particularly if they are not optimized for efficiency, whereas traditional

methods, which benefit from optimized classical tools, tend to be faster at inference.

Depending on the scale and quality of the model, LLMs frequently attain higher levels of accuracy than Traditional APR, which usually produces results that are moderate.

LLM-based APR relies transformer-based architectures, whereas traditional approaches are based on custom search logic or classical deep learning. While Traditional APR depends on sizable, manually selected repair datasets, LLMs can perform well even in few-shot or zero-shot scenarios because of their thorough pretraining.

In traditional APR, patch evaluation is typically done by hand or based on test results, in contrast, LLM-based approaches can automate this process and can even self-evaluate in some setups.

Finally, in integration, LLM-based solutions are flexible and integrates easily into modern development pipelines, while traditional APR tools are frequently stand-alone tools.

3. Model Architectures for LLM based APR

3.1. Prompt-based Approaches

Prompt-based approaches rely on the pre-trained capabilities of large language models to perform program repair without the need for extensive fine-tuning on specialized bug-fixing datasets [47]. Examples include OpenAI Codex and CodeLlama.

Within this approach, two main strategies are commonly used: Zero-shot learning, where the model is prompted to perform bug fixing without being shown any prior examples, and Few-shot learning [43], where a small number of bug-fix examples are included in the prompt to guide the model on how to execute the repair and format the expected output.

3.2. Fine-Tuned LLMs

Fine-tuned models are pre-trained LLMs that are further trained on specific bug-fixing datasets to enhance their performance in APR, particularly for handle complex and multi-location bugs. In this approach, a broadly trained general-purpose model is adapted to become more specialized and effective for the repair task.

These models typically produce more accurate and context-aware patches but need more training resources. Examples include CodeBERT fine-tuned on ManySStuBs4J [48] and RepairLLaMA trained on repair datasets [49].

3.3. Hybrid Models

Hybrid approaches in LLM-based APR combine LLMs with other agents or tools to enhance patch quality and correctness, like verifiers or completion engines, to improve patch

correctness. For example, RePilot framework combines an LLM (like GPT-3.5) with a completion engine to validate and refine patches [50].

Common strategies include conversational repair, where models like ChatGPT iteratively refine patches based on test feedback; engine-based repair, where LLMs fill structured templates or work alongside code completion engines; and self-repair, where the model evaluates and corrects its own output based on execution or analysis [43].

4. Related Work on APR using LLMs

4.1. Direct Code Repair with Task-Specific LLMs

Model	Model Family	Model Architecture	Dataset	Language used in Datasets
TFix	T5	Encoder-Decoder	LintBugs; Static Analysis Corpora	TypeScript
FitRepair	CodeT5	Encoder-Decoder	Defects4J, Bugs2Fix	Java
AlphaRepair	CodeBERT	Encoder	CodeXGLUE, Custom Patch Datasets	Java, Python

Table 3.2: Overview of Task-Specific LLMs for APR

Task-specific LLMs used for program repair, while these models differ in architecture, training datasets, and accuracy, they all aim to generate correct patches for buggy code.

FitRepair [52] reports the highest accuracy among the evaluated models, likely due to fine-tuning on diverse Java bug-fix datasets such as Defects4J and Bugs2Fix, both of which include real-world patches (Table 3.2).

TFix [51], targets TypeScript code and is trained on data derived from static analysis tools and developer edits. It repairs bugs by conditioning on both the buggy code and the associated bug message, yet its accuracy remains moderate, possibly reflecting the complexity of TypeScript linting errors and the narrow domain of its training data.

Although AlphaRepair [53] uses CodeBERT, an encoder-only model that supports multiple languages, its performance does not reach the same level. This result suggests that encoder-decoder models might work better for generating code patches.

4.2. Infilling and Span-Based Repair

Approach	Model	Model Architecture	Dataset Used	Language Used in Datasets
RepairLLaMA	CodeLlama	Decoder-Only	RepairLLaMA Dataset	Java
CoCoNut	Transformer (custom)	Encoder-Decoder	Bugs2Fix, ManySStuB	Java
Bug-INFER	BART	Encoder-Decoder	Bugs2Fix, CodeXGLUE	Java

Table 3.3: Overview of Infilling and Span-Based Repair Approaches in APR

Infilling and span-based repair methods in APR adopt a more localized strategy than direct patch generation. Instead of identifying and replacing specific spans or lines within buggy code. This often leads to more precise and minimal edits, enhancing both interpretability and generalization.

4.2.1. Overview of RepairLLAMA

RepairLLaMA proposes to leverage parameter-efficient fine-tuning methods to fine-tune LLMs for particular representations of code to further augment APR for improved bug-fixing abilities. The model applies Low-Rank Adaptation (LoRA) to build a smaller repair adapter, fine-tuning just a subset of the parameters around 4 million instead of the full 7 billion parameters in the base LLM.

The fine-tuning dataset is crafted to feature appropriate pairs of buggy and fixed code examples, balancing diversity, quality, and size to achieve improved generalization of the model. The overall methodology takes into consideration using an initial pre-trained model, specifying input and output code representations, and using a fine-tuning dataset created specifically for APR tasks.

The training strategy is based on parameter-efficient fine-tuning, which ensures decreased GPU memory consumption and prevents overfitting while also testing the model using measures designed to correct data leakage issues. Experimental results demonstrate RepairLLaMA's effectiveness across multiple benchmarks, achieving repairs on 144 bugs in Defects4J v2, 109 in HumanEval-Java, and 20 in GitBug-Java, exceeding the performance of existing baselines [49].

The strengths of RepairLLaMA include enhanced efficiency of fine-tuning, improved repair-specific code representations, and better performance in repairing outside of the fine-tuning data distribution.

4.2.2. Overview of CoCoNuT

CoCoNuT is an advanced program repair approach that combines context-aware neural machine translation (NMT) with fully convolutional (FConv) models to promote improved automated bug fixing. Using dual encoders to process buggy code lines and context independently.

The model was evaluated across six programming languages across four benchmarks datasets, successfully repairing 509 bugs [54]. Its architecture utilizes convolutional neural networks to represent code features at varied levels of granularity.

CoCoNuT is also noted for its portability, representing the first method demonstrated to repair bugs in both JavaScript and Python with minimal manual intervention.

4.2.3. Overview of Bug-INFER

Bug-INFER is a new bug detection system that employs a combination of static and dynamic analysis to recognize bugs from trace executions. It is scalable to work efficiently even in large-scale, real applications.

Bug-INFER enhances bug detection precision by incorporating learning models trained from prior bug data to identify patterns and infer potential bugs with ease. The system puts a special focus on minimizing false positives, thereby making results more reliable. Comparative experiments show that Bug-INFER outperforms existing bug detecting tools under different scenarios, highlighting its strength and practical value [55].

4.3. Retrieval-Augmented Generation (RAG)

Approach	Model + Retriever	Model Architecture	Dataset Used
RAP-Gen	LLM + RAG	Decoder-Only + Retriever	External codebase
InferFix	LLM + static analyzer	Decoder-Only + Static Analyzer	Fix corpus from linters
RagVerus	RAG-based verifier	RAG-based verifier	Repository-level code

Table 3.4: Overview of Retrieval-Augmented Generation Approaches in APR

Retrieval-Augmented Generation has recently emerged as a promising direction in APR, enabling models to retrieve relevant external information, such as similar bug-fix patterns, linter suggestions, or repository-level code snippets [56].

This approach enhances the quality and accuracy of suggested fixes by grounding them in real, contextually similar examples, thereby mitigating hallucinations and increasing relevance. Table 3.4 highlights three notable RAG-based methods; RAP-Gen, InferFix, and RagVerus.

4.3.1. Overview of RAP-Gen

RAP-Gen is an advanced model for APR that aims to reduce developers' manual debugging effort and enhance software reliability. The model is based on a hybrid patch retriever and CodeT5 patch generator, using a retrieval-augmented generation framework.

The methodology consists of three main stages: first, training a hybrid retriever to classify similar bug-fix patterns; second, training the CodeT5 model to generate candidate patches; and third, ranking these patches to facilitate developer validation. During inference, the retriever identifies relevant fix patterns, which are concatenated with the buggy input and passed to the CodeT5 model to generate ranked repair candidates.

RAP-Gen was evaluated on three benchmark datasets: TFix for JavaScript, and Code Refinement and Defects4J for Java [56]. Results show that RAP-Gen outperforms previous models across all benchmarks, demonstrating its effectiveness in automated bug fixing.

Its main contributions include the introduction of a novel retrieval-augmented framework for APR, the integration of a code-aware pretrained language model, and a flexible design capable of adapting to diverse bug types.

4.3.2. Overview of InferFix

InferFix is an end-to-end automated program fix framework for repairing critical performance and security bugs for Java and C#. It integrates advanced static analysis with a transformer-based language model to enable accurate and efficient bug fixing.

The framework consists of three principal components: a static analysis component for detection and bug classification of bugs, a retrieval component for fetching similar bugs from the past and their respective fixes, and a generator component that employs machine learning for synthesising plausible repairs.

InferFix is shown to perform well, with 76.8% top-1 fix accuracy for Java and 65.6% for C# over the InferredBugs corpus. InferFix is successfully integrated and used in Microsoft's internal continuous integration pipelines for the purpose of increasing development productivity and the reliability of the code. Experimental evaluations show that InferFix achieves 76.8% top-

1 fix accuracy on Java and 65.6% on C# using the InferredBugs corpus [57].

4.3.3. Overview of RagVerus

RagVerus is a retrieval-augmented framework designed to aid LLM-based program verification, particularly for large-scale Verus projects. It addresses the complexity of verification and proof synthesis by integrating RAG with context-aware prompting.

The framework employs persistent vector storage and FAISS-based indexing to retrieve relevant code and specification context from multi-module repositories efficiently.

Experimental results indicate that RagVerus achieves a 27% relative improvement over the RepoVBench benchmark [58].

Additionally, RagVerus is designed to be extensible, enabling its application to tasks such as specification inference and code generation, thereby supporting broader use cases in program verification workflows.

5. Conclusion

This chapter described how LLMs transformed repairing computer programs. In the past, manual search and slow code fixes were used. Now, LLM methods can read code and repair bugs more accurately. We considered three types of LLM approaches: prompt-based, fine-tuned models, and hybrids. A number of new systems such as RepairLLaMA, CoCoNuT, and RAP-Gen produce excellent results. LLMs are able to repair various types of bugs and support numerous programming languages.

CHAPTER 4
Implementation

1. Introduction

In this chapter, we present the practical realization of our proposed a new approach for APR, which integrates a LoRA-based fine-tuned CodeLLaMA model with the GraphRAG retrieval framework. We begin by fine-tuning the CodeLLaMA-7b model on the RepairLLaMA dataset using Low-Rank Adaptation (LoRA) to achieve efficient training on domain-specific bug-fix data. Subsequently, we construct a context-aware retrieval pipeline by embedding buggy and fixed code snippets from Defects4J into a FAISS vector store using CodeBERT. These embeddings are further enhanced through a graph-based retriever that captures structural relations among code examples. A custom strategy is then applied to guide the retrieval process, ensuring that the most relevant code snippets are selected to assist in generating accurate repairs.

Subsequently, we built a simple web interface that connects our model with the GraphRAG system to provide real-time code repair patches.

2. Datasets Overview

We chose RepairLLaMA-datasets collection hosted on Hugging Face to support effective learning, strong generalization, and real-world applicability in our work.

We selected this existing dataset specifically to fine-tune our model, as it is tailored for program repair tasks and built on Megadiff, a large-scale dataset of Java code edits, where each sample captures a real bug and its corresponding fix, mined from open-source GitHub repositories [49].

Each data sample consists of paired buggy and fixed functions, structured into different IRxOR formats, each designed for specific training scenarios and model architectures. The dataset provides four key formats, combining IR (Input Representation) and OR (Output Representation) [49]:

IRxOR Format	Format Name	Description
IR1	Full Context Format	Provides full buggy/fixed functions without markup.
IR2	Marked Lines Format	Buggy lines are marked with comment tags; fix is isolated.

IR3	Fill-in Format	Buggy parts replaced with <FILL_ME> placeholders.
IR4	Commented Buggy Format	Buggy code commented out, <FILL_ME> tags inserted for fix generation.

Table 4.1: IRxOR Dataset Representation Formats

To investigate the impact of different code representations on repair performance, the table 4.2 presents the results of fine-tuning a LLM for program repair using the Defects4J v2 dataset (488 bugs) [49]. The performance is assessed using four metrics:

- **Plausible Match**: Output compiles and passes all tests.
- **Exact Match**: Output matches ground truth exactly.
- **AST Match**: Abstract Syntax Tree structure is the same.
- **Semantic Match**: Code is semantically equivalent.

Code Representations	Defects4J v2 (488 bugs)			
	Plausible	Exact Match	AST Match	Semantic Match
IR3 x OR2 (no fine-tuning)	131	52	70	83
IR4 x OR2 (no fine-tuning)	107	50	60	69
IR1 x OR1	79	29	31	45
IR1 x OR3	41	15	17	24
IR1 x OR4	12	2	2	3
IR2 x OR2	**198**	121	122	139
IR3 x OR2	153	83	86	102

Table 4.2: Comparison of Code Representations for Program Repair on Defects4J v2 [49]

The table 4.2 clearly shows that the IR2 × OR2 representation significantly outperforms others across all metrics. It achieves:

198 plausible fixes, meaning the generated code compiles and passes all test cases.

121 exact matches with the ground truth.

122 AST matches, indicating structural similarity in code.

139 semantic matches, showing functional equivalence.

In contrast, other representations especially those without fine-tuning (e.g., IR3 × OR2 or IR4 × OR2) perform substantially lower. For instance, IR1 × OR4 achieved only 12 plausible

matches and very low scores across the rest of the metrics.

These results demonstrate the strong impact of code representation on repair quality. The IR2 × OR2 configuration provides the most reliable and accurate bug fixes, making it the most suitable choice for fine-tuning the LLM. This configuration allows the model to better understand both the intent and structure of the code, leading to more meaningful and correct repairs.

2.1. Why RepairLLaMA-Dataset?

For selecting a dataset for the APR task, we chose the RepairLLaMA dataset based on several key factors:

- This dataset contains paired examples of buggy and fixed code, allowing the model to learn how to transform incorrect code into its corrected version.
- The nature of the errors in the dataset closely resembles the types of real-world bugs our model is expected to fix.
- We selected this dataset because it offers a good balance between size and quality.
- It is built from publicly available open-source repositories with proper curation, allowing us to address legal and ethical concerns in data usage.

3. Choice of the Initial Large Language Model

We trained several modern language models that that have shown strong performance in order to choose the most suitable one for our APR pipeline.

The models we worked with included DeepSeek, Qwen, and CodeLLaMA-7B. The table below summarizes their key characteristics during our fine-tuning process:

Model Name	Training Time	GPU Utilization
DeepSeek	~17 hours	High
Qwen	~14 hours	High
CodeLLaMA-7B	~11 hours	Moderate

Table 4.3: Training Performance of LLMs on 10% of the RepairLLaMA-Dataset

The table 4.3 summarizes the resource usage and training duration for each language model during initial experiments. Each model was fine-tuned using 10% of the RepairLLaMA dataset.

Although DeepSeek and Qwen were promising in theory, in practice they required significantly longer training times and posed challenges in our Colab environment due to high GPU demands. We initially chose these models for their relatively modest parameter sizes, but

fine-tuning even a small portion of our dataset took over 12 hours.

On the other hand, CodeLLaMA-7B provided the best balance between training efficiency and memory usage, especially when paired with LoRA and 8-bit quantization. As a result, it became our preferred choice for the repair task.

3.1. Why CodeLLaMA-7B?

For selecting a foundational large language model for fine-tuning in the APR task, we chose CodeLLaMA-7B based on the following key criteria:

- Open-source and freely available, with accessible weights that support reproducibility and avoid the high costs associated with closed-source APIs.
- Pre-trained on a large-scale code corpus, enabling the model to better understand and reason about real-world programming patterns and errors.
- Supports the infilling objective, allowing the model to generate code given both left and right contexts, which is an essential feature for bug localization and repair.
- Compatible with parameter-efficient tuning methods such as LoRA and quantization, which makes training feasible on limited hardware like Colab GPUs.

4. Our Proposed Approach

4.1. LoRA-Based Fine-Tuned CodeLLaMA Model on the RepairLLaMA Dataset

We apply parameter-efficient fine-tuning techniques, specifically Low-Rank Adaptation (LoRA) to adapt a large-scale LLaMA-based language model (CodeLLaMA-7b) for the task of APR. This fine-tuning process is performed using domain-specific datasets.

A CodeLLaMA model requires several important stages to be fine-tuned. First, we imported the necessary libraries and loaded the pre-trained CodeLLaMA-7b model, which had been trained on large-scale code data.

```python
import os
import torch
import json
import functools
import logging
import numpy as np
import networkx as nx
from dataclasses import dataclass, field
from typing import Dict, Optional, Sequence
from sentence_transformers import SentenceTransformer

from transformers import (
    AutoTokenizer, AutoModelForCausalLM, Trainer, TrainingArguments,
    DataCollatorForSeq2Seq, BitsAndBytesConfig
)
from peft import (
    LoraConfig, get_peft_model, prepare_model_for_kbit_training
)
from datasets import load_dataset
from sentence_transformers import SentenceTransformer
```

Figure 4.1: Importing Necessary Libraries

Then, we integrated the LoRA (Low-Rank Adaptation) technique to enhance fine-tuning efficiency, enabling the update of only a small subset of trainable parameters while preserving the core model weights.

```python
# Lora settings
LORA_R = 8
LORA_ALPHA = 16
LORA_DROPOUT = 0.05
LORA_TARGET_MODULES = ["q_proj", "v_proj"]
MODEL_NAME = "codellama/CodeLlama-7b-hf"
CACHE_PATH = "/content/drive/MyDrive/LLM-APR/"
OUTPUT_DIR = CACHE_PATH
MODEL_MAX_LENGTH = 1024
IS_LORA = True
NUM_PROC = 4
```

Figure 4.2: Model and Training Configuration

We load the base model and tokenizer with the LoRA configuration for parameter-efficient fine-tuning. The model is loaded using 8-bit quantization to reduce memory usage.

We prepare the model for low-bit training, then we configure and apply LoRA adapters, targeting specific attention modules. Finally, we load the tokenizer set the maximum sequence length, and enable left-side padding.

```python
#Load the Pretrained Base Model with Quantization
model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    cache_dir=CACHE_PATH,
    torch_dtype=torch.float16,
    trust_remote_code=True,
    quantization_config=BitsAndBytesConfig(load_in_8bit=True, llm_int8_threshold=6.0),)

model = prepare_model_for_kbit_training(model)

# Define LoRA Configuration for Parameter-Efficient Fine-Tuning
lora_config = LoraConfig(
    r=LORA_R,
    lora_alpha=LORA_ALPHA,
    target_modules=LORA_TARGET_MODULES,
    lora_dropout=LORA_DROPOUT,
    bias="none",
    task_type="CAUSAL_LM",)

# Apply LoRA to the Base Model
model = get_peft_model(model, lora_config)

# Load and Configure Tokenizer
tokenizer = AutoTokenizer.from_pretrained(
    MODEL_NAME,
    cache_dir=CACHE_PATH,
    model_max_length=MODEL_MAX_LENGTH,
    padding_side="left",
    trust_remote_code=True,
    use_fast=True,)
```

Figure 4.3: Model and Tokenizer Initialization

After that, we load the RepairLLaMA dataset from HuggingFace's hub:

```python
dataset = load_dataset("ASSERT-KTH/repairllama-datasets", "ir2xor2")

train_dataset = dataset["train"]
test_dataset = dataset["test"]
```

Figure 4.4: RepairLLaMA Dataset Loading

We apply tokenization to all training and testing samples in parallel, then remove any samples with empty tokenized inputs to ensure clean data for training and testing.

```python
train_dataset = train_dataset.map(generate_and_tokenize_prompt, num_proc=num_proc)
test_dataset = test_dataset.map(generate_and_tokenize_prompt, num_proc=num_proc)
train_dataset = train_dataset.filter(lambda sample: len(sample["input_ids"]) > 0)
test_dataset = test_dataset.filter(lambda sample: len(sample["input_ids"]) > 0)
```

Figure 4.5: Tokenization and Filtering of Training and Testing Datasets

Then, we fine-tune the CodeLLaMA model using parameter-efficient LoRA adapters. We configure the training loop with optimized settings, including mixed-precision (fp16) training and run it for 2 epochs. These settings ensure efficient use of hardware resources while maintaining model quality. After training, we save the fine-tuned model for later integration into our GraphRAG-based APR pipeline.

```
trainer = Trainer(
    model=model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    args=training_args,
    data_collator=data_collator,
    compute_metrics=compute_metrics_fn,
)
trainer.train()
trainer.save_state()
trainer.save_model()
```

Figure 4.6: Fine-Tuning and Saving the LoRA-Adapted CodeLLaMA Model

4.2. Integrating GraphRAG with a Fine-Tuned CodeLLaMA Model

In our project, we aim to merge the LoRA-based fine-tuned CodeLLaMA model with the GraphRAG retrieval framework to build a robust, context-aware APR pipeline that leverages both learned repair patterns and structural reasoning from historical bug-fix graphs.

We load buggy and fixed code pairs from the Defects4J dataset and convert them into LangChain Document objects. Each document contains the code as page_content and metadata like bug ID and file names.

```
buggy_doc = Document(
    page_content=buggy_code,
    metadata={
        "type": "buggy",
        "bug_id": bug_dir,
        "file_buggy": buggy_file,
        "file_fixed": fixed_file,
        "example": False
    }
)

fixed_doc = Document(
    page_content=fixed_code,
    metadata={
        "type": "fixed",
        "bug_id": bug_dir,
        "file_buggy": buggy_file,
        "file_fixed": fixed_file,
        "example": True,
        "references": [f"{bug_dir}_{buggy_file}"]
    }
)

all_docs.extend([buggy_doc, fixed_doc])
```

Figure 4.7: Converte Buggy/Fixed Code Pairs from the Defects4J Dataset into LangChain Documents

To make the system retrieve relevant code snippets quickly, we use FAISS to store and search vector representations (embeddings) of code documents. We generate these embeddings using CodeBERT.

```python
embedding_model = HuggingFaceEmbeddings(model_name="microsoft/codebert-base")

faiss_store = FAISS.from_documents(
     documents,
     embedding_model
)
```

Figure 4.8: Building FAISS Index with CodeBERT Embeddings

We build a graph-based retriever by combining FAISS similarity search with custom graph edges. We first wrap the FAISS store using GraphAdapterFAISS, then initialize the GraphRetriever with predefined edges that represent relationships between code snippets. Using the invoke function, we start with the top 3 similar code snippets, traverse up to 2 levels deep, and select the 6 most relevant results.

```python
adapter = GraphAdapterFAISS(vectorstore=faiss_store)

default_retriever = GraphRetriever(store=adapter, edges=edges)

results = default_retriever.invoke(
    input=buggy_code_snippet,
    select_k=6,
    start_k=3,
    max_depth=2
)
```

Figure 4.9: Graph-Based Code Context Retrieval Using FAISS and Custom Edges

We define a custom strategy, BugFixCodeStrategy, to guide how our GraphRAG system selects code examples. It tracks visited nodes, separates buggy from fixed code using metadata, and prioritizes fixed examples first. This helps the model focus on more relevant context for generating accurate repairs.

```python
@dataclasses.dataclass
class BugFixCodeStrategy(Strategy):
    _nodes: dict[str, Node] = dataclasses.field(default_factory=dict)

    def iteration(self, *, nodes: Iterable[Node], tracker: NodeTracker) -> None:
        self._nodes.update({n.id: n for n in nodes})
        new_count = tracker.traverse(nodes=nodes)

        if new_count == 0:
            buggy_nodes = []
            fixed_nodes = []

            for node in self._nodes.values():
                if node.metadata.get("type") == "fixed":
                    fixed_nodes.append(node)
                elif node.metadata.get("type") == "buggy":
                    buggy_nodes.append(node)

            tracker.select(fixed_nodes)
            tracker.select(buggy_nodes)
```

Figure 4.10: Custom Traversal Strategy for Selecting Buggy and Fixed Code Nodes

5. Evaluation Results on the Test Set (500 Samples):

To assess the effectiveness of our model, we evaluated it on 500 test samples using two standard metrics:

- **Exact Match Accuracy**: This metric reflects the percentage of generated code predictions that exactly match the reference fixed code.
- **BLEU Score**: This measures the similarity between the generated and reference code based on overlapping n-grams.

We compared two configurations:

1. LoRA Fine-Tuned CodeLLaMA (baseline model).
2. GraphRAG + LoRA CodeLLaMA, which incorporates graph-based context retrieval into the generation process.

The table below shows a clear improvement in both metrics when using GraphRAG in combination with LoRA, confirming the effectiveness of integrating contextual information for code repair.

Metric	LoRA Fine-Tuned CodeLLaMA	GraphRAG + LoRA CodeLLaMA
Exact Match Accuracy	32.5%	41.7%
BLEU Score	0.56	0.64

Table 4.3: Performance Comparison Between LoRA and GraphRAG + LoRA

The table 4.3 presents a comparison between two configurations: the LoRA fine-tuned CodeLLaMA model and the combination of GraphRAG with the same model. The results indicate that integrating GraphRAG leads to a notable improvement in performance across both metrics.

Exact Match Accuracy increases from **32.5%** to **41.7%**, indicating a higher percentage of code predictions that exactly match the reference solution. Additionally, the BLEU Score improves from **0.56** to **0.64**, reflecting better n-gram similarity between the generated and reference code. These findings demonstrate that incorporating GraphRAG for context retrieval enhances the model's ability to produce accurate and semantically aligned bug fixes.

6. The Used Programming Languages, Libraries and Tools

6.1. Programming Language

We chose Python programming language due to its dominance in artificial intelligence research and development. Its extensive ecosystem of libraries, especially those designed for machine learning and natural language processing, makes it easy to build powerful solutions with less effort.

As a high-level, general-purpose language, Python is known for its clean syntax and emphasis on readability, which makes writing and understanding code much more intuitive. This is further supported by a strong and active community.

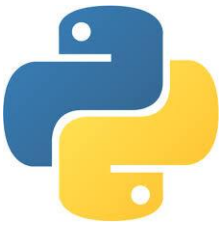

Figure 4.11: Python Logo

6.2 Libraries

- **Transformers**: is a popular open-source library developed by Hugging Face that provides pre-trained natural language processing models, including the CodeLlama families. It is used to build inference applications and train models on specific datasets. The library allows seamless integration and fine-tuning of transformer-based models for tasks like code generation and repair.
- **PEFT (Parameter-Efficient Fine-Tuning):** is a library that enables efficient fine-tuning of large language models using techniques like LoRA. We used it to efficiently fine-tune the CodeLlama-7b model for program repair tasks. It is widely utilized for customizing large pretrained models for various downstream applications.
- **Sentence Transformers:** is a library that enables encoding of textual and code data into dense embeddings using models. It was used in our project to compute semantic similarity between buggy and fixed code during the graph-based augmentation phase.
- **Torch:** is a core component of the PyTorch framework, which is an open-source deep learning library that offers flexible tools for building and training neural networks.
- **NetworkX:** is a Python library for creating, manipulating, and visualizing complex graphs and networks.

• **NumPy:** is a Python library for numerical computing that simplifies working with large arrays and perform complex mathematical operations efficiently.

• **Faiss (Facebook AI Similarity Search):** is a library that helps find the most similar items in huge collections of data, like text or code embeddings. It contains algorithms that search in sets of vectors of any sizeeven when the data is too big to fit in memory.

• **LangChain:** is a Python framework that simplifies building applications with LLMs connecting them to external data sources and tools. In Google Colab, it enables efficient prototyping of advanced NLP workflows that combine pretrained models with custom retrieval and reasoning components.

• **Gradio:** is a Python library that enables fast creation of interactive web interfaces for machine learning models. In this work, it is used to allow users to input buggy code and receive repaired output, facilitating real-time testing and evaluation of the program repair system.

Figure 4.12: Gradio Logo

6.3. Tools and IDEs

6.3.1. Google Colab Pro

Google Colab Pro is a premium version of cloud-based Jupyter notebook environment, offering access to faster GPUs and TPUs, longer runtime sessions, and increased memory compared to the free tier. We used it for all experimentation and fine-tuning steps, especially for training our model.

Figure 4.13: Google Colab Pro Logo

❖ **Reasons for Choosing Colab Pro in our project**

• **High-Performance Hardware:** Provided access to premium GPUs, which allowed us to train LLMs and run inference efficiently.

• **Extended Runtime:** We benefited from longer session durations, allowed us to run fine-tuning and generation tasks without frequent interruptions.

• **Increased RAM:** The high-memory virtual machines in Colab Pro supported loading LLM and datasets in memory without crashes or slowdowns.

• **Hugging Face Integration:** Direct access to models and datasets via the transformers and datasets libraries.

• **Google Drive Integration:** We stored our models, datasets, and experiment outputs persistently using Google Drive.

• **Collaboration and Reproducibility:** Easy to share notebooks and results with collaborators, supporting a reproducible workflow.

6.3.2. Hugging Face Platform

Hugging Face is an essential tool in our workflow. It provides pre-trained models, datasets, and powerful libraries such as transformers, datasets, and PEFT.

We used it to load models like CodeLlama, access benchmark datasets, and implement fine-tuning techniques such as LoRA.

Figure 4.14: Hugging Face Logo

6.3.3. CodeBert-base Model

The microsoft/codebert-base model is used as the embedding generator due to its effectiveness in understanding and representing source code. CodeBERT is a transformer-based model pre-trained on a large corpus of source code and natural language documentation across multiple programming languages. It is specifically designed for code-related tasks such as code search, summarization, and defect detection.

In this work, codebert-base is utilized to generate dense vector embeddings of both buggy and fixed code snippets. These embeddings are stored in a FAISS index to enable efficient similarity search. By incorporating semantically rich code representations, the retrieval component of the GraphRAG framework is able to retrieve relevant and contextually similar code examples, thereby improving the quality and accuracy of automated program repair.

7. Web-Based Interface

To make the automated program repair system accessible and interactive, we use Gradio to build a simple web interface. This interface allows users to input buggy Java code and receive the corrected version in real time.

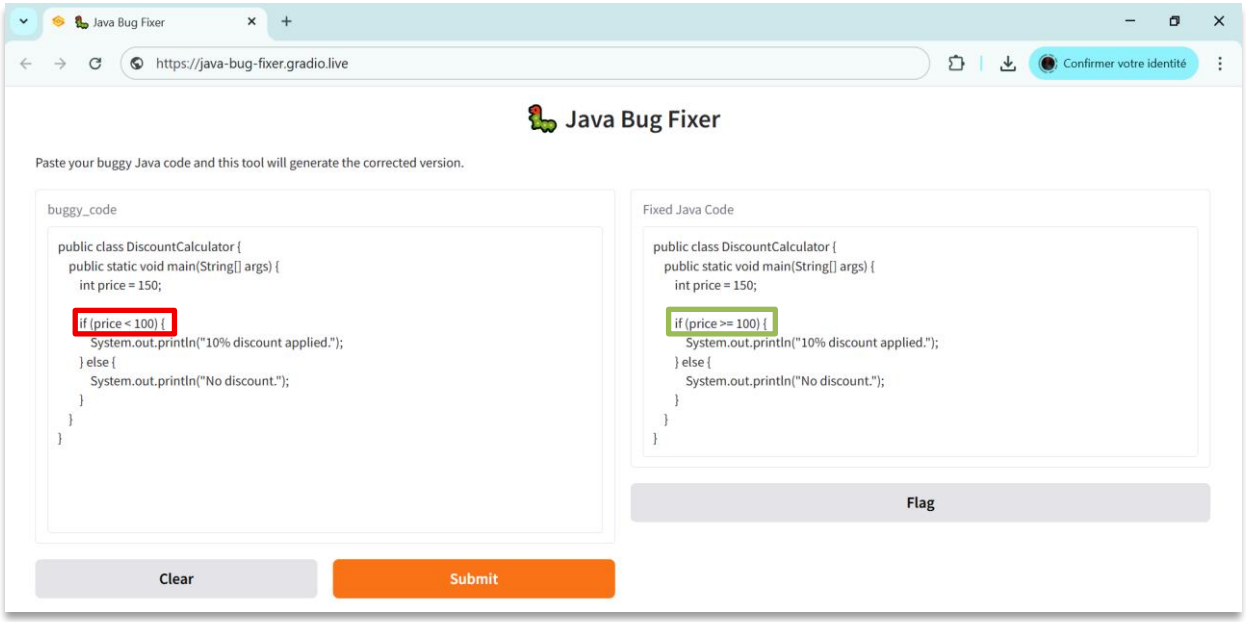

Figure 4.15: Web-Based Interface of the Java Bug Fixer Tool

8. Conclusion

This chapter detailed the step-by-step implementation of our APR approach. We demonstrated how the fine-tuned CodeLLaMA model and GraphRAG retrieval were effectively integrated to form a powerful, context-aware repair pipeline. From dataset preprocessing and embedding generation to graph construction and strategy definition, each component plays a critical role in supporting the model's ability to generate informed and precise code fixes.

General Conclusion

In conclusion, the application of LLMs for APR represents a promising advancement in the domain of software engineering and AI-assisted debugging. By integrating fine-tuned transformer-based architectures with retrieval-augmented techniques such as GraphRAG, our approach demonstrates the capacity of modern LLMs to understand, contextualize, and rectify buggy code with increased precision. This not only enhances the efficiency of debugging workflows but also paves the way to provide real-time code corrections.

Our methodology leverages a fine-tuned CodeLLaMA model trained on a curated subset of the RepairLLama dataset, enhanced with contextual retrieval using semantic embeddings and graph-based strategies. Despite the limitations imposed by constrained computational resources and dataset size, the results validate the feasibility of using instruction-tuned LLMs in conjunction with graph-based retrieval to support meaningful code fixes.

However, there are several limitations to consider. The training process was restricted to a limited number of samples due to constraints in the Colab Pro environment, which affected batch size and training duration. Resource limitations also affected model scalability and experimentation flexibility, highlighting the computational demands of large-scale fine-tuning.

Looking ahead, several directions remain open for future enhancement. Expanding the training dataset to include the full RepairLLama corpus could improve model generalization and robustness. Incorporating more domain-specific embedding models may yield more accurate graph structures for context retrieval. Furthermore, adding functional validation through test case execution would ensure that generated fixes are not only syntactically plausible but also functionally correct.

Ultimately, packaging this approach into an accessible tool or plugin could provide developers with real-time debugging support, significantly accelerating software development and maintenance workflows. Through continued exploration of LLM fine-tuning, graph-enhanced retrieval, and real-world integration, we can push the boundaries of what is possible in intelligent code repair, contributing to more reliable, maintainable, and efficient software systems.

Bibliography

[1] IBM, "AI vs. Machine Learning vs. Deep Learning vs. Neural Networks," IBM Think, [Online]. Available: https://www.ibm.com/think/topics/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks. [Accessed 3 Apr 2025].

[2] O. A. M. López, A. M. López, and J. Crossa, "*Multivariate statistical machine learning methods for genomic prediction"*. 2022. doi: 10.1007/978-3-030-89010-0.

[3] M. Lanhenke, "NLP-Day 1: The Magical Powers of Natural Language Processing," *Medium*, Apr. 7, 2022. [Online]. Available: https://medium.com/@marvinlanhenke/nlp-day-1-the-magical-powers-of-natural-language-processing-56c0208a6b9f, [Accessed 3 Apr 2025].

[4] Sunscrapers, "Deep Learning for NLP - An Overview, " sunscrapers, July. 27, 2023. [Online]. Available: https://sunscrapers.com/blog/deep-learning-for-nlp-an-overview. [Accessed 5 Apr 2025].

[5] A. Bohnert, "10 Types of neural networks explained, " HackerRank Blog, May. 17, 2023. [Online]. Available: https://www.hackerrank.com/blog/types-of-neural-networks. [Accessed 6 Apr 2025].

[6] A. Vaswani *et al.*, "Attention is all you need," *arXiv.org*, *Proceedings of the 31st Conference on Neural Information Processing Systems,* vol. 30, n° %11, Aug. 2, 2017. Available: 1706.03762.

[7] R. Kassel, "Graph Neural Networks (GNN): qu'est-ce que c'est ?," DataScientest, Apr. 25, 2023. [Online]. Available: https://datascientest.com/graph-neural-networkSs. [Accessed 11 Apr 2025].

[8] A. Prasad, "Tokenization in Transformers," Medium, Feb. 27, 2024. [Online]. Available: https://medium.com/@abhijithprasadmkp/tokenization-in-transformers. [Accessed 4 Apr 2025].

[9] G. Lokare, "Preparing Text Data for Transformers: Tokenization, Mapping and Padding," Medium, 10 February 2023. [Online]. Available: https://medium.com/@lokaregns/preparing-text-data-for-transformers-tokenization-mapping-and-padding-9fbfbce28028. [Accessed 22 Apr 2025].

[10] IBM, "What are Large Language Models?," IBM Think, Nov. 2, 2023. [Online]. Available: https://www.ibm.com/think/topics/large-language-models. [Accessed 12 April 2025].

[11] Naveed *et al.*, "A comprehensive overview of large language models," *arXiv.org*, Jul. 12, 2023. Available: https://arxiv.org/pdf/2307.06435.

[12] DataCamp, "Fine-Tuning LLMs: A Guide With Examples," DataCamp, Dec. 4, 2024.

[Online]. Available: https://www.datacamp.com/fine-tuning-large-language-models [Accessed 12 Apr 2025].

[13] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, "Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment," *arXiv.org*, Dec. 19, 2023. Available: 2312.12148.

[14] E. J. Hu *et al.*, "LORA: Low-Rank adaptation of Large Language Models," *arXiv.org*, Oct. 16, 2021. Available: 2106.09685.

[15] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLORA: Efficient Finetuning of Quantized LLMS," arXiv.org, May 23, 2023. Available: 2305.14314.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv.org*, May. 24, 2019. Available: 1810.04805.

[17] Y. Liu *et al.*, "ROBERTA: A robustly optimized BERT pretraining approach," *arXiv.org*, Jul. 26, 2019. Available: 1907.11692.

[18] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," *arXiv.org*, Jul. 22, 2020. Available: 2005.14165.

[19] H. Touvron *et al.*, "LLAMA: Open and Efficient Foundation Language Models," *arXiv.org*, Feb. 27, 2023. Available: 2302.13971.

[20] M. Lewis *et al.*, "BART: Denoising Sequence-to-Sequence Pre-training for natural language generation, Translation, and Comprehension," *arXiv.org*, Oct. 29, 2019. Available: 1910.13461.

[21] C. Raffel *et al.*, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," 2020. *Journal of Machine Learning Research (JMLR),* vol. 21, no. 140, p. 1–67, Available: https://jmlr.org/papers/volume21/20-074/20-074.pdf.

[22] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP tasks," *arXiv.org*, Apr 12, 2021. Available: 2005.11401.

[23] Y. Gao *et al.*, "Retrieval-Augmented Generation for Large Language Models: A survey," *arXiv.org*, Mar. 27, 2024. Available: 2312.10997.

[24] N. A. Uzoka, N. E. Cadet, and N. P. U. Ojukwu, "Leveraging AI-Powered chatbots to enhance customer service efficiency and future opportunities in automated support," *Computer Science & IT Research Journal*, vol. 5, no. 10, pp. 2485–2510, Oct. 2024. Available: 10.51594/csitrj.v5i10.1676.

[25] S. Minaee *et al.*, "Large Language Models: a survey," *arXiv.org*, Feb. 09, 2024. Available: 2402.06196v2.

[26] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for

code Generation," *arXiv.org*, Nov. 10, 2024. Available: 2406.00515.

[27] P. Adragna, "Software debugging techniques," Queen Mary, University of London.

[Online]. Available: https://cds.cern.ch/record/1100526/files/p71.pdf.

[28] B. Siegmund, M. Perscheid, M. Taeumel and R. Hirschfeld, "Studying the Advancement in Debugging Practice of Professional Software Developers," 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, Italy, 2014, pp. 269-274, Available: 10.1109/ISSREW.2014.36.

[29] M. I. W. Santander. "An Interactive Debugging Approach Based on Time-traveling Queries,". Computer Science. Inria, 2023.

[30] Sematext, "What is a memory bottleneck: Definition, Causes & fixes," Sematext, Mar. 19, 2025. https://sematext.com/glossary/memory-bottleneck.

[31] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: An Approach for Debugging Evolving Programs," European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/main-41.pdf.

[32] M. Arab, J. T. Liang, V. Hong, and T. D. LaToza, "How Developers Choose Debugging Strategies for Challenging Web Application Defects," arXiv Preprint, Jan. 20,2025. Available: 2501.11792.

[33] T. D. LaToza, M. Arab, D. Loksa, and A. J. Ko, "Explicit Programming Strategies," arXiv Preprint. Nov. 6, 2019. [Online]. Available: 1911.00046.

[34] D. Spinelli, "Modern Debugging: The Art of Finding a Needle in a Haystack," Communications of the ACM (CACM), Nov. 2018. Available: 10.1145/3186278.

[35] F. Gregor, "Tiny86 Debugger," Czech Technical University in Prague, Faculty of Information Technology, Dept. of Theoretical Computer Science, Apr. 2023.

[36] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," International Symposium on Code Generation and Optimization, 2004. CGO 2004., San Jose, CA, USA, 2004, pp. 75-86. Available: 10.1109/CGO.2004.1281665.

[37] Python Software Foundation, "pdb — The Python Debugger,". [Online]. Available: https://docs.python.org/3/library/pdb.html.

[38] GeeksforGeeks, "Java Debugger (JDB)," *GeeksforGeeks*, Apr. 28, 2025. Available: https://www.geeksforgeeks.org/java-debugger-jdb/. [Accessed 27 Apr 2025].

[39] L. Zhong, Z. Wang, and J. Shang, "Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step," *arXiv.org*, Feb. 25, 2024. Available: 2402.16906.

[40] A. Z. H. Yang, R. Martins, C. L. Goues, and V. J. Hellendoorn, “Large language models for Test-Free fault localization,” *arXiv.org*, Oct. 03, 2023. Available: 2310.01726.

[41] M. Baqar, R. Khanda, and S. Naqvi, “Self-Healing Software Systems: Lessons from Nature, Powered by AI,” *arXiv.org*, Apr. 25, 2025. Available: 2504.20093.

[42] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Guest Editors’ Introduction: Automatic Program Repair," *IEEE Software*, vol. 38, no. 4, pp. 22–27, Jul.–Aug. 2021, doi: 10.1109/MS.2021.3072577.

[43] Q. Zhang *et al.*, “A Systematic Literature Review on large language Models for Automated Program Repair,” *arXiv.org*, May 02, 2024. Available: 2405.01466.

[44] K. Huang *et al.*, “A survey on automated program repair techniques,” *arXiv.org*, Mar. 31, 2023. Available: 2303.18184.

[45] X. Gao, Y. Noller, and A. Roychoudhury, “Program repair,” *arXiv.org*, Nov. 23, 2022. Available: 2211.12787.

[46] K. Wang *et al.*, “Software Development Life Cycle Perspective: A survey of benchmarks for code large language models and agents,” *arXiv.org*, May. 8, 2025. Available: 2505.05283.

[47] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” *ICSE 2023: Proceedings of the 45th International Conference on Software Engineering*, pp. 1482–1494, Dec. 9, 2024. doi: 10.1109/icse48619.2023.00129.

[48] E. Mashhadi and H. Hemmati, “Applying CodeBERT for automated program repair of Java simple bugs,” *arXiv.org*, Mar. 22, 2021. Available: 2103.11626.

[49] A. Silva, S. Fang, and M. Monperrus, “RepairLLAMA: Efficient Representations and Fine-Tuned Adapters for Program repair,” *arXiv.org*, Dec. 25, 2023. Available: 2312.15698.

[50] Y. Wei, C. S. Xia, and L. Zhang, “Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair,” Nov. 30, 2023. Available: 2309.00608.

[51] B. Berabi, J. He, V. Raychev, and M. Vechev, “TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer,” *PMLR*, Jul. 01, 2021.

[52] C. S. Xia, Y. Ding, and L. Zhang, “The plastic surgery hypothesis in the era of large language models,” *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 522–534, Sep. 2023, doi: 10.1109/ase56229.2023.00047.

[53] C. S. Xia and L. Zhang, “Less training, more repairing please: revisiting automated program repair via zero-shot learning,” *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 959–971, Nov. 2022, doi: 10.1145/3540250.3549101.

[54] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and Lin Tan, *CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair*. 2020, p. 14.

[55] A. Kharkar *et al.*, "Learning to reduce false positives in analytic bug detectors," *Proceedings of the 44th International Conference on Software Engineering*, May 2022, doi: 10.1145/3510003.3510153.

[56] W. Wang, Y. Wang, S. Joty, and S. C. H. Hoi, "RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair," *arXiv.org*, Sep. 12, 2023. Available: 2309.06057.

[57] M. Jin *et al.*, "InferFix: End-to-End Program Repair with LLMs," *arXiv.org*, Mar. 13, 2023. Available: 2303.07263.

[58] S. Zhong, J. Zhu, Y. Tian, and X. Si, "RAG-Verus: Repository-Level Program Verification with LLMs using Retrieval Augmented Generation," *arXiv.org*, Feb. 07, 2025. Available: 2502.05344.