MÄLARDALEN UNIVERSITY
SCHOOL OF INNOVATION, DESIGN AND ENGINEERING
VÄSTERÅS, SWEDEN

January 10, 2025

Thesis for the Degree of Bachelor in Computer Science - 15.0 Credits

Achieving self-healing code with LLMs

Johan Bablo Toma
jta20002@student.mdu.se

Supervisor: Alessio Bucaioni
Mälardalen University, Västerås, Sweden

Examiner: Gabriele Gualandi
Mälardalen University, Västerås, Sweden

Abstract

This study investigates LLMs for their potential in achieving self-healing capabilities in code. Ten LLMs were evaluated using a dataset of 76 problems sourced from Leetcode in two programming languages: C++ and Java. The programming problems are categorized into three difficulty levels: easy, medium, and hard. Leetcode classifies the programming problems into different difficulty levels. Each model was given a single attempt to correct the run-time errors, and their effectiveness was measured by their success rate, which calculates their ability to resolve errors and create functioning code. An experiment was conducted LLMs were prompted with Leetcode programming problems and their erroneous solutions, in order to evaluate these LLMs. In light of the findings considerable variations in model performance, with success rates ranging from 30.26% to 94.74%. The unified model approach achieved a moderate success rate of 65.53%. Integrating multiple models within a system can enhance reliability, when addressing coding problems. To ensure the solutions were corrected by the LLMs, manual testing through Leetcode's hidden test cases provided the necessary evaluation framework. However, this method had disadvantages, such as being time-intensive and the possibility of introducing human error into the fray. The results demonstrate the feasibility of LLMs in achieving self-healing code. Even though current models are promising, some challenges remain, especially in integrating LLMs into software. Without a method to integrate the LLMs into actual software, their potential remains theoretical. Future research in this area should expand the dataset to include more problems and programming languages.

Table of Contents

Introduction.......1

Research Plan.......3

Problem Formulation.......3

Research Methodology.......3

Background.......4

Autonomic Computing.......4

Natural Language Understanding.......5

Natural Language Processing.......5

Large Language Models.......6

Leetcode.......7

Related Work.......8

Implementation.......10

Data Collection.......11

Prompt Generation and Engineering.......12

The Process of Sending Prompts to LLMs.......15

The Evaluation Pipeline.......15

Results.......17

How can the effectiveness of each large language model be evaluated and compared to other models with a larger dataset in this research? (RQ1).......17

What is the overall effectiveness of the large language models in achieving self-healing code? (RQ2).......21

Discussion.......22

Societal and Ethical Considerations.......26

Threats to Validity.......26

Conclusions and Future Work.......27

References.......28

Appendix.......31

List of Figure

1 The Implementation Steps....11

2 Data Collection Process....12

3 Prompt Template....13

4 Min Max Game Prompt....14

5 Testing Process....16

6 LLM's Overall Success Rates....17

7 LLM's Easy Success Rates....18

8 LLM's Medium Success Rates....18

9 LLM's Hard Success Rates....19

10 Performance Overview of Different Programming Languages....20

11 Performance Overview of Different Difficulty Levels 20

12 Unified Model Overall Success Rate....21

13 Distribution of Difficulty Levels in Programming Problems....22

14 Distribution of Programming Languages in Programming Problems....23

15 Success Rates across Difficulty Levels....24

16 Success Rates across Programming Languages....24

Introduction

The concept of autonomous machines has captured human imagination for decades, appearing in science fiction, 1980s action movies, and pioneering concepts like those of IBM in the early 2000s. Today, computers are ingrained in our lives [1], and advancements in artificial intelligence (AI) have empowered machines to simulate human intelligence [2]. These advancements enable AI systems to comprehend and respond to human interactions while continuously learning from new and past experiences [2]. AI has revolutionized numerous industries, including finance, healthcare, transportation, and national security [3]. In healthcare, for instance, AI has improved diagnostics, treatment planning, and patient management by utilizing predictive analytics tools capable of forecasting potential outcomes [4]. Although AI encompasses several tech-related sectors, the most notable progress has occurred in the development of large language models (LLMs). Over the past five years, LLMs have rapidly gained prominence due to their exceptional capabilities and widespread adoption. Models like ChatGPT, one of the most significant and fastest-growing LLMs, exemplify this trend. LLMs are versatile, handling tasks ranging from natural language understanding to automated text generation [7]. These abilities stem from their training on vast datasets, equipping them with the capacity to generate coherent, contextually relevant, and accurate content [6]. Notably, automated content generation is a hallmark of generative AI, with LLMs forming a subset of this field. These models possess capabilities such as language comprehension, content generation, and task automation. These attributes align closely with autonomic computing, a paradigm that emphasizes systems capable of self-management with minimal human intervention.

Autonomic computing is guided by four fundamental principles: self-configuration, self-optimization, self-healing, and self-protection [9]. Among these principles, this study focuses solely on self-healing, a capability that enables systems to repair and maintain themselves independently during run-time. In addressing issues dynamically, self-healing systems can ensure software continues to function correctly without human input, even as errors occur. Software maintenance represents one of the most significant expenses for organizations, requiring extensive developer effort to analyze and resolve issues deeply buried within vast codebases. Many of these issues arise during the development phase, and if left unresolved, they can compromise the final product, leading to increased costs, reduced productivity, and potential reputational damage. Self-healing systems powered by LLMs offer a promising solution to this issue by automating error detection and correction. LLMs would then reduce maintenance costs, improve adaptability, enhance productivity, and increase the reliability of the software systems.

The purpose of this study is to evaluate the ability of LLMs to resolve run-time erroneous code in software. This research focuses on ten different models, which are listed in the *Appendix*. For clarity and simplicity reasons, any reference to the models throughout the study pertains to these specific ten models. Each of the model's effectiveness is analyzed in achieving self-healing properties. The evaluation is conducted using a dataset of programming problems and their corresponding solutions, sourced from Leetcode. To ensure diversity in testing, programming problems are randomly selected and verified for run-time errors using Leetcode's built-in environment and associated test cases. Additionally, solutions from a prior study in the same field are included to expand the dataset for analysis [27]. Each LLM has a single attempt to solve the run-time errors in a solution. The outcome of these attempts from the different models is

categorized as either a success or a failure. In the case of a failure, additional details are documented, such as the type of failure, which includes run-time errors, redefinitions, memory errors, or various compile-time errors. Outcomes are documented in a Google Sheet, detailing information such as the programming problem, programming language, difficulty level, and the model's outcome. The programming languages selected for this study are C++ and Java, enabling a comparative analysis of the models' effectiveness in resolving run-time errors. Effectiveness is measured by the success rate of each model, representing how successfully the model has resolved run-time errors in the solutions. The success rate is calculated as a percentage by dividing the number of problems successfully solved by the model by the total number of programming problems in the study, then multiplying the result by a hundred. This approach not only enables a comparison of the LLMs' effectiveness in correcting run-time errors but also provides a clear and quantitative evaluation of their potential in achieving self-healing properties. Additionally, it allows for the calculation of an overall success rate or average performance if all the LLMs were combined into a single unified model.

Research Plan

In this section, I will explain the research methodologies, why they were chosen and how they were applied. This will be followed by a discussion on the research questions (RQs) and their relevance to the study, effectively addressing the problem formulation.

Problem Formulation

The objective of this thesis is to investigate if LLMs can be utilized in the function of self-healing for applications. As AI and LLMs continue to expand, the interest in autonomic computing grows larger due to current infrastructure limitations. My research strives to evaluate the effectiveness of LLMs in correcting run-time errors, as I seek out to see if these models are capable of autonomously healing erroneous code without human intervention. Therefore with this in mind these RQs can be posed:

> **RQ 1** *How can the effectiveness of each large language model be evaluated and compared to other models with a larger dataset in this research?* Answering this RQ will provide a framework for evaluating success rate of each large language model, allowing for comparative analysis of their strengths and weaknesses in self-healing functions.
>
> **RQ 2** *What is the overall effectiveness of the large language models in achieving self-healing code?* Answering this RQ will provide insights regarding large language models and their cumulative potential for automating the self-healing process.

To answer the RQs above, the method of evaluating the overall and individual effectiveness of an LLM is based on that LLMs success rate in correcting run-time errors across various programming languages and difficulty levels.

Research Methodology

The methodologies employed in this thesis are consistent with the principles of experimentation and quantitative research. The experimental research approach focuses on manipulating variables and observing their outcomes within a controlled environment [28]. According to Em [28], the foundational principles of experimental research include variable manipulation, controlled environments, random assignment, and the ability to infer causality. Experimentation, in this context, involves exploring causal relationships by altering one or more independent variables. In this study the independent variable is represented by the different LLMs, while the dependent variable is the observed success rates of the LLMs in resolving run-time errors in Leetcode solutions. In addition to the experimental research approach, this study incorporates quantitative research methods, as it involves the collection and analysis of numerical data. The success rates of the LLMs expressed as percentages, form the basis of this data. Quantitative research focuses on identifying patterns, testing hypotheses, and making predictions based on statistical analysis [29]. The importance of a quantitative research approach lies in its ability to process and interpret data through statistical analysis. The use of statistical methods in this study is employed to make comparisons, generalizations and draw conclusions based on the success rates of the various LLMs, while also taking into account factors such as programming languages and difficulty levels.

This research builds upon previous work, which involved a smaller sample size of 22 problems, excluding the "construct binary tree" problem, all of which are also used in this study. The dataset is taken from Bäverlind's public GitHub repository[1], where their C++ solutions were shared. In this study, these solutions serve a similar purpose: to evaluate the self-healing capabilities of LLMs by testing their ability to correct run-time errors. The key difference between this study and Bäverlind's research lies in the testing methodology. Bäverlind developed an application, which they called for "statistics builder", to simulate the actions of an autonomous agent that monitors and corrects run-time errors using an LLM [27]. The statistics builder allowed Bäverlind to evaluate the effectiveness of the agent's self-healing process. In contrast, my research does not follow this approach, since the statistics builder only uses 2-5 visible test cases disclosed by Leetcode. However, once a solution is submitted, Leetcode runs it against hidden test cases that can range from a few to over a thousand, covering a wider range of potential errors. When a solution fails, Leetcode reveals the specific test case at which it failed and the total number of hidden test cases, although it does not provide access to the full set of test cases. My approach involves a manual process where each solution is submitted and tested directly on Leetcode, allowing it to be evaluated against all hidden test cases to ensure run-time errors are corrected, effectively. Even though my research employs a manual approach rather than a fully autonomous system, it still demonstrates the potential of LLMs in achieving self-healing code.

Since it was not possible to access all test cases from Leetcode, creating an application to autonomously test for any remaining run-time errors in the solutions is not feasible. This testing process, while not automated, utilizes the hidden test cases provided by Leetcode to thoroughly assess the potential self-healing capabilities of LLMs. It exposes LLMs to a wide variety of hidden errors beyond basic functionality checks, enabling a more comprehensive evaluation of the models' problem-solving skills, reasoning, and adaptability in addressing these coding problems. Leetcode's undisclosed test cases provide a more valuable and rigorous testing environment than custom-built test cases, specifically from someone without experience in designing test cases. Attempting to create equivalent test cases would not only lack the same depth but also require tremendous focus, effort, and time to achieve the same outcome. While previous studies have explored the capabilities of LLMs in tasks like code generation and bug detection, their potential for achieving self-healing properties remains largely unexplored. This study provides a larger dataset, multiple programming languages, and includes a greater variety of models to enhance the comparative analysis of LLMs in achieving self-healing properties.

Background

In this section, I will explain concepts foundational to this thesis such as autonomic computing, natural language understanding and processing, large language models, and Leetcode.

Autonomic Computing

The concept of autonomic computing is that systems should have the ability to manage, maintain and resolve system problems by themselves [9]. Autonomic computing was coined by IBM in the early 2000's, driven by the need to address the growing complexity of computing infrastructure, which was struggling to keep up with modern applications. The human autonomic nervous system,

[1] https://github.com/MartenStromBaverlind/ThesisCode

along with other man-made systems such as aircraft autopilot and self-regulating power grids, served as inspiration for autonomic computing [10]. Modern systems are increasingly complex due to the integration of new technologies and different system components, making them difficult to manage. Therefore, the need for autonomic computing surfaces from the challenge of managing these ever-growing and sophisticated systems, both past and present [10]. The core tenets of autonomic computing are self-healing, self-optimization, self-protection and self-configuration [9]. Self-healing refers to an autonomic system's ability to repair itself from errors, with the capability to detect and heal faulty components without causing any further harm to the system [15]. Self-optimization guarantees that the system continually improves its performance through enforcing efficient algorithms for all computing operations [15]. In addition to these, self-protection safeguards the system's integrity by identifying and addressing security concerns [15]. Lastly, self-configuration enables the system to adapt to changes in its environment, automatically adjusting its settings and replacing outdated or faulty components with minimal human intervention [15]. In this thesis, the focus lies on autonomic computing, specifically the self-healing aspect, to explore if self-healing can be achieved using LLMs.

Natural Language Understanding

Natural language understanding (NLU) is a subfield of natural language processing (NLP), which itself falls under the umbrella of AI. NLU has many use cases, in terms of applications such as chatbots, voice assistants, and automated translation services [11]. A deeper look into the components of NLU uncovers parsings, which take text written in natural language and convert it into a structured format that computers can understand [11]. However, parsing is not the only component of NLU; it also includes sentiment analysis, named entity recognition, and semantic role labeling. Sentiment analysis classifies chunks of text with regard to its sentiment, which can be positive, negative, or neutral [12]. Another key aspect is the ability to identify predefined categories in bodies of text, which refers to named entity recognition [13], while semantic role labeling identifies the underlying relationships in sentences, such as who did what to whom [14]. The components of NLU are a fundamental part of LLMs, as they provide the capability to process text data.

Natural Language Processing

NLP focuses on enabling computers to process and comprehend language similarly to humans [16]. Machine learning algorithms and deep learning are combined to power the computational linguistics of NLP [8]. Moreover, language models are reshaping traditional text analytics, with ChatGPT being the first model capable of doing advanced tasks like programming and solving math problems [16]. The work process of NLP is characterized by analyzing relationships between language segments such as letters, words, and sentences [17]. However, this process consists of multiple methods for data preprocessing, feature extraction, and modeling. Data preprocessing refers to processing text with a specific task in mind, starting with stemming and followed by lemmatization, which converts words to their base forms [17]. This process continues with sentence segmentation, where text is broken into meaningful units, in the case of the English language punctuation (.) indicates the end of a sentence [17]. Stop word removal discards commonly occurring words that do not contribute new information to the text, and tokenization splits the text into individual words or word fragments. Furthermore, tokenization creates a word index or tokenized text, representing words as numerical tokens [17]. Techniques like Bag-of-Words and TF-IDF are used to transform

text into numerical features for machine learning tasks, recent techniques such as Word2Vec and GLoVE learn features during the training of neural networks [17].

TF-IDF (Term Frequency-Inverse Document Frequency) defines a word's importance in a document relative to a corpus. The term frequency (TF) tracks how often a word appears in a document, whereas inverse document frequency (IDF) measures the word's importance or rarity across the entire corpus [17]. Models like Word2Vec and GLoVE create word embeddings that capture semantic meaning, albeit their methods in capturing semantic meanings differ. Word2Vec uses vanilla neural networks to learn high-dimensional words from raw text but GLoVE relies on matrix factorization rather than neural learning [17]. Currently, many applications use NLP, for instance, autocomplete in search engines, chatbots, voice assistants, language translators, and more. These are a few of the popular use cases, as NLP can also be used for monitoring and survey purposes. However, NLP still faces its own challenges with bias and incoherence, and at times its behavior is erratic [17]. Nevertheless, the future of NLP is promising, with further improvements in fields like neural learning likely to improve its accuracy. NLP allows LLMs to process and understand natural language, without it, for instance, LLMs would not be capable of reading code.

Large Language Models

In recent years, LLMs have become widely recognized by the general public as generative AI matures and advances due to continuous developments within the field. Many of these improvements within the field occur at NLU and NLP levels, refining LLMs' contextual awareness, improving their understanding of user preferences, and reducing errors in reasoning and language generation. As NLP and NLU improve, they directly impact LLMs, enabling a wider range of use cases [6].

LLMs are classified as a branch of foundation models, defined as AI neural networks trained on raw data, generally through unsupervised learning [18]. Machine learning algorithms are categorized into supervised and unsupervised learning methods. Alpaydin illustrates supervised learning through an example involving family cars, which are shown to a group of people to gauge their sentiments, whether positive or negative, toward the cars [19, p.21]. The exchange of positive or negative sentiment occurs if the individuals believe the cars they were shown are family cars. Moreover, Alpaydin explains that class learning involves the act of finding a representation shared by all positive examples excluding the negative ones. Once a new vehicle is introduced, predictions are based on the learned description [19, p.21]. According to Alpaydin, supervised learning is the process of learning a class from positive and negative examples [19, p.21]. In unsupervised learning, there is no supervisor that can guide the process towards the correct path when problems arise. The goal, nonetheless, is to discover patterns in the input data, as certain patterns tend to occur more often than others; density estimation [19, p.11]. Clustering, a method of density estimation, aims to find groupings within the input data. Additionally, clustering groups data based on shared attributes [19, p.11]. In LLMs, this technique helps models identify structures within language, it improves word embeddings, text summarization, and topic modeling.

Unsupervised learning grants LLMs the capability of understanding language patterns, which is then reinforced through fine-tuning with supervised learning to improve performance on specific tasks. Transfer learning refers to how models are able to apply their knowledge from one domain or task to another, even when labeled data in the new domain is limited [21]. This reduces data requirements while maintaining performance across various tasks. Zero-shot learning, a subfield of

transfer learning, expands on the concept through enabling models to handle tasks or recognize new categories without prior examples during training [21]. Without zero-shot learning, models would struggle to generalize unfamiliar situations. Another distinct paradigm is reinforcement learning, it enhances LLMs by improving their decision-making and reasoning. In reinforcement learning, an agent learns to map situations to actions, with the intent to maximize the rewards the agent receives [22]. Unlike supervised learning, the agent in reinforcement learning discovers what the most optimal route is through trial and error. Although this might seem inefficient, it improves the agent's decision-making in uncertain environments, with applications in fields such as robotics and self-driving vehicles. Deep-thinking models, such as OpenAI's o1, represent a new type of LLM that commits more time to thinking through problems before responding [20]. OpenAI's o1 model excels in STEM topics as complex problem-solving and reasoning are fundamental [20]. As Devlin et al. [23] discuss, large-scale pre-training for language understanding forms the foundation for further innovations in LLMs, in areas such as reasoning, problem-solving and code correction. In Devlin et al. study, this is evident in ChatGPT-4 and CodeT5+. Understanding how LLMs process and generalize information through techniques like transfer learning, zero-shot learning and reinforcement learning provides the groundwork for exploring how these models can autonomously fix run-time errors.

Leetcode

Leetcode is a website intended for people who want to improve and practice their coding skills through a variety of coding challenges. These programming challenges are categorized by topic and difficulty levels, the topics range from arrays to dynamic programming. Whilst there are three difficulty levels to choose from: easy, medium and hard. Leetcode and similarly designed websites' purpose is to be used by programmers to prepare for their technical interviews at tech companies. Each challenge includes a description that defines the task at hand for the user to solve. Once a challenge has been solved by the user, the solution is tested against multiple unique test cases, if the solution succeeds the user receives detailed feedback like memory usage and run-time. The metrics received can be used to compare with those of other users, allowing you to see how your performance fared. Leetcode serves as the dataset for evaluating the self-healing capabilities of LLMs by providing the various programming problems that allow the models to attempt and correct run-time errors.

Related Work

This section I will review and discuss studies that have explored the use of LLMs in software systems to attain self-healing properties.

In this study, Tihanyi et al. [24] analyze whether LLMs with formal verification strategies can enable automatic software vulnerability repairs. They initially used BMC (Bounded Model Checking) to identify the software vulnerabilities and collect data from BMC that was later fed into an LLM to fix the code. To verify that the code was error free, BMC would also be utilized in this aspect. ESBMC (Efficient SMT-based Context-Bounded Model Checker) was used alongside the LLMs. The researchers proposed a framework ESBMC-AI, to detect and correct errors in C programs. My thesis differs from theirs in that LLMs are investigated for their self-healing properties by resolving run-time errors in Leetcode solutions across various programming languages. This approach does not involve formal verification strategies, such as BMC. Ekedahl et al. [25] investigated whether ChatGPT can replace programmers by conducting an experiment using Leetcode problems. ChatGPT is fed a wide range of programming problems, with various difficulty levels and topics. To evaluate the potential of ChatGPT in replacing programmers, data like run-time performance, memory usage and code correctness is taken into consideration. This study is somewhat similar to my research, as both use programming problems from Leetcode. However they differ as Ekedahl et al. [25] focuses on the performance of ChatGPT code generation in comparison to human programmers. While the subject of this thesis is the potential of LLMs to achieve self-healing properties within applications, it does not focus as intently on their problem-solving capabilities. Notably, in my research, the models have only one attempt to resolve the run-time errors in the solutions. In contrast, Ekedahl et al. [25] allowed up to three attempts to solve the programming problems.

Zhang et al. [26] explores LLMs and how they can be utilized in realizing the vision of autonomic computing, strictly focusing on managing microservices in cloud environments. The researchers proposed a LLM-based multi-agent framework, which combined elements from autonomic computing to handle management tasks [26]. Low-level agents are responsible for managing service components by monitoring their own health, analyzing potential issues and autonomously enforcing procedures to resolve these issues [26]. On the other hand, high-level group managers oversee entire groups of microservices and coordinate their response to incidents within the cloud environments [26]. Additionally, the high-level managers break down tasks into smaller subtasks that the low-level agents can address. This process is enforced through the Plan-Execute feedback loop, where high-level managers create detailed step-by-step instructions for low-level agents to follow, while they monitor the progress of these tasks [26]. Zhang et al. [26] used Sock Shop, an ecommerce application that simulates real-world operations. This was carried out to demonstrate their framework. Each microservice, such as front-end or payment services are managed by a low-level agent, whilst the high-level manager monitors these services [26]. In the same spirit, my research explores autonomic computing, albeit focusing on one tenet of autonomic computing: self-healing. Zhang et al. [26] focused on large-scale cloud environments and how LLMs are applicable in that environment to achieve the vision of autonomic computing. Their research had a wider scope, encompassing all tenets of autonomic computing, with the main goal to achieve the vision of autonomic computing.

A study by Bäverlind aims to address whether LLMs can be used to enable self-healing code [27]. In this research the ChatGPT-3.5 model is used to correct run-time errors in C++ applications, which were randomly selected user solutions from Leetcode [27]. Once the model has corrected the solution, it is retested using Leetcode's test cases to verify whether the errors were resolved. For this purpose a Python application was developed by Bäverlind to automate the testing process. The Python application reads the user submitted solutions stored in a JSON-file, compiles each solution and runs them against test cases to trigger run-time errors [27]. If a run-time error is triggered, a prompt is generated for ChatGPT, detailing information such as the problem description, source code, and the test cases [27]. The solution is retested against the same test cases, if the errors persist, the healing process is considered a failure, and the results are stored in a text file for users to verify [27]. In both my thesis and Bäverlind's, the end goal is the same: to determine whether LLMs can be used to enable self-healing code. Although there are key differences, such as sample size and programming language, my research includes a total of 76 solutions across two programming languages, C++ and Java, whereas their study had only 22 solutions, all in C++. Furthermore, ten different models were used in my experiments, whereas Bäverlind's study relied solely on ChatGPT-3.5. Another key distinction between our research lies in our testing approaches: my methodology utilized the hidden test cases of Leetcode's submission environment. In contrast, Bäverlind used the visible test cases designed for basic functionality checks. My approach was manual: each solution was submitted through Leetcode's built-in environment, which contains more test cases that the solutions must pass to be considered successful. Bäverlind's research serves as a foundational starting point for my study. This research builds upon Bäverlind's work by comparing and analyzing a larger sample size of solutions, programming languages, and models.

Implementation

This section describes the steps taken in this research, covering key elements of the implementation. It begins with data collection, followed by prompt generation, the development of applications to handle communication with the LLMs, and the evaluation process of the code generated by the LLMs.

For this research, several key tools were utilized, including Leetcode, Google Sheets, OpenRouter API, OpenAI API, Canva[2], and various Python applications and JSON-files. Access to the various models was facilitated through the APIs, the models are listed in the *Appendix A*. To enable the use of these APIs, credits were purchased totaling €20. However, a smaller amount could suffice, as much of the cost came while learning how these APIs function. Each of these tools were essential for conducting experiments, solving erroneous solutions, and analyzing the performance of LLMs. The dataset for this study was sourced from Leetcode and a prior study [27], consisting of programming problems and their solutions with run-time errors. Although other details, such as the problem name, description, programming language, difficulty level, additional information, and unique integer identifier, were documented within two separate JSON-files: one for Java and one for C++. Prompts were generated for each problem in the datasets using a Python application for that purpose. These prompts were subsequently stored in a separate set of JSON-files. The prompts were then sent to the different LLMs to correct the run-time errors solutions. Once these solutions had been corrected, they were stored in a separate set of JSON-files. This process is repeated multiple times across ten different LLMs. Each model has its own set of JSON-files to differentiate them from other models. The file-naming convention also includes the model name that solved the solution. Each solution within the resulting JSON-files is tested through Leetcode's built-in environment according to their submission criteria. The solutions have only two possible outcomes: success or failure. Both are documented in a Google Sheet, which logs the outcomes for each LLM in this thesis across various programming problems. The Google Sheet also includes information such as the problem name, programming language, and difficulty level. This simplifies the process of creating graphs and figures from the data. By linking the problem name and programming language, the Google Sheet and JSON-files ensure that the data is easily accessible, which facilitates effective analysis of the results from solution submissions in the Leetcode environment. As shown in *Figure 1*, the implementation can be summarized into four main steps.

[2] https://www.canva.com/

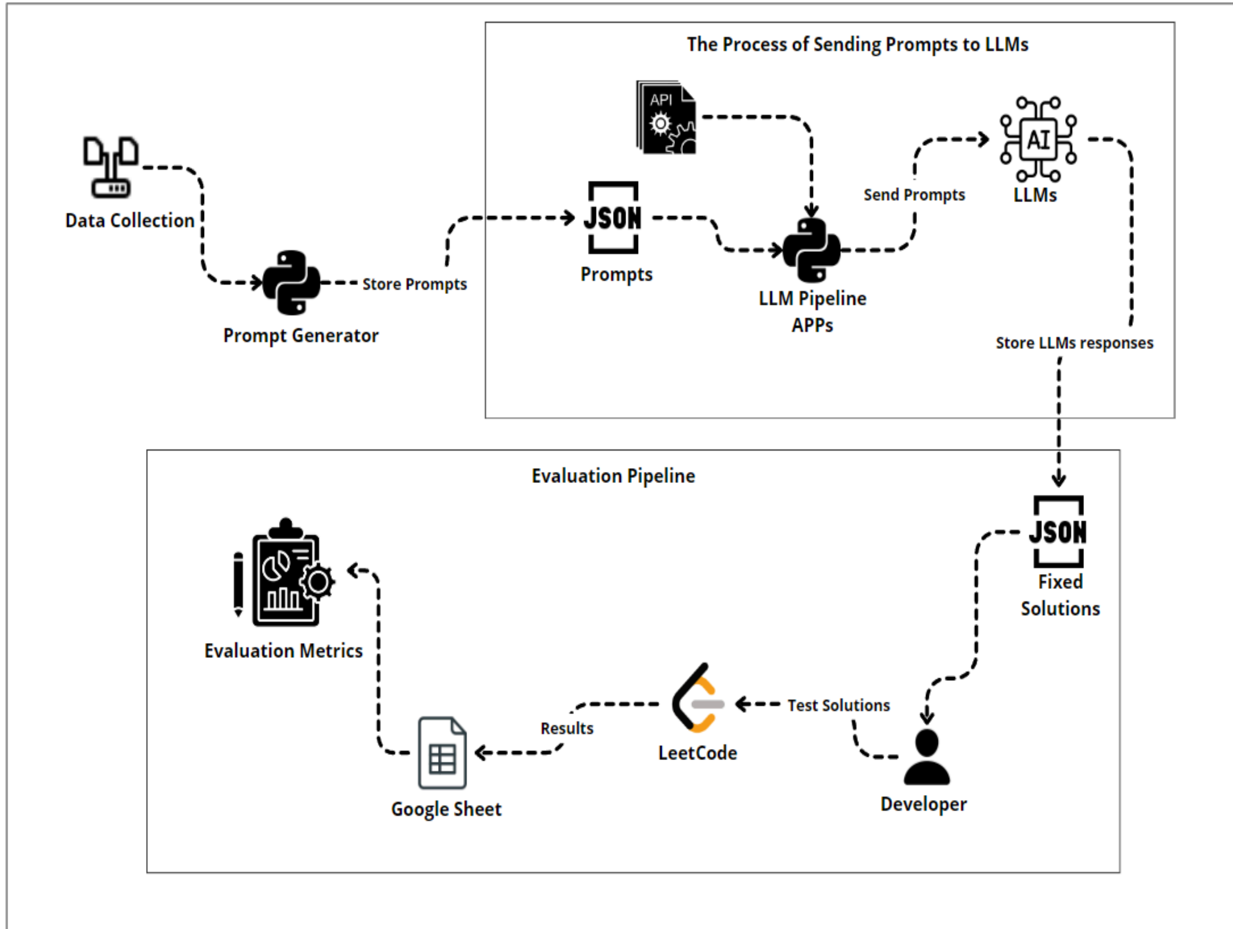

Figure 1: The Implementation Steps

- Programming problems and their run-time error solutions gathered from Leetcode and stored in JSON-files, categorized by programming language.
- Python-based applications were developed to generate prompts for each problem.
- Prompts were sent to LLMs through Python applications using APIs to enable the interaction.
- Corrected solutions are submitted to Leetcode for validation against hidden test cases, with results documented in Google Sheets for analysis.

Data Collection

Leetcode served as the source for the data collection step, and Bäverlind's C++ solutions were also used in combination for this study. My goal was to find code containing run-time errors, although many errors were compiler-related and had to be ignored. Programming problems were selected randomly, without a predefined methodology. My process involved selecting as many programming

problems as possible until my browser tabs slowed down my computer. The number of programming problems to review is determined by the individual and can be as many as they desire or limited by their computer's or device's memory capacity. Thereafter, I would filter solutions by the most recent postings and then focus on those written in Java or C++. For storing programming problems, two JSON-files were created: one for Java and the other for C++. Once a solution with the correct type of error was found, it was documented in one of two JSON-files, noting the problem's name, description, solution, programming language, difficulty, and assigning it a unique numerical identifier. Before adding a solution to the JSON-file, data preprocessing took place to ensure the code was clean, any personal identifiers or comments from the code were removed. This was also to ensure that the comments would not affect the outcome when sent to an LLM for correction. Initially, these files were stored locally, but they are now available in my GitHub repository[3]. Each of the user submitted solutions was manually reviewed and tested on Leetcode built-in compiler then stored in my JSON-files. This approach was taken due to the lack of a public API for gathering user submissions. Additionally, Leetcode's bot-detection measures prevented me from automating data collection with tools like Selenium[4], a python library commonly used for web scraping.

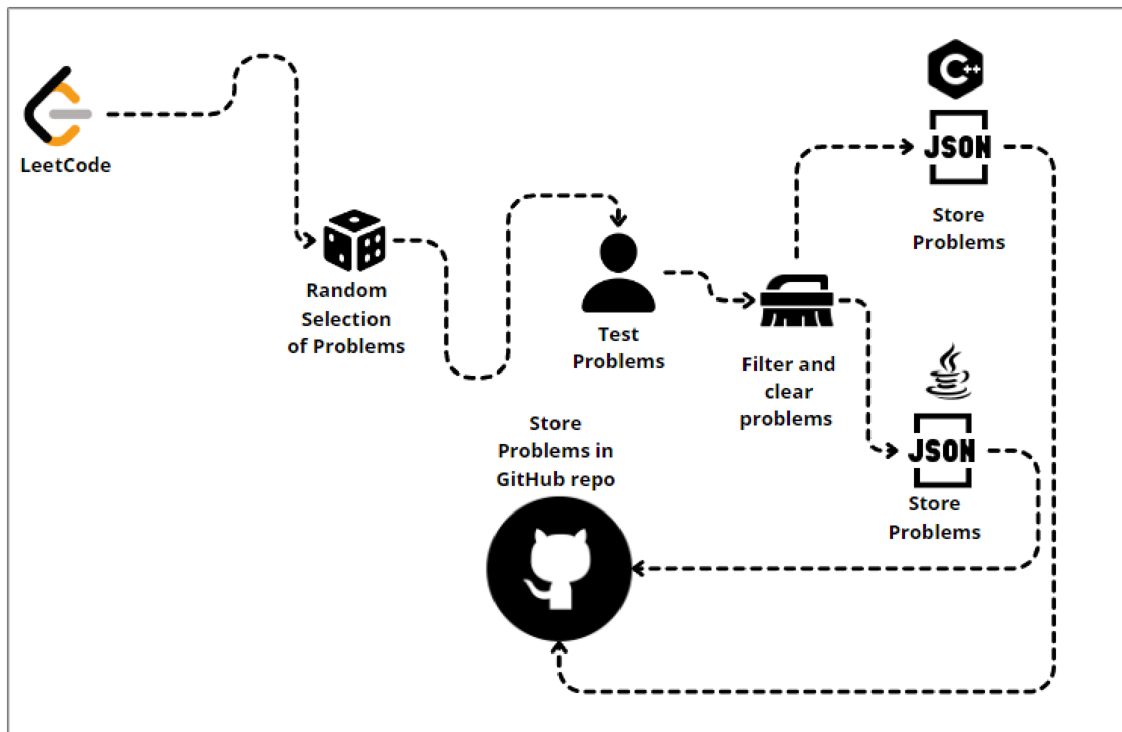

Figure 2: Data Collection Process

Prompt Generation and Engineering

Creating clear and effective prompts for LLMs is necessary to obtain accurate, useful and consistent outputs; the quality of the output is defined by the quality of the prompt provided [30]. This process, referred to as prompt engineering, involves designing the input data in a way that guides

[3] https://github.com/johan-toma/Self-Healing-Thesis
[4] https://www.selenium.dev/documentation/

the model to generate responses that align with the task at hand [30]. In this research prompt engineering played a vital role in ensuring that LLMs corrected the run-time errors within the solutions. A well-crafted prompt can ensure that AI-generated output aligns with the desired outcomes and criteria [30]. In general, prompt engineering aims to bridge the gap between raw queries and meaningful AI-generated responses. LLMs rely on these prompts to interpret queries accurately and produce results. The prompts were structured to ensure that the LLMs focus on solving run-time errors, given key details of the programming problems to provide the necessary information. Initially, a key component of prompting the LLMs was not a part of it: the system prompts. These system prompts set the model's behavior for addressing erroneous solutions, providing general instructions that influenced the outcome. The purpose was to avoid pitfalls such as general conversation and summarization that LLMs typically generate and instead focus solely on delivering the correct solution. Additionally, system prompts had another purpose to simulate the interactions an LLM would have had within an application, making it necessary for the model to return only the code. A visual representation of the prompt to be sent to the LLMs is shown below:

I have a set of programming problems with provided solutions, each of these solutions contains run-time errors. For each problem, please analyze the code to identify the source of the error and fix it. Provide a corrected version of the solution given. For each problem only give the code solution in response, do not give any details regarding to what was fixed, why and how.

Please help me solve this code with: {Programming Language}

{Programming Problem Name}

This is the programming problem description: {Problem Description}

This code contains a run-time error.
{Additional Information}
{Run-time Error Solution}

Figure 3 : Prompt Template

To avoid any ambiguities, emphasis is placed on the use of precise language. This approach streamlined the testing process by allowing solutions to be taken directly from the JSON-files and tested immediately, with minimal work required. In designing the prompts, inspired by Yin et al. [31] findings that overly impolite prompts may degrade performance, while moderate politeness enhances comprehension without compromising task focus. Therefore, my prompts included polite language to guide the LLM in a way that possibly improves the error correction. The additional information field seems out of place but the reason it is a part of the JSON-file is to serve as a boilerplate for tools like structs provided by Leetcode. These structs can include nodes used in lists. Leetcode provides these structs to aid users in solving programming problems. When such information is available in a problem, it is also provided to the LLMs to aid in their goal to resolve run-time errors.

With a foundation in prompt engineering established, the second step can be taken which is the development of the prompt generation application. In generating prompts, the format followed is shown in *Figure 3*, while *Figure 4* provides a more nuanced example of how the prompts are formatted after being processed through the prompt generator.

I have a set of programming problems with provided solutions, each of these solutions contains run-time errors. For each problem, please analyze the code to identify the source of the error and fix it. Provide a corrected version of the solution given. For each problem only give the code solution in response, do not give any details regarding to what was fixed, why and how.

Please help me solve this code with **cpp**

Min Max Game

This is the programming problems desc: **Given an array nums, apply a specified algorithm to transform it and return the last remaining number**.

This code contains a run-time error.

```
class Solution {
    public:
    int minMaxGame(vector<int>& nums) {
        int n = nums.size();
        int maxx = 0;
        int minn = 0;
        vector<int> newNum(n / 2, 0);
        if (n == 1) {
            return newNum[0];
        }
        for (int i = 0; i < n / 2; i++) {
            if (i % 2 == 0) {
                minn = min(nums[2 * i], nums[2 * i + 1]);
                newNum.push_back(minn);
            } else {
                maxx = max(nums[2 * i], nums[2 * i + 1]);
                newNum.push_back(maxx);
            }
        }
        minMaxGame(newNum);
        return newNum[0];
    }
};
```

Figure 4: Min Max Game Prompt

After the data collection step, the prompt generator application was developed. The development of this application stemmed from the desire to hasten the prompt generation process, facilitating quicker interaction with the LLMs. The JSON-files created to store Java and C++ solutions are processed through the prompt generator to create a prompt for each problem in those files. A key priority in the development of the application was to ensure it could read data directly from JSON-files and generate prompts for each problem. These prompts are stored in a set of JSON-files, one for Java and one for C++. In these JSON-files, include key-value pairs such as the prompt data, problem's name, and a unique numeric identifier starting at zero. Built entirely with basic Python code, the prompt generator application relied on standard libraries available by default. The only additional libraries used was json, as the applications mainly handled JSON-files throughout.

The Process of Sending Prompts to LLMs

After generating the prompts for the entire dataset and storing them in two respective JSON-files, the third step involves sending the prompt data to the LLMs. To achieve this, a method for sending prompts to the different LLMs to correct run-time errors must be developed. This process must be easily repeatable across ten different LLMs. Therefore, two Python applications were created with this in mind. Since it would be too slow sending prompts individually through their respective platforms. Hence, this process was automated, allowing each prompt to be sent sequentially through Python and receive responses containing the corrected solutions. To store these corrected solutions, a separate set of JSON-files is used. Each model, however, has its own set of JSON-files to differentiate them from other models. The file-naming convention included the model name that solved the solution. Trying to send prompts to various LLMs using Python requires using the LLMs API. Initially, OpenAI API was used to communicate with ChatGPT-4o and ChatGPT-4o-mini models. However, it became apparent that a streamlined solution was necessary since continuously adding €5 or €10 in credits would become too costly. Which, in turn, led me to find a solution with API access to multiple models, OpenRouter. Functionally, the application developed using OpenAI API operates similarly to the OpenRouter API application used for other models, slight difference in the coding structure due to different syntax and implementation. Although, the overall process, such as communicating with the LLMs remained the same. Both OpenAI[5] and OpenRouter[6] has readily available documentation, which was useful setting up the API for the first time and whenever issues were encountered.

The Evaluation Pipeline

The final step involves implementing an evaluation pipeline to assess the self-healing capabilities of LLMs in correcting run-time errors. However, the approach was not automated, as this was infeasible due to the hidden nature of test cases on Leetcode. A previous study used the visible test cases within Leetcode, typically ranging from two to five cases, designed primarily to test basic functionality of a solution. Notably, this is insufficient, for this reason it was circumvented by submitting a solution through Leetcode's submission environment, which evaluates solutions using anywhere from a few to thousands of test cases. In total, the programming problems, including

[5] https://platform.openai.com/docs/api-reference/introduction
[6] https://openrouter.ai/docs/quick-start

both Java and C++, amount to 76. With the third step finalized, each LLM's response stored in their respective set of JSON-files will be tested through Leetcode's built-in coding environment according to their submission criteria. The results are documented in a Google Sheet, which includes details for each programming problem, such as its name, programming language, difficulty level, and outcomes. In the outcome section, provides details on whether the solution was a failure or successful for a specific model, the solution must meet all submission criteria without any errors. When dealing with a solution marked as a failure in Google Sheets, includes also the type of error that caused the failure. These types of failures included redefinitions, incomplete answers, incorrect answers, run-time errors, and various compile-time errors.

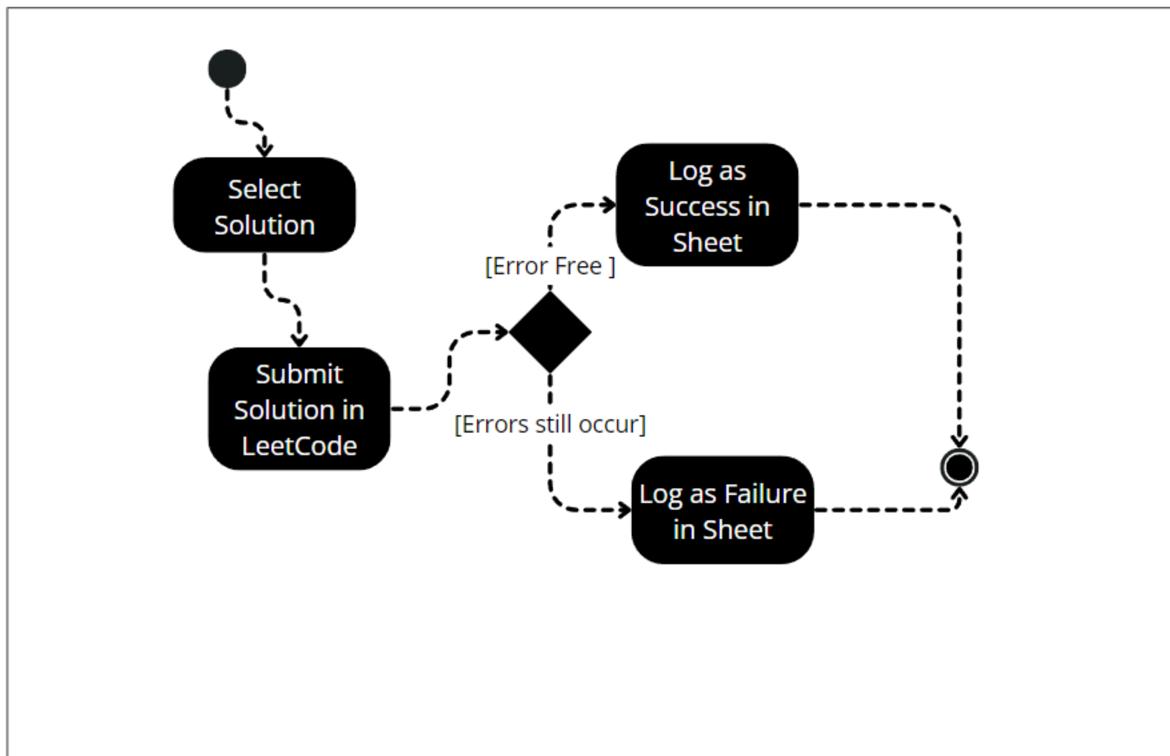

Figure 5: Testing Process

Figure 5 presents the manual testing process, where each solution is tested and their outcomes logged in the Google Sheet. The Google Sheets ensures data is well-organized and easily accessible, ensuring efficient analysis of the results from submitting solutions through the Leetcode environment.

Results

In this section, the findings of the study are presented, based on the RQs outlined earlier in the *Research Plan* section. The results provide an overview of the model's performance in achieving self-healing code, analyzed across different metrics, difficulty levels and programming languages.

How can the effectiveness of each large language model be evaluated and compared to other models with a larger dataset in this research? (RQ1)

According to RQ1, how can the performance of LLMs be evaluated and compared with other LLMs when tasked with solving run-time errors in code. To determine the performance or effectiveness of each LLM, success rate is used as a key metric. The success rate is calculated through the Google Sheets containing the results of the LLMs attempt in solving run-time erroneous code. Each LLMs performance or effectiveness is tracked based on whether it successfully resolved the errors, with the results documented in the Google Sheet. Represented as a percentage, the success rate is calculated by dividing the number of problems successfully solved by the total number of problems the LLM attempted. This can be represented mathematically as follows:

$$Success\ Rate\ (\%) = \frac{Total\ \ of\ Successful\ Solutions}{Total\ Number\ of\ Programming\ Problems} \cdot 100$$

Calculating the individual success rate for each LLM simplifies the process of addressing **RQ1.** These success rate percentages, representing each LLM's performance, can then be compared across models in this study to determine which has the best overall performance.

Displayed in *Figure 6* are the corresponding success rates, presented in an overall format for each model included in the research. Starting with ChatGPT-4o achieved a success rate of 72.36%, while the 4o-mini model had 71.11%. Claude 3.5 Sonnet and Claude 3.5 Haiku achieved success rates of 80.26% and 94.74%, respectively. Mistral Nemo, Command R+, Grok Beta, and Gemini 1.5 Flash achieved success rates of 53.95%, 55.26%, 69.74%, and 69.74%. The worst-performing model was Llama 3.2, with a success rate of 30.26%.

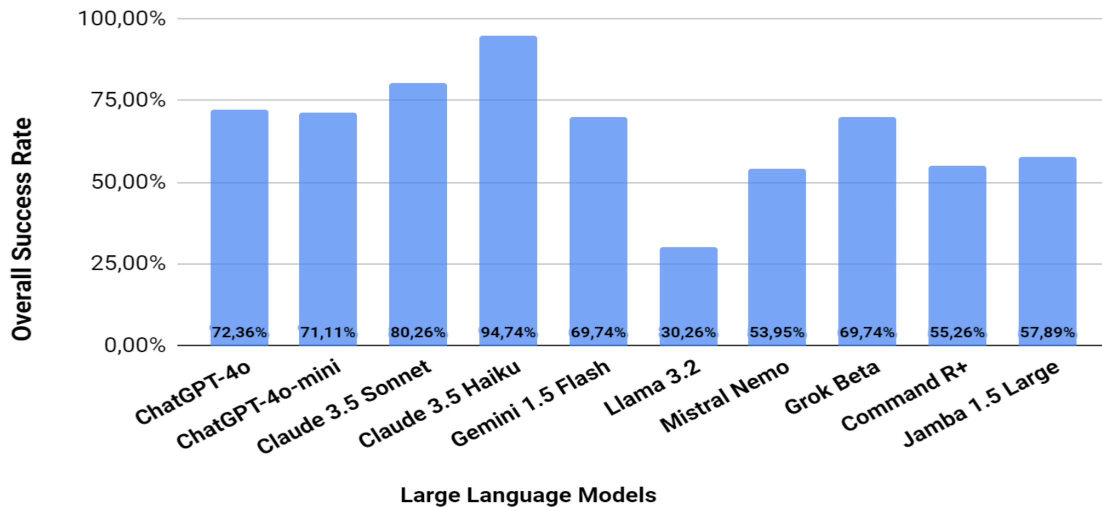

Figure 6: LLM's Overall Success Rates

The overall success rate is composed of three different difficulty levels: easy, medium, and hard. In total, there are 76 problems distributed across three difficulty levels. The majority are in the easy range, with 50 problems, followed by 20 problems in medium, and finally 6 problems in the hard difficulty.

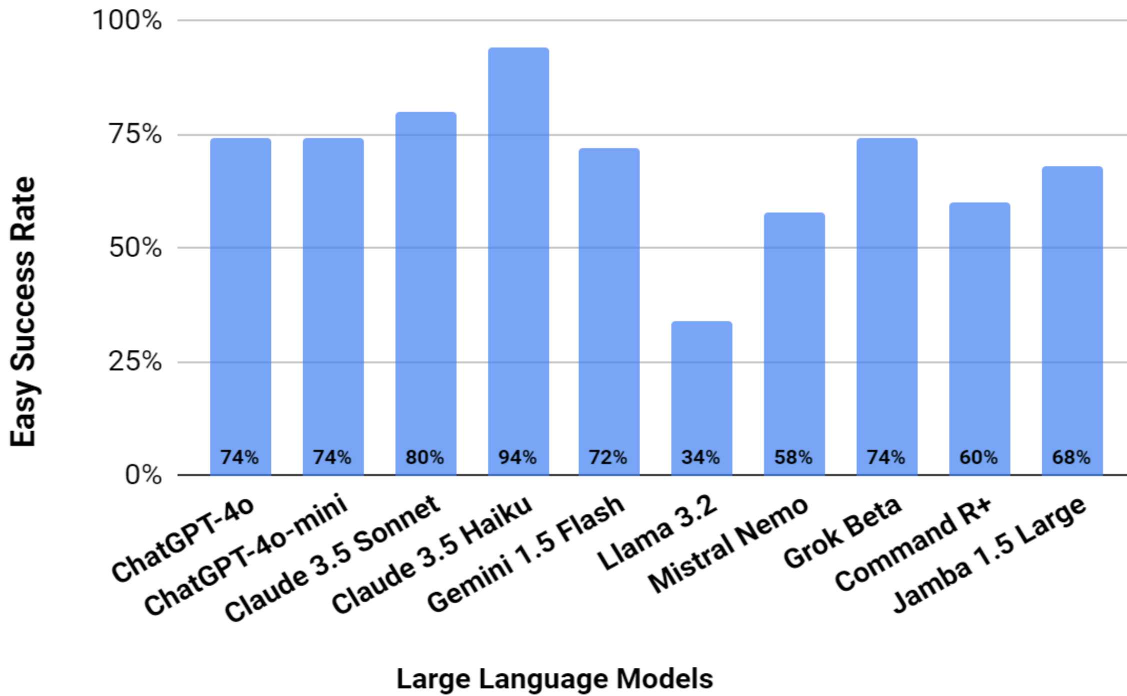

Figure 7: LLM's Easy Success Rates

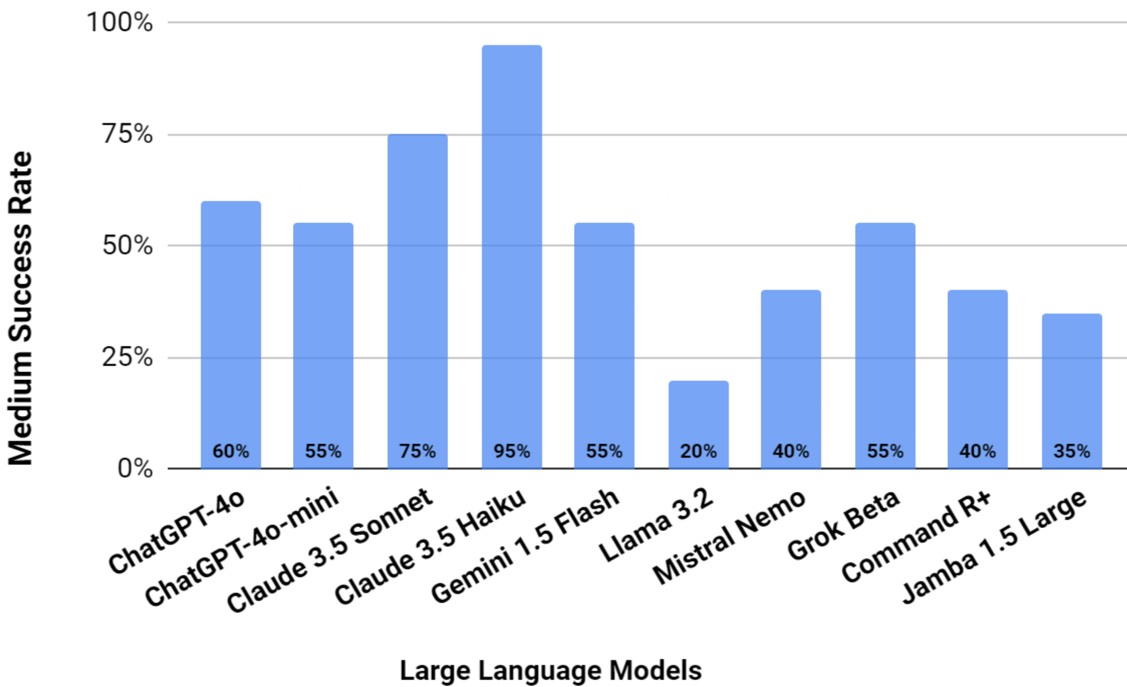

Figure 8: LLM's Medium Success Rates

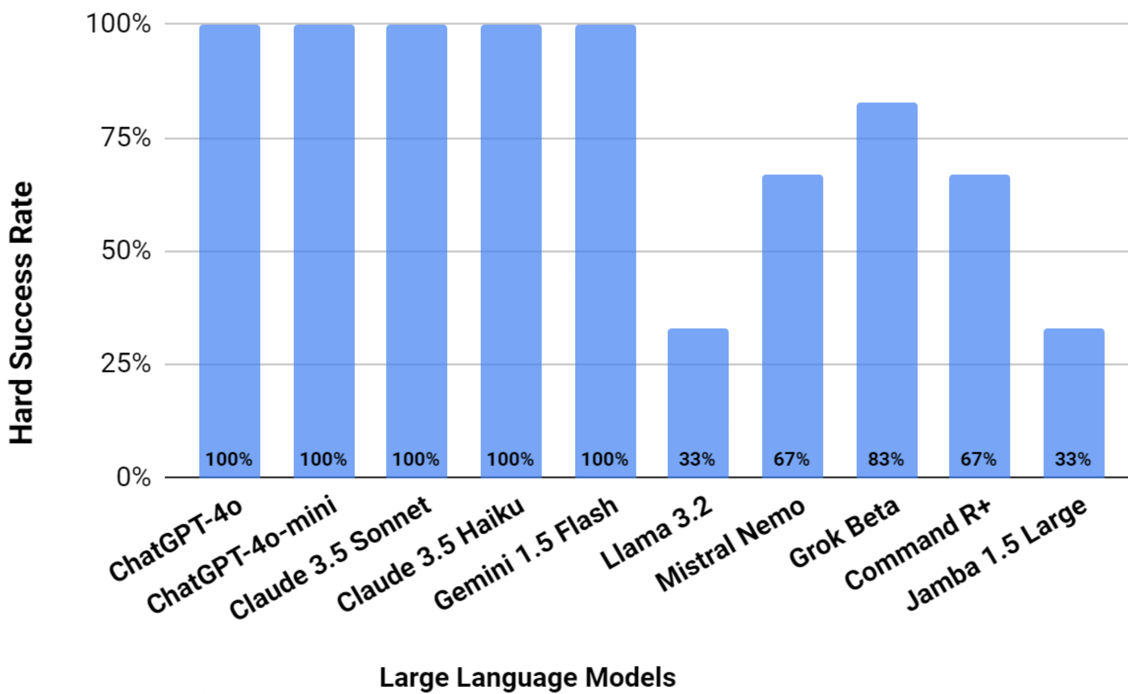

Figure 9: LLM's Hard Success Rates

The success rates illustrated in *Figure 7*, *Figure 8*, and *Figure 9* provide insights into the effectiveness of the LLMs in resolving run-time errors across varying difficulty levels of programming problems. *Figure 7* displays the success rate for the easy programming problems and how each LLM performed on these solutions, with the mean appearing to be around ~70%. Llama 3.2 had the worst performance, while the best success rate in resolving easy problems was achieved by Claude 3.5 Haiku. *Figure 8* illustrates the success rates achieved by the LLMs on medium programming problems. Initially, it appears similar to *Figure 7*; however, a slight downward shift occurs directly after Claude 3.5 Haiku. When moving to *Figure 9*, the disparity in performance becomes pronounced. Only a few models sustained 100% success rate, while others saw their success rates drop significantly. However this 100% success rate on hard problems does not mean these models can solve all hard problems easily. The bias can be attributed to the small sample size of hard problems in the dataset. Some easy or medium difficulty problems could be classified as hard, while some of the hard problems can be considered easy. The classification of problems into easy, medium, or hard is entirely determined by Leetcode. Perfect success rates for hard problems results from a combination of small sample size, misclassification, and alignment of these problems with the model's strengths. Considering these factors, they do not represent the model's overall ability to handle other hard problems.

In *Figure 10* the distribution of corrected solutions by the LLMs and how many were correct in Java or C++ is presented. Adding the Java and C++ solutions together provides the total number of correct solutions the LLM achieved during the experiments. *Figure 11* presents a diagram of the correct solutions each LLM achieved according to the difficulty level. The correct solutions referred to in *Figure 10* and *Figure 11* are the solutions successfully healed by the LLMs.

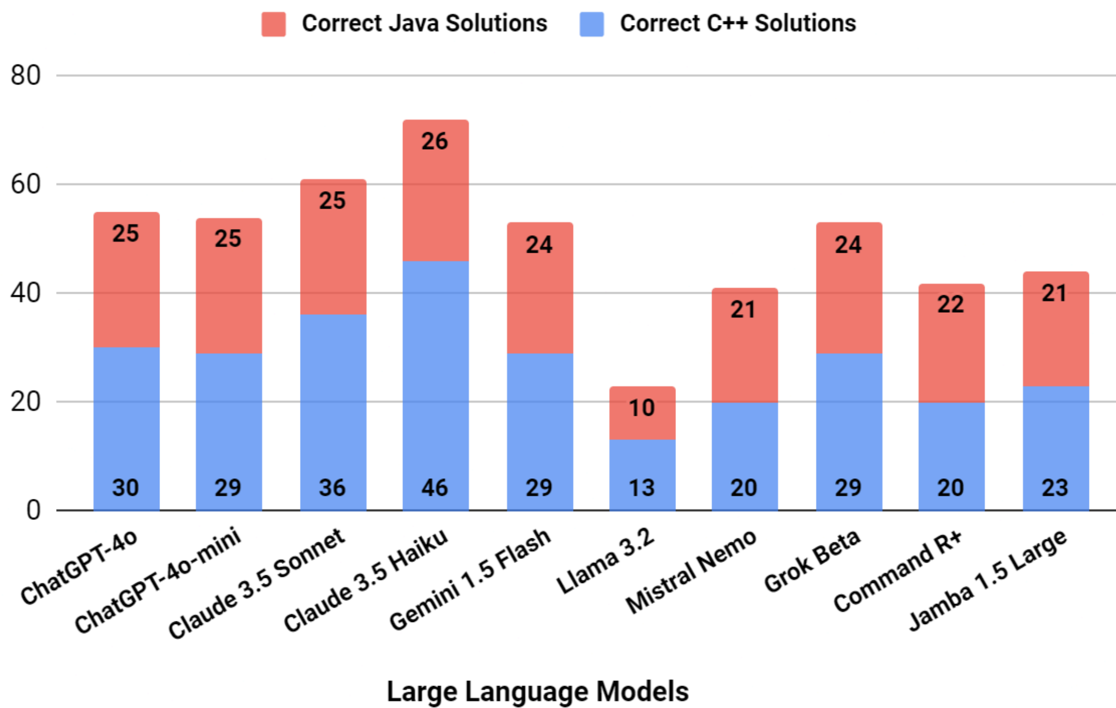

Figure 10: Performance Overview of Different Programming Languages

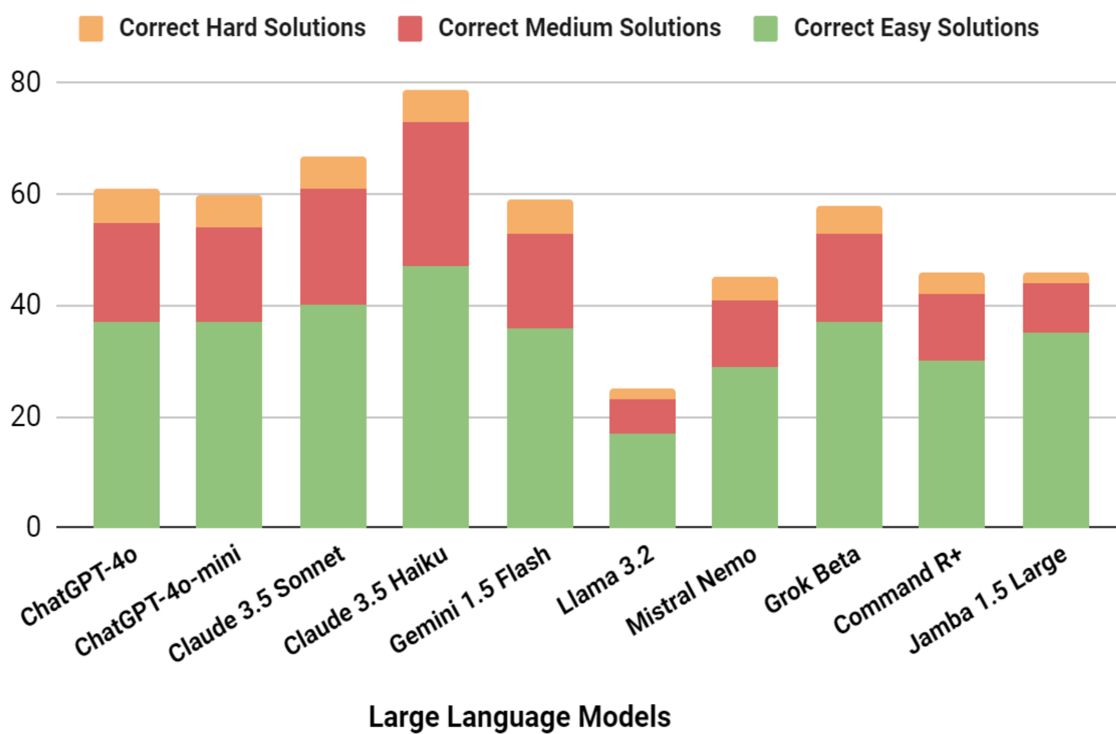

Figure 11: Performance Overview of Different Difficulty Levels

What is the overall effectiveness of the large language models in achieving self-healing code? (RQ2)

In answering **RQ2**, the original equation used to calculate the individual success rates for each LLM must be revised, as it only represents the calculation for a single LLM. However, **RQ2** requires considering all the LLMs collectively to determine an overall success rate, as if they were combined into a single unified model. The overall success rate was calculated using this formula:

$$Average\ Success\ Rate\ (\%) = \frac{\sum Success\ Rate_{LLM}}{Total\ Number\ of\ LLMs}$$

This average success rate formula results in a single percentage value that serves as a collective measure of the effectiveness of all LLMs as a unified model. An aggregated success rate provides insights into the overall reliability of LLMs and their suitability for autonomic computing applications, specifically to self-healing functions.

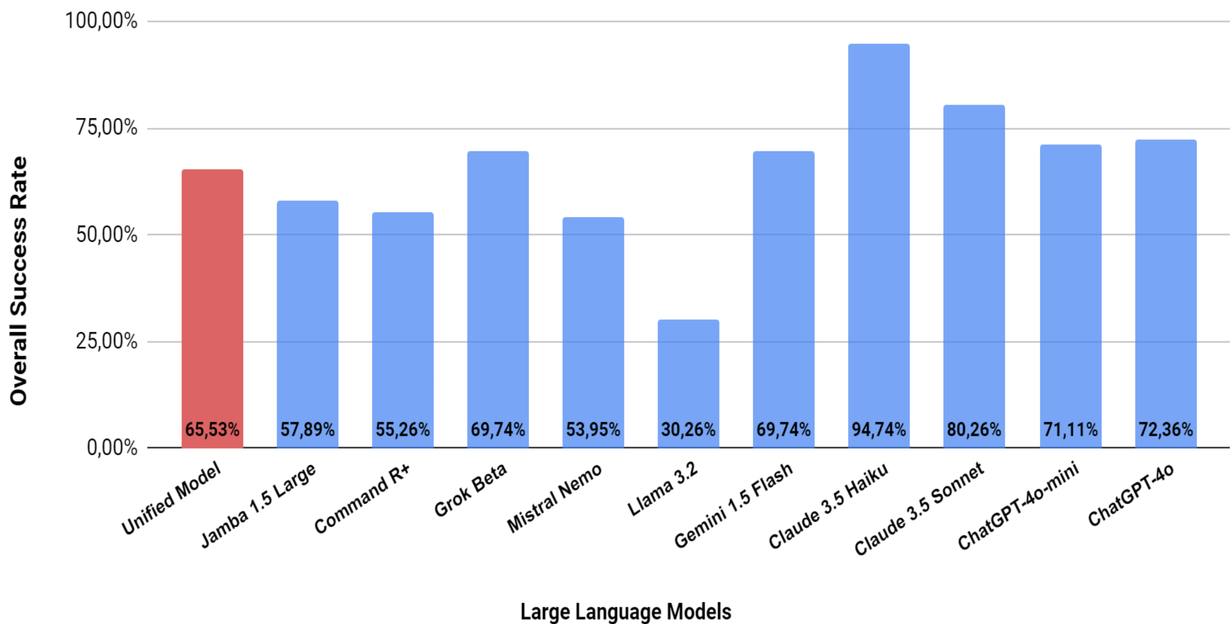

Figure 12: Unified Model Overall Success Rate

Figure 12 compares the unified model to other models in the research. The aggregated success rate for the unified model was 65.53%, indicating that, on average, the models were moderately effective. However, the success rate provides a holistic view of LLM performance, there were noticeable discrepancies between individual models. For instance Claude 3.5 Haiku consistently ranked higher in success rates across all difficulty levels, while Llama 3.2 showed significantly lower success rates. In scenarios where the performance of a single model varies significantly, combining multiple models may provide a more reliable solution. As shown in the bar chart, the high performance of Claude 3.5 Haiku appears to offset the lower performance of Llama 3.2 in the unified model's calculations. However, the unified model may not fully capture high-level accuracy of top-performing models, potentially limiting its applicability for complex logical problems.

Discussion

The results of this study demonstrate that, with the exception of Llama 3.2, each individual LLM and the unified model have the potential to autonomously address run-time errors in software. This aligns with the self-healing principles of autonomic computing. As can be seen with the success rates, which varied across the ten models, with Claude 3.5 Haiku achieving the highest overall success rate 94.74% and Llama 3.2 the lowest 30.26%. This disparity underscores the variability in model architecture and training quality among the different LLMs. Additionally, the unified model achieved a success rate of 65.53%, which highlights its overall effectiveness, suggesting potentially that no single model is universally optimal, a collective approach utilizing many LLMs could yield more robust results. This can be further elaborated upon, as some models achieved results below expectations. Relying on such a model in certain instances could result in longer times for healing that erroneous code, as it might require multiple attempts to be resolved. In critical scenarios, such as aircraft systems, ensuring a 100% success rate is paramount. Therefore, solutions that utilize multiple models to provide code responses and test them simultaneously should be prioritized. This potentially can generate multiple fixes at the same time as well as offering different solutions to promptly handle the key issue at hand. However, the integration of many LLMs into a single software is beyond the scope of this research. The unified model is envisioned as multiple models working symbiotically with the system to resolve errors and effectively heal the system.

The main objective of this thesis was to investigate whether LLMs have the potential to achieve self-healing code. To investigate this topic several programming problems were sourced from Leetcode and then stored in two different JSON files: one for Java and one for C++. Each problem was categorized by difficulty level: easy, medium, or hard, determined by Leetcode. The total distribution amongst the three difficulty levels was 50 easy problems, 20 medium problems, and 6 hard problems. However, the distribution was unequal, with 48 problems in C++ and 28 problems in Java.

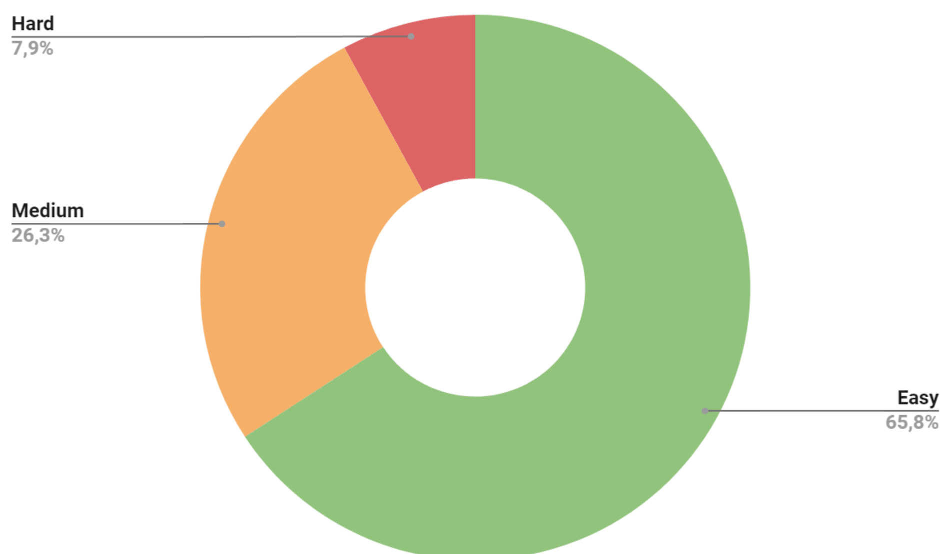

Figure 13: Distribution of Difficulty Levels in Programming Problems

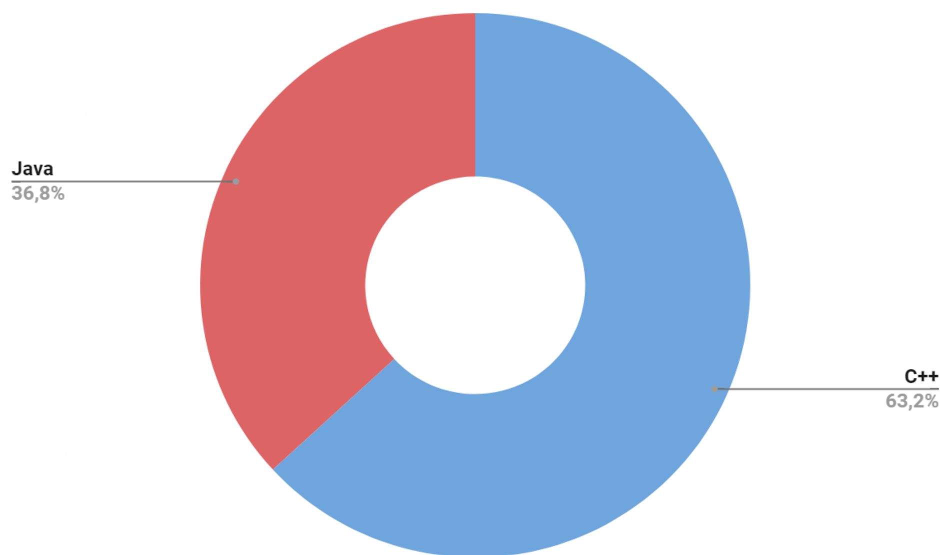

Figure 14: Distribution of Programming Languages in Programming Problems

Success rates were highest for problems categorized as easy for most LLMs with some discrepancies, then performance deteriorated for medium level problems. However, hard problems had fewer solutions for the models to correct, and the problems provided to the models were solved without issue, at least by the first five models shown in *Figure 9*. This is a direct consequence of the smaller sample size of hard problems. If a larger sample size of hard problems was used, the percentages would likely normalize around ~50-60%. However, with the current dataset, hard problems were solved most effectively by the models. That said, this does introduce a potential bias to the research. Although it would be difficult to claim that the LLMs struggle with reasoning capabilities, aside from a few discrepancies, many models performed consistently well across all difficulty levels. *Figure 16* presents an overview of the success rates across programming languages, showing that more models performed better in Java than C++. Many factors could have influenced this, such as training data bias, where some models might be better equipped to handle Java code compared to C++ code. Another factor is the dataset used in the research may have included more Java problems that were easier to solve compared to C++ problems. This imbalance could have influenced the success rates. One key issue during the results, was the inclusion of extraneous text in the code responses from many models for both C++ and Java. This occurred more frequently for C++ problems in the dataset. The additional information key-value field was occasionally misinterpreted by the models as a part of the erroneous solution, leading to incorrect outputs. Although not all LLMs made this mistake, the majority did, which consistently resulted in failed solutions. However, at times, textual information was included in the response, even though the prompt explicitly stated that only code should be provided. In such cases, due to the sheer amount of those responses, I chose to test the code rather than instantly consider those responses as failures. Despite the fact that un-commented text in actual software systems would cause compile-time errors. Allowing these cases to proceed may have skewed the results, potentially lowering the success rates for both C++ and Java solutions. An exception was made to give the LLMs a fair chance to solve the erroneous solutions, but this consideration was not extended to redefinitions of the additional information field.

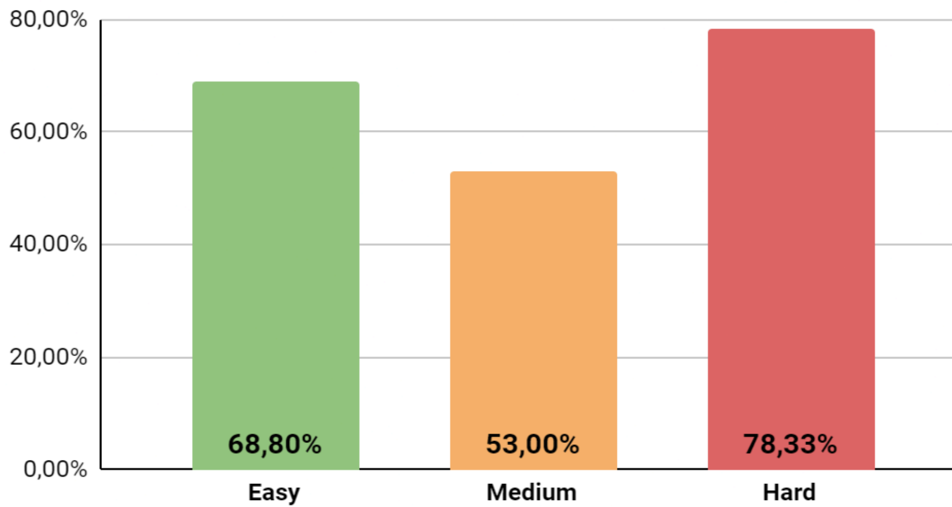

Figure 15: Success Rates across Difficulty Levels

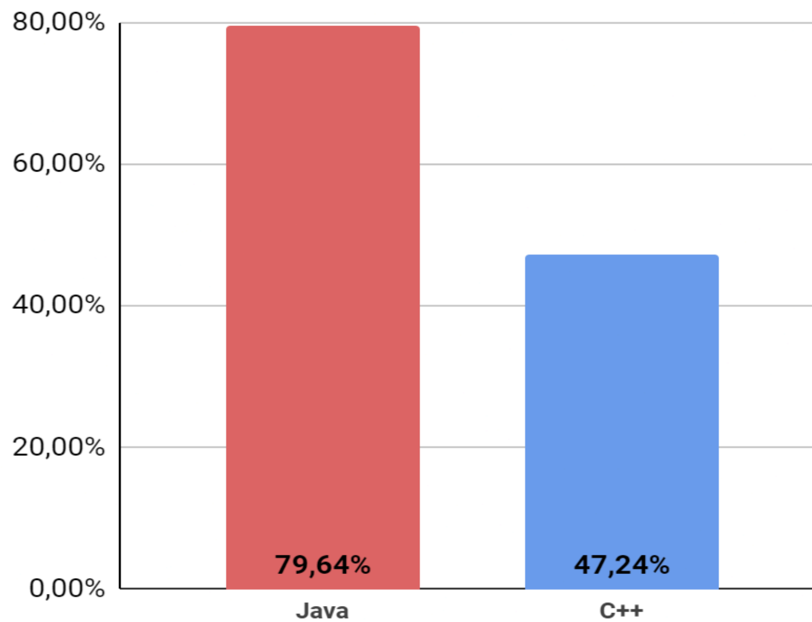

Figure 16: Success Rates across Programming Languages

To evaluate whether LLMs can achieve self-healing by autonomously resolving run-time errors in code. My findings affirm that LLMs exhibit varying degrees of success in correcting such errors, thereby demonstrating the potential for self-healing code, by establishing a success rate metric as the framework for evaluating and comparing the effectiveness of each LLM. Data from Google Sheets, which documented the outcomes for each solution, is used to calculate the success rates. When analyzing *Figure 6* it is evident that the top-performing model is Claude 3.5 Haiku and the worst-performing model is Llama 3.2. Even though the other models hover around 50-80%. The success rates varied significantly across the ten evaluated models. This highlights the differences in training data, architecture, and optimization strategies among the models. For the unified model, the success rate was 65.53%, as shown in *Figure 12*, reflecting moderate effectiveness in correcting run-time errors across a varied dataset. However, models like Claude 3.5 Haiku excelled

across all difficulty levels, demonstrating a strong ability to generalize and handle complex problems. In contrast, Llama 3.2 struggled even with basic problems. The unified model approach employs the strengths of models, such as Claude 3.5 Haiku, whose consistent success across all difficulty levels elevates the overall effectiveness. In addition, the unified model provides a holistic measure of LLMs' potential for self-healing code when working collaboratively to address eventual challenges. The inconsistencies in weaker models are masked by the average success rate for the unified model, raising concerns about their ability to generalize to other datasets or domains. The results based on programming languages C++ and Java, may not directly apply to other languages, like Python, or fields, such as cybersecurity or healthcare. Considering the success rates observed here, it suggests that LLMs are well-adapted to multiple programming languages. However, adding Python or Javascript could further demonstrate this concept and its generalizability. Nonetheless, it is important to acknowledge the growing utility of LLMs in software engineering and autonomic computing. Although still in its infancy, their potential is tremendous, offering the possibility to reduce the time programmers spend on fixing code. Instead, LLMs could be seamlessly integrated into software systems to alleviate such concerns. However, the costs and feasibility of such integration remain uncertain until further testing in this field is conducted.

In this study, a manual testing process was employed to ensure thoroughness. This approach was effort-intensive, requiring each solution to be tested individually using Leetcode's built-in submission environment. Manual testing also added the possibility of human error into the equation, specifically during documentation of the outcomes. An automated testing pipeline would be an ideal approach to streamline the evaluation process, as it eliminates human error and simplifies the workflow. However, this is not feasible as long as Leetcode's test cases remain undisclosed. Creating test cases to replicate Leetcode's thorough evaluation would require tremendous time and effort to achieve comparable results, making both approaches time-consuming, albeit to varying degrees. This study has a larger size in comparison with the prior study conducted by Bäverlind, encompassing ten different LLMs and 76 coding problems across two programming languages. My research relied on Leetcode for gathering and testing solutions. The process of gathering solutions was strenuous and mentally exhausting, since each solution had to manually be reviewed. Some programming problems contained thousands of solutions. If this study were to be repeated, I would personally recommend using a pre-existing dataset or gathering solutions by posting in Leetcode's forums and encouraging users to submit their run-time erroneous code.

The findings of this research have significant implications for software engineering and related fields. Through automation of the detection and healing of run-time errors, LLMs have the potential to reduce the time and cost traditionally associated with software maintenance. This would enable developers to focus on more innovative or complex tasks. Integrating LLMs into software systems with the intent to self-heal can also extend into areas such as cybersecurity, where they could autonomously detect and patch vulnerabilities. The reliance on LLMs could become problematic as it impacts the job market through removing entry-level roles as automation increases. Nevertheless, technological innovations do not always render industries obsolete, instead they transform. For example, the introduction of tractors revolutionized agriculture, ensured food security, reduced manual labor, and created new opportunities. Similarly, the integration of LLMs is likely to drive innovation, creating these new opportunities for developers. This study demonstrates the potential

of LLMs in achieving self-healing functions in software. These findings underline the limitations and potential of LLMs, they also show the transformative abilities of LLMs.

Societal and Ethical Considerations

In this thesis, ethical considerations were minimal, since the focus of this study lies on working with publicly available datasets and does not involve any participants, sensitive personal data, or proprietary information. Even then the code gathered is inspected for any personal information to remove, in essence this became a process of removing user comments from the solutions. Essentially, none ethical concerns are raised in this thesis. Further on, societal concerns lie in the integration of LLMs in software systems and how this might negatively affect the available software engineering jobs for up and coming developers. As LLMs improve in programming tasks, this could potentially reduce the demand for programmers, especially entry-level positions.

Threats to Validity

Four types of validity threats must be examined to ensure the applicability of the findings. Internal validity concerns factors that unintentionally affect the outcomes studied without the researcher's knowledge [32]. External validity refers to the ability to generalize the results and to which extent [32]. Construct validity examines the relationship between the concepts and theories behind the experiments, aspects such as what is measured and affected [32]. Lastly, the conclusion validity evaluates the accuracy of drawing correct conclusions about the relationship between treatments and outcomes of an experiment [32].

Internal validity can correlate the observed outcomes to the models' performance rather than external or uncontrolled factors. The manual approach of testing and data collection, introduces the potential of human error. These errors can occur during documentation of the test results or in the categorization of failures, such as incorrect labeling of compile-time errors as run-time errors. Additionally, the limited size of hard problems creates a bias in success rates, as even a few successful results can disproportionately inflate success rates. Continuing with external validity threats, the dataset in this study is exclusively from Leetcode and focuses on two programming languages, C++ and Java. These are widely used, although the findings may not generalize to other languages such as Python or Javascript, or to real-world applications beyond simple programming problems. Moreover, the LLMs represent only a subset of existing models, and their performance might not be indicative of other LLMs or future versions. The reliance on Leetcode's difficulty level classifications is another issue in this study. These classifications may not align with actual difficulty levels of some programming problems.

This in fact does affect the evaluation of model performance. A notable example of this occurring is in *Figure 9* where some models performed with a perfect success rate. This was not repeated in the lower difficulty classifications. The interpretation that can be made from this is that some programming problems in medium or easy might be considered hard, while certain hard problems belong to either easy or medium. Conclusion validity assesses the conclusions drawn from the results, a key limitation in this regard is the small dataset size relative to the potential scope of programming problems. The 76 programming problems provide a meaningful sample, a larger dataset would improve the generalizability of the results.

Conclusions and Future Work

This thesis's objective was determining whether LLMs can achieve self-healing code. Hence to investigate this matter, a dataset composed of Leetcode problems spanning various models, difficulty levels and two programming languages was used. Each solution was manually tested through Leetcode's submission environment. The purpose of the manual testing approach was to observe and analyze the LLMs' ability to correct run-time errors. In the absence of an autonomous system, this approach served as a simulation of how LLMs might perform if integrated into software for self-healing purposes. The findings indicate that some models demonstrate consistent success rates in healing erroneous code across all difficulty levels. Although, the opposite is also true where some models exhibit an inability to effectively correct erroneous code. Claude 3.5 Haiku emerged as the top-performing model, its potential could make self-healing code a reality. Conversely, models like Llama 3.2 require more attention and improvements to reach a comparable level of performance and reliability. Moreover, the overall success rate, calculated through the average equation established in the results, does display the feasibility of using LLMs for self-healing code. In conclusion, the results point out the potential of LLMs in achieving self-healing code. Their current abilities to fully realize self-healing functions in code is uncertain, as the integration of LLMs is an unknown factor. However, their promise in this area is undeniable. In the coming years, with more improvements in reasoning capabilities, problem-solving skills, and decision-making, it will become reality.

Future research in this area should expand the dataset and also lessen the reliance on Leetcode, such as sourcing data from alternative platforms. Leetcode lacks a public API in accessing its programming problems and solutions. As a result, each solution had to be manually reviewed. This process is strenuous, as some programming problems have thousands of solutions, each of which must be reviewed. Developing an application to automatically gather solutions would be a more efficient approach. However, obstacles such as bypassing bot-detection systems and simultaneously scraping the platform must be overcome. If the research were to be redone today, I would ask Leetcode users to share their run-time erroneous code. Before even considering sifting through each solution within the programming problems. Additionally, datasets such as Project_CodeNet[7] contains over 13 million submissions half of which are correct. These submissions are distributed amongst eight programming languages: C++, Java, Python, C, Ruby, C#, Rust, and Go. This dataset is from 2021 and meant for training AI. Using this dataset, however, would be more efficient paired with creating your own test cases and integrating the specific LLM(s) into that automated application to test the solutions. Since individually testing each erroneous solution with this volume is infeasible. Another consideration is to increase the amount of attempts for each model, ensuring the clarity of the prompts provided. Moreover, assisting the lower-performing models in achieving higher success rates. Future work has many directions such as increasing or decreasing the amount of LLMs, programming languages, and solutions. For reasons, such as improving the generalizability and understanding of LLM(s) potential in self-healing systems.

[7] https://github.com/IBM/Project_CodeNet/tree/main

References

[1] M. Roser. *How has AI developed over the years and what's next?* World Economic Forum (Dec 12. 2022). [Online]. Available: https://www.weforum.org/agenda/2020/09/ai-is-here-this-is-how-it-can-benefit-everyone/. (Accessed: 2024-08-31)

[2] *What is AI?*, IBM (n.d) [Online]. Available: https://www.ibm.com/topics/artificial-intelligence (Accessed: 2024-09-01)

[3] D. West and J. Allen. *How artificial intelligence is transforming the world*. Brookings, (Apr 24. 2018). [Online]. Available: https://www.brookings.edu/articles/how-artificial-intelligence-is-transforming-the-world/ (Accessed: 2024-09-01)

[4] Q. An, S. Rahman, J. Zhou and J. Kang. *A Comprehensive Review on Machine Learning in Healthcare Industry: Classification, Restrictions, Opportunities and Challenges*. Sensors (Basel), vol.23, no.9. (Apr 22. 2023), doi: 10.3390/s23094178

[5] A. Wilson. *Branches of AI: A simple guide to 28 fields of Artificial Intelligence*. ApproachableAI (May 14. 2024). [Online]. Available: https://approachableai.com/branches-of-ai/ (Accessed: 2024-09-02)

[6] *What are large language models (LLMs)?*, IBM (n.d) [Online]. Available: https://www.ibm.com/topics/large-language-models (Accessed: 2024-09-01)

[7] A. Stöffelbauer. *How large language models work*. Medium (Oct 24. 2023). [Online]. Available: https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f (Accessed: 2024-09-02)

[8] *What is NLP (Natural Language Processing)?*, IBM (n.d). [Online]. Available: https://www.ibm.com/topics/natural-language-processing (Accessed: 2024-09-03)

[9] A. Banafa. *What is Autonomic Computing?* OpenMind BBVA. (July 14. 2016). [Online] Available: https://www.bbvaopenmind.com/en/technology/digital-world/what-is-autonomic-computing/ (Accessed: 2024-09-03)

[10] G. Sisinna. *How Artificial Intelligence is Powering the Next Wave of Autonomic Computing*. Linkedin (Oct. 5 2022). [Online]. Available: https://www.linkedin.com/pulse/how-artificial-intelligence-powering-next-wave-giovanni-sisinna (Accessed: 2024-09-04)

[11] *What is Natural Language Understanding & How does it work?* Simplilearn (Aug 11. 2023). [Online]. Available: https://www.simplilearn.com/natural-language-understanding-article (Accessed: 2024-09-04)

[12] *What is Sentiment Analysis?* IBM (n.d). [Online]. Available: https://www.ibm.com/topics/sentiment-analysis (Accessed: 2024-09-05)

[13] *What is named entity recognition?* IBM (n.d). [Online]. Available: https://www.ibm.com/topics/named-entity-recognition (Accessed: 2024-09-05)

[14] A. Padmanabhan. *Semantic role labeling*. Devopedia (Jan 10. 2020). [Online]. Available: https://devopedia.org/semantic-role-labelling (Accessed: 2024-09-05)

[15] *Autonomic Computing*. Engati (n.d). [Online]. Available: https://www.engati.com/glossary/autonomic-computing (Accessed: 2024-09-05)

[16] R. Gruetzemacher. *The Power of Natural Language Processing*. Harvard Business (Apr. 19 2022). [Online]. Available: https://hbr.org/2022/04/the-power-of-natural-language-processing (Accessed: 2024-09-05)

[17] *Natural Language Processing*. DeepLearningAI (Jan 11. 2023). [Online]. Available: https://www.deeplearning.ai/resources/natural-language-processing/ (Accessed: 2024-09-06)

[18] R. Merritt. *What Are Foundation Models?*, Nvidia (Mar 13. 2023). [Online]. Available: https://blogs.nvidia.com/blog/what-are-foundation-models/ (Accessed: 2024-09-07)

[19] E. Alpaydin. *Introduction to Machine Learning*. 2nd ed. Massachusetts, USA, MIT Press, 2010.

[20] Introducing OpenAI o1-preview. OpenAI (Sep 13. 2024). [Online]. Available: https://openai.com/index/introducing-openai-o1-preview/ (Accessed: 2024-09-15)

[21] S. Ruder. *Transfer Learning - Machine Learning's Next Frontier*. Ruder (Mar 21. 2017). [Online]. Available: https://www.ruder.io/transfer-learning/ (Accessed: 2024-09-16)

[22] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. 2nd ed. Massachusetts, USA, MIT Press, 2014

[23] J. Devlin, M. Chang, K. Lee and K. Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arxiv preprint arXiv:1810.04805, (Oct 11. 2018).

[24] N. Tihanyi, R. Jain, Y. Charalambous, M. Ferrag, Y. Sun and L. Cordeiro. *A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification*. arXiv preprint arXiv:2305.14752, 2023

[25] H. Ekedahl and V. Helander, *Can artificial intelligence replace humans in programming?* DIVA, (Jun 26. 2023).

[26] Z. Zhang, F. Yang, X. Qin, J. Zhang, Q. Lin, G. Cheng, D. Zhang, S. Rajmohan and Q. Zhang, *The Vision of Autonomic Computing: Can LLMs Make It a Reality?* arXiv preprint arXiv:2407.14402, 2024

[27] M. Bäverlind, *Using large language models to enable self-healing code*. DIVA, (Jun 11. 2024)

[28] S. Em. *Exploring Experimental Research: Methodologies, Designs, and Applications Across Disciplines. SSRN Electronic Journal (*Mar 2024*)*. 10.2139/ssrn.4801767.

[29] P. Bhandari. *What is Quantitative Research? | Definition, Uses & Methods*. [Online]. (Jun 12. 2020) Available: https://www.scribbr.com/methodology/quantitative-research/ (Accessed: 2024-10-18)

[30] *What is prompt engineering?*, IBM (n.d) [Online]. Available: https://www.ibm.com/topics/prompt-engineering (Accessed: 2024-11-02)

[31] Z. Yin, H. Wang, K. Horio, D. Kawahara and S. Sekine. *Should We Respect LLMs? A Cross-Lingual Study on the Influence of Prompt Politeness on LLM Performance*. arXiv preprint arXiv:2402.14531, 2024.

[32] C. Wohlin, M. Höst and K. Henningsson. *Web Engineering*. Berlin, Germany. Springer, 2006.

Appendix

Table 1. Overview of LLMs

Model Name	Developer	Success Rate (%)
ChatGPT-4o	OpenAI	72.36%
ChatGPT-4o-mini	OpenAI	71.11%
Claude 3.5 Haiku	Anthropic	94.74%
Claude 3.5 Sonnet	Anthropic	80.26%
Gemini 1.5 Flash	Google	69.74%
Llama 3.2	Meta	30.26%
Mistral Nemo	Mistral AI	53.95%
Grok Beta	xAI	69.74%
Command R+	Cohere	55.26%
Jamba 1.5 Large	AI21Labs	57.89%