IMPACT OF REVOLUTIONARY ARTIFICIAL INTELLIGENCE BASED CODE AUTOMATION MODELS ON SOFTWARE DEVELOPMENT INDUSTRY AND SOFTWARE DEVELOPERS

by

Varsha Jain

MS (Software Systems), BE (Computer Science)

DISSERTATION

Presented to the Swiss School of Business and Management Geneva

In Partial Fulfillment

Of the Requirements

For the Degree

DOCTOR OF BUSINESS ADMINISTRATION

SWISS SCHOOL OF BUSINESS AND MANAGEMENT GENEVA

May, 2023

IMPACT OF REVOLUTIONARY ARTIFICIAL INTELLIGENCE BASED CODE AUTOMATION MODELS ON SOFTWARE DEVELOPMENT INDUSTRY AND SOFTWARE DEVELOPERS

by

Varsha Jain

APPROVED BY

____
Prof. Sasa Petar,Phd, Chair

____
Milica Popovic Stijacic, Phd, Committee Member

____
Kishore Kunal, Phd,DBA, Committee Member

RECEIVED/APPROVED BY:

____
SSBM Representative

Dedication

This thesis is dedicated to my family for their constant support and encouragement. My parents, who have been the greatest source of inspiration. My, late mother, with her undying fighting spirit, taught me the importance of resilience and determination. She has always been the biggest pillar of my strength. My father, who has always stood by me. He always supported and encouraged me in everything that I wanted to do. This thesis is dedicated to both of them.

My husband whose unwavering support and belief in me has been my driving force. A big thanks to him for always being there. He gave me the strength and motivation to overcome challenges and pursue my dreams.

To my children, who have been my constant source of inspiration. Am deeply indebted to them for their patience, understanding and most importantly for giving me the time and space to pursue this journey. Their support in letting me spend long hours immersed in my research and writing, undisturbed, greatly helped me complete this journey in a short time.

Acknowledgements

I extend my sincere thanks to my mentor Dr. Kishore Kunal for his continuous guidance and help all through the completion of the thesis. His expertise, guidance and patience has been instrumental in shaping this work. The review comments and insightful feedback from Dr. Kunal pushed me to strive for excellence. I am very thankful for his mentorship and guidance throughout this journey.

I would also like to express my gratitude to the research participants for sharing their time and knowledge for the purpose of this study. Their willingness to participate and provide valuable insights has enriched the findings and contributed valuably towards the completion of this thesis.

My family members kept my morale high throughout the process. Without their support and motivation to pursue the Doctorate program, it would have been impossible to complete the thesis and Doctorate in this manner.

ABSTRACT

IMPACT OF REVOLUTIONARY ARTIFICIAL INTELLIGENCE BASED CODE AUTOMATION MODELS ON SOFTWARE DEVELOPMENT INDUSTRY AND SOFTWARE DEVELOPERS

Varsha Jain
2023

Dissertation Chair: Dr. Kishore Kunal

The purpose of this study is to explore the impact of large language models on the software industry. The study seeks to answer the research question, “How does automating software development impact the job of software professionals and how much effort and cost-saving will organizations get by augmenting software development with artificial intelligence”. The goal is to analyze the various software development tasks that can be automated using artificial intelligence-based code automation models and study the impact of this automation on software professionals and the industry.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I:

INTRODUCTION

1.1 Introduction

Artificial intelligence has positively impacted many industries. In healthcare, artificial intelligence can assist in clinical data processing and bringing value from unstructured data. Similarly, in the banking industry, artificial intelligence can help with fraud detection and loan default. The software industry also has constantly been working on improvements in the software development process. Software development encompasses everything from programming and unit testing to debugging and documentation. The manual process of software development is both error-prone and expensive. It relies heavily on the availability of skilled software professionals. The recent innovations in the field of Artificial Intelligence (AI) and especially large language models show promise in helping tackle these challenges.

Software systems have become exponentially complex since the early days of programming (Bahdanau, Cho and Bengio, 2014). The Standish Group reported that for the period 2011 to 2015, only 44% of projects were on a budget. The percentage of projects that were completed on time was only 40% while those which met the project objectives were 56% (CHAOS REPORT 2015, no date). According to the comprehensive CHAOS 2020 report published in 2021 and referenced in Henny Portman's QRC (Project Success, Quick reference card, 2021), the percentage of successful software projects was only 31% while the percentage of projects which failed was 19%. 50% of the software projects are still facing challenges to achieve success. In addition to this, there is a chronic shortage of skilled software developers. Reports (Krasner, 2018) suggest, that it was estimated that the United States incurred approximately $319 billion in costs due to substandard quality software in the year 2018.

Recently, the field of Artificial Intelligence has seen some innovations and in particular with large language models. This study analyzes these different AI language models and different programming tools based on AI. It questions how these models and tools can contribute to speeding the software development process. The effort and cost-saving the industry can get by employing these models can help accelerate their adoption in the industry. The goal is to understand how artificial intelligence will affect the job of software professionals and its impact on the growth of the software industry.

1.2 Research Problem

The software development industry faces significant challenges, such as a shortage of skilled software developers, cybersecurity breaches, and complex technologies, as revealed by the 2022 Global DevSecOps Survey (Bendor-Samuel, 2022). The COVID-19 pandemic has further compounded these issues, with remote and hybrid work scenarios becoming more prevalent, reducing the supply of skilled staff. According to Code.org (Code.org 2016 Annual Report, no date), there are over 500,000 unfilled computing jobs in the United States alone, as the number of computer science graduates does not keep pace with the growing demand for software developers. This talent gap is causing several issues for businesses, such as increased development timelines, rising project costs, and reduced ability to innovate efficiently. A survey by Harvey Nash (Harvey Nash Group, 2021) found that 67% of technology leaders believe that the talent gap is hurting their businesses.

Furthermore, the software development industry faces a high project failure rate, resulting in a considerable drain on resources, revenue loss, and reputational damage (The curious case of the CHAOS Report 2009, no date). The rapid pace of technological change, increased competition for talent, and talent shortage make it challenging for the industry to keep up with constantly evolving technologies and accelerate digital transformation.

1.3 Purpose of Research

In the last few years, there have been many developments in the field of artificial intelligence. A range of large pre-trained models has been released that show promise in automating code-related tasks. With the rise in development costs and the growing dearth of talent, the purpose of this research is to assess how these large pre-trained models can be leveraged. The aim is to assess the impact of these models in different phases of software development. The research aims to study how traditional techniques compare to performing code automation using large language models. The study will also analyze the cost and effort benefits organizations will get by leveraging large pre-trained models for code automation.

The study also aims to analyze the impact this automation will have on the jobs of software professionals. Automation in the field of software development is not new and has been happening for several decades now. Integrated Development Environments (IDEs) that support code writing, and rule-based systems capable of translating programming languages are all forms of automation. However, large pre-trained models have brought a revolutionary change. The purpose of the research is to analyze the new skills that will be needed in the future if large language models could augment code-related tasks.

1.4 Significance of the study

Artificial intelligence and machine learning is the fastest-evolving technology. With a plethora of large pre-trained models released in the last two years, from GPT-3 (OpenAI, 2022), GPT-J (EleutherAI - text generation testing UI, no date) to Palm-Coder (Chowdhery et al., 2022), CodeGeeX (Zheng et al., 2023) and GPT-4 (OpenAI, 2023), there is a lack of study on how these models impact large software enterprises. The results from this study will provide insights into various code-related tasks that can be automated. They will also help understand the efficiency of these models in performing such tasks.

When such insights are available, they can help organizations change the way they think of software development today. Based on the results of the study, organizations can make informed decisions on what stages in the software development cycle, automation using large language models can be applied, and how they will benefit from the same.

The study also aims to come up with a framework that can help perform a cost-benefit analysis when augmenting code-related tasks with large pre-trained models. This framework can help organizations assess and measure the benefits, if any, that they may or may not get based on the problem they want to solve.

Automation invariably leads to reskilling. When certain tasks are automated, it means that the machine can now perform those tasks more efficiently and effectively than humans. This often leads to job displacement, but it also creates opportunities for individuals to learn new skills that are in demand.

Reskilling is important because it allows individuals to adapt to new technologies and the dynamic employment landscape. By learning new skills, individuals can remain competitive and increase their value in the workforce. It also helps to ensure that individuals can find new job opportunities as old ones become obsolete. The results of this study will help identify the new set of skills that will be needed as different code-related tasks get automated using artificial intelligence and machine learning.

1.5 Research Purpose and Questions

The research study's overall objective is to assess the impact of large pre-trained models on the software industry as well as on software professionals.

- What software development tasks can be automated using artificial intelligence-based language models?
- How do large language models' code automation capabilities compare against traditional code generation?
- How much effort saving and cost-benefit will organizations get by augmenting these AI-based models with a software programmer?
- What types of jobs will be completely replaced, if any, due to the adoption of AI in the software industry?
- What is the new set of skills that will emerge as AI is increasingly adopted across the organization for software development?

CHAPTER II:

REVIEW OF LITERATURE

2.1 A brief history of computing

A brief history of how computing and artificial intelligence has progressed over the years. In the early 1940s and 1950s, scientists began to explore the concept of artificial intelligence. Alan Turing published his work on “Computing Machinery and Intelligence” (Turing, 2004) in 1950, where he first explored the concept of making machines intelligent. The Logic Theorist, developed by Newell and Simon is considered a stepping stone in the field of Artificial Intelligence (Kaur, no date). However, many challenges like the need for high computation power, huge data requirements, and many technological bottlenecks hindered the growth of the field of Artificial Intelligence.

As technology advanced, the realm of Artificial Intelligence also began to grow. AI saw significant growth from 1993 to 2011 (enterpriseitworld, 2018). As computing became faster with the availability of cost-effective memory and data storage, access to different types of data (structured as well as unstructured) became possible. This significantly aided the advancement of machine learning and artificial intelligence. It became possible to apply complex and compute-intensive deep learning algorithms to solve bigger problems.

2.2 Automation Approaches

AI is transforming the software development process in numerous ways - from AI-powered tools for code suggestion to automated generation of test cases, fixing and catching bugs in the code.

2.2.1 Rule-based systems

The traditional code automation approach relied heavily on programming language rules and grammar (Imam, Rousan and Aljawarneh, 2014). However, this approach requires a substantial level of expertise in the specific domain and significant manual effort. Also, when dealing with intricate systems, it can be exceedingly time consuming and challenging to generate the rules. Automation tools built with this approach cannot learn and improve themselves. AI-based large language models, on the other hand, can comprehend like a human and learn and improve based on their training data.

2.2.2 Neural Machine Translation

With the increase in computation power and advances in deep learning, neural networks like RNN or recurrent neural networks, LSTM or long-short-term neural networks, and GRU or Gated Recurrent Unit became very popular. A typical neural machine translation (NMT) model contains an encoder-decoder structure (Yang, Wang and Chu, 2020). The encoder is used to encode the input sequence while the decoder makes the predictions for the target sentence. Based on the requirement, these encoder-decoders can be either feed-forward neural networks or recurrent neural networks.

Recurrent Neural Networks (Zargar, 2021) are designed for data that is sequential such as a sequence of words, images, sentences, and so on. A recurrent neural network considers time as one of the dimensions and changes its state with time. They have proved very powerful in solving NLP tasks. However, they suffer from Vanishing gradient and exploding gradient problems (Zargar, 2021). Models like LSTM and GRU help overcome this problem.

One drawback of using the NMT approach for machine learning-based coding tasks is training a model from scratch with a large volume of labeled data. This approach has an inherent dependency on the availability of task-specific large labeled datasets – thus increasing the effort and cost of training.

2.2.3 Pre-trained language models

To overcome the bottlenecks of traditional encoder-decoder architectures, Bahdanau, in 2016, introduced the concept of attention mechanism (Bahdanau, Cho and Bengio, 2014) using which the decoder decides which part of the sentence to pay attention to. Later, the transformer architecture, which is built on attention mechanism was proposed by Vaswani et al., 2017. This architecture is much more powerful than traditional sequence to sequence models. Several models emerged as a result of this concept like BERT, GPT-1/2/3, T5, and so on. In a study, Galanis et al., 2021 described how these transformer-based models have evolved.

2.3 AI in Software Development

Artificial Intelligence can help in various stages of the software development process. As discussed by Vinugayathri, no date, AI can impact the following areas in software development:

- Requirement Gathering
- Design
- Code Generation
- Testing and
- Deployment

Mithas, Kude and Whitaker, 2018, believe that AI can play two different roles in software development. In its first role, AI can be used as a tool to develop software programs, and in the other role, AI itself serves as the software. In its first role, AI can be used to generate code, thus, helping the software developer to shift focus to other business tasks and help generate value. It is assumed that AI will replace code in its second role. However, it is believed that conventional code will still be relevant.

2.4 AI-Based Tools used in Programming

A report by Forrester (Hammond, 2020) predicts how low-code platforms and breakthroughs like GPT-3 will change the way software development teams work. There is a consensus building upon the need for such AI-based coding tools today. In terms of developing code – even today, we continue to write code by hand. The programmer references multiple resources - primary resources like the programming language documentation and secondary resources like books, code repositories like GitHub, discussion forums like Quora, and many more. Over the years, we have seen many coding assistants that have become available to programmers to assist them in their journey. While many are rules or grammar-based, recently, many of the code assistants are based on Artificial Intelligence. Some examples are:

- Tab Nine – a deep learning model which is a code auto-completion tool. As the programmer types the code, it predicts or suggests the remaining code for that line. This is a GPT-2 model trained on around 2 million files from GitHub. It supports 23 programming languages and has around 14 integrations with software like IntelliJ IDEA, PyCharm, and so on (Synced, 2019).
- Kite (Hung, 2021) is another ML-based tool that is good at code completion. In addition, it also uses recommendation techniques to suggest similar or related code within its codebase. Kite is trained on over 25 million files. It supports around 16 programming languages and is integrated with 16 code editors.
- Microsoft's IntelliCode (Ramel, 2020), (Svyatkovskiy et al., 2020) is another code assistant which is based on the LSTM model architecture. It assists in statement completion. It also provides signature help and recommends the most likely overload for the given context. IntelliCode supports C#, C++, TypeScript/JavaScript, or XAML in Visual Studio 2019 or higher.

- There are many more such tools like AiXcoder, IntelliSense, Jedi, and Wing Python IDE.
- Asiroglu et al., 2019 suggest generating HTML code automatically, from individual web page mock-ups, using machine learning techniques.

Although these tools are very powerful, there are a few drawbacks:

1. Most of these tools assist in only statement completion or signature selection
2. In many cases, with these models, the accuracy attained on synthetic benchmarks fails to translate well on practical completion tasks in real-life settings.
3. Limited or specific language support.
4. Inability to interpret natural language and generate code and vice-versa.

Given this context, AI-based language models like Codex, GPT-3, CodeBert, and GPT-J are extremely powerful.

2.5 Large language Models

As discussed under “Automation Approaches – Pre-trained language models”, the last three years have seen the growth of pre-trained language models. The code writing, code understanding, and natural language understanding of these pre-trained models which are based on transformer architecture is far superior to any existing tool or technology. In the paper “Attention is all you need” (Vaswani et al., 2017), the authors explain the transformer architecture in detail. A typical transformer model contains both an encoder and decoder stack. There are some models like BERT which are encoder-only while some like GPT-3 which are decoder-only models. Encoder-only models are good for language modeling while decoder-only models are used for predictive tasks. There are other models like T5 and Code-T5 which contain both the encoder and decoder stacks.

Table 1 below gives a list of pre-trained language models that are built on transformer architecture (Qiu et al., 2020) and (Han et al., 2020).

Table 1
List of pre-trained language models

Pre-trained Model	**Transformer Architecture**	**# of Params**	**Data**	**Fine-tuning**
GPT	Decoder Only	117M	BookCorpus	Yes
GPT-2	Decoder Only	117M-1542M	WebText	No
GPT-3	Decoder Only	125M-175B	CommonCrawl + WebText2 + Books1 + Books2 + Wikipedia	No
BERT	Encoder Only	110M-340M	WikiEn + BookCorpus	Yes
RoBERTa	Encoder Only	355M	BookCorpus + CC-News + OpenWebText + STORIES	Yes
XLNet	Two-Stream Encoder Only	≈ BERT	WikiEn + BookCorpus + Giga5 + ClueWeb + Common Crawl	Yes
ELECTRA	Encoder Only	335M	same as XLNet	Yes
UniLM	Encoder Only	340M	WikiEn+BookCorpus	Yes
BART	Transformer (Encoder + Decoder)	110% of BERT	same as RoBERTa	Yes
T5	Transformer (Encoder + Decoder)	220M-11B	Colossal Clean Crawled Corpus (C4)	Yes
ERNIE (THU)	Encoder Only	114M	WikiEn + Wikidata	Yes
KnowBERT	Encoder Only	253M-523M	WikiEn + WordNet/Wiki	Yes

GPT-3, short for Generative Pre-trained Transformer, is a 175 billion-parameter auto-regressive language model (Brown et al., 2020). Recently, Chen et al., 2021 introduced Codex – a GPT language model finetuned on GitHub code. Models like GPT-3 and Codex are pre-trained on a vast amount of unlabeled data followed by discriminative fine-tuning on a specific task. Codex can understand multiple languages like Python, Java, C++, Javascript, Perl, and many more and can be used for many code-related tasks. Figure 1 below shows the various code-related tasks that can be performed using Codex. Github Copilot (Your AI pair programmer, no date) is also powered by OpenAI Codex.

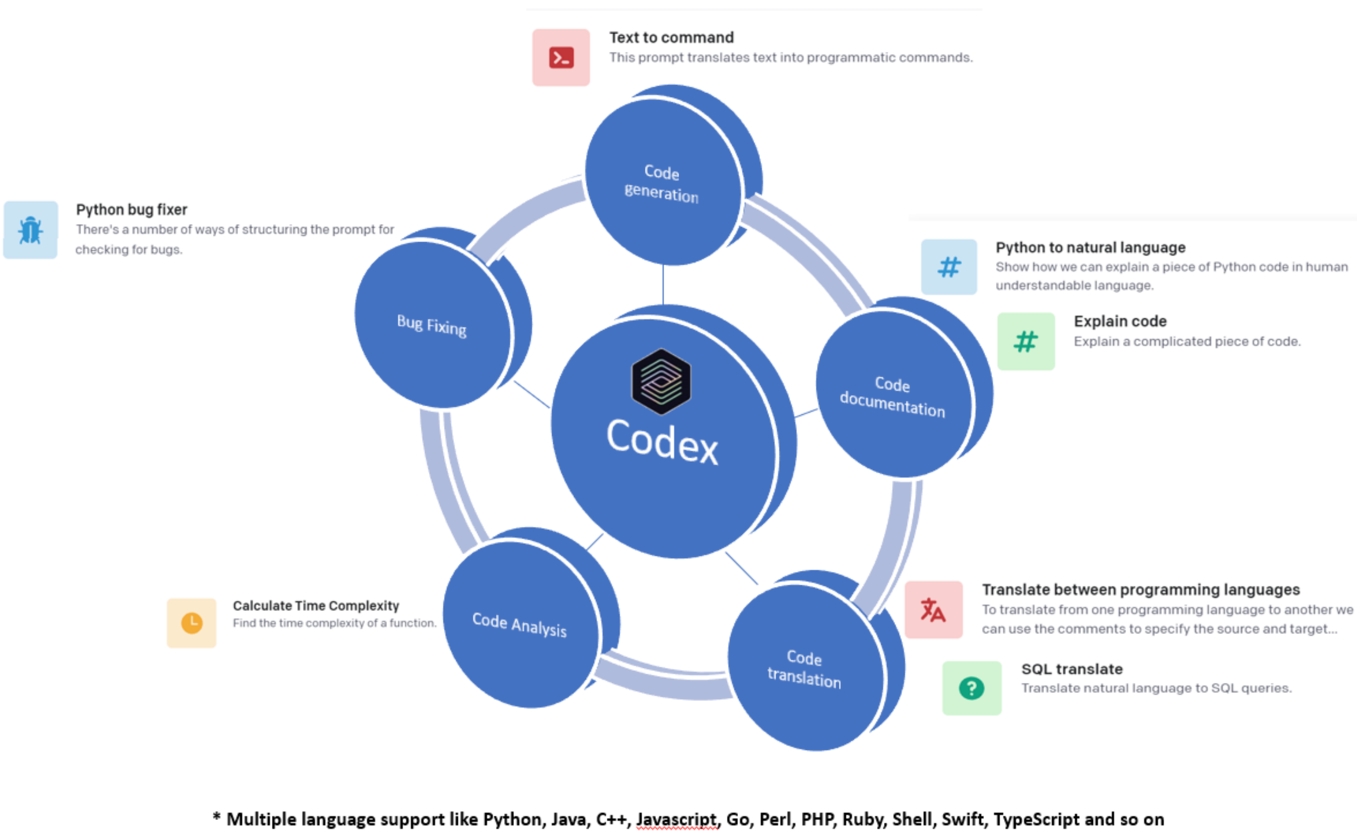


Figure 1
Codex

OpenAI recently released new versions of its models, ChatGPT (Introducing ChatGPT, 2022) also known as gpt3.5-turbo, and GPT-4 (OpenAI, 2023). Both these models are based on the previous version of the instruct series of GPT-3 models but have been trained differently, using Reinforcement learning with Human Feedback. Both these models can understand different programming languages and can perform code-related tasks either similar to or even better than models like Codex.

Models like GPT-3, Codex, and GPT-4 are only available as APIs. Hence, they cannot be deployed in one's environment. EleutherAI released GPT-J (EleutherAI - text generation testing UI, no date) – a 6 billion parameter open-source model similar to GPT-3. GPT-J training data contains a good amount of GitHub code. Hence, the code-related capabilities of GPT-J are better than GPT-3. Since GPT-J is open-source, it can be deployed in one's environment and it is also possible to fine-tune it with custom datasets for different code-related tasks. Fine-tuning greatly helps in improving the performance of the pre-trained model on custom code-related tasks. Perez, Ottens and Viswanathan, 2021 have demonstrated how fine-tuning the small GPT-2 (117 million parameters) model improved its code-generation capability achieving a BLEU score of 0.22.

CodeBERT (Feng et al., 2020) is another transformer-based model built for various natural language – programming language tasks. CodeBERT can be used for predicting defects (Pan, Minyan and Biao, 2021) and to automatically fix bugs in Java programs (Mashhadi and Hemmati, 2021). Gandhi, 2020 proposes how CodeBERT can be used for the generation of code documentation as well. BERT models can also be used for code completion as discussed by Ciniselli et al., 2021.

Recently many transformer-based models have been released like GPT-Neox (GPT-NeoX, no date), OPT (Alford, 2022) and Bloom (BigScience, 2022). All these models are open-source models and can be utilized for different code-related tasks. Like GPT-J, these models can also be finetuned for a given data set and given requirement.

AI-based language models can perform many code-related tasks other than sentence completion or signature recommendation as opposed to the different code assistant tools. These models have not yet been fully explored and can positively impact the software development industry and the role of software developers.

2.6 Impact of Artificial Intelligence on Software Professionals

Over the years, there have been many debates on whether Artificial Intelligence will replace humans. In a detailed survey (Alkashri, Siyam and Alqaryouti, 2020), Siyam N talks about how Artificial Intelligence is slowly moving towards Artificial General Intelligence and the possibility of Artificial Super Intelligence. It also discusses the impact of Artificial Intelligence on employment and how the role of software engineers will change gradually. A 2016 study from McKinsey indicates that almost 78% of work falls under the "predictable physical work" category and 69 % of jobs involving data processing will be replaced by machines. Concerning the role of Software engineers, there are different opinions on whether AI will replace them. Guelfi, 2018 discusses his idea about "senseware" engineers and believes that software engineers will be replaced with advanced AI technology. Similarly, there have been predictions that almost 70% of IT jobs will be killed due to automation (Venkatesh, 2017).

On the other hand, there are other opinions (Schatsky and Bumb, 2020) that suggest that AI will only help augment the work of software developers. In one of the studies (Zohair, 2018) Lubna studies the impact of AI on software engineers and the future of software engineering by 2050. Lubna brings out the fact that although software engineers cannot be replaced, they will need to reskill and adapt to the latest changes in technology and advances in AI. In "Artificial Intelligence and IT Professionals" (Mithas, Kude and Whitaker, 2018), the authors discuss that though there have been predictions that software developer jobs will be reduced by 70% in India, such predictions are not new. Almost six decades ago, Herbert Simon made a similar prediction on how “self-programming techniques” could lead to the extinction of software developers. The authors use an approach to first identify the contributing factors to the changes in demand for software programmers, study the relation of these factors with AI and then analyze how AI can impact them.

Thus, with the rise in powerful transformer-based large language models or LLM, it is important to study their impact on the industry.

2.7 Evaluation of AI models

To correctly assess the impact of AI models on the job of software developers, it is vital to evaluate the efficiency and usefulness of these models. Kulkarni and Padmanabham, 2017, integrate different AI activities like machine learning, statistical model, knowledge representation, intelligent decision-making, and so on into both extended waterfall models and agile models. The authors then study the effectiveness of this integration with the software development process. They have used the metrics UGAM and IoI in their study.

In the context of source code, there are many metrics for static source code. However, when it comes to automated code generation, limited research has been conducted. Li et al., 2020 propose a metric model for automatic code generation consisting of six different metrics based on the efficiency and quality of generated code.

Narasimhan, Venkatesha Rao and M B, 2021 propose a metric CGEM (Code Generation Evaluation Metrics) to validate the code generated using GPT-3. Amongst other metrics, CGEM contains metrics like LOC, BLEU score, ROUGE-1, 2, L score, number of edits required, number of compilation errors, and so on. Eighty codes are generated using GPT-3 and these metrics are then applied to them. Based on the results, the codes are then classified as acceptable or not-acceptable using artificial neural network modeling. During training, the model gave an accuracy of 76.81% and during testing, the accuracy was 61.54%.

The metric models mentioned above are only for automatic code generation. However, AI models like GPT-3, Codex, and GPT-J can augment other code-related tasks as well like code documentation, code translation, and fixing bugs. There is a need to come up with some framework that could help assess the impact of AI-based models on both cost and effort. With the help of this framework, it should be possible to correctly gauge how these models will affect the software developers and positively favor the software industry.

In "How much automation does a data scientist want" (D. Wang et al., 2021), the authors study how much automation data scientists and machine learning engineers need in their workflows. The authors have suggested a human-centric framework with different personas, tasks, sub-tasks, and levels of automation. Based on this framework, they further design an online survey study that helps them conclude. A similar approach can be used for measuring how much code automation can be done using the pre-trained models. This can then, further, be extended to analyze the kind of jobs that this automation can replace.

CHAPTER III:

METHODOLOGY

3.1 Research Purpose and Questions

As we have seen so far, the software development process is costly and becoming increasingly difficult especially due to the chronic shortage of specialized software developers. Code automation serves as a good option to overcome this challenge. We have also seen that rule-based systems for code automation have several drawbacks. Similarly, approaches in AI like Neural Machine Translation need a huge amount of labeled data making this approach difficult and costly. This led to the revolution in AI with pre-trained models like GPT-3, GPT-J, and Codex which have code generation and code documentation capabilities.

Although large language models are effective for various natural language tasks, their use for code automation has not been explored in depth. It is unclear how large language models can be effectively used to automate code tasks such as code completion and error correction. There are not many studies comparing the efficiency and effectiveness of large language models to traditional code completion tools. Also, the potential benefits of using large language models for code automation have not been fully explored. There is also a lack of research on how code automation will impact the software industry and software professionals. There is a need to investigate how code automation will change the way software is developed and maintained, and how it will impact the career paths of software professionals. Thus, to summarize, there is a need for research that evaluates the effectiveness of using large language models for code automation in comparison to other methods of code development and measure its impact on software professionals.

This research aims to fill in this research gap and answer the following questions:

1. What software development tasks can be automated using artificial intelligence-based language models?
2. How do large language models' code automation capabilities compare against traditional code generation?
3. How much effort saving and cost-benefit will organizations get by augmenting these AI-based models with a software programmer?
4. What types of jobs will be completely replaced, if any, due to the adoption of AI in the software industry?
5. What is the new set of skills that will emerge as AI is increasingly adopted across the organization for software development?

3.2 Research Design and Strategies

3.2.1 Research Design

To answer the proposed research question, a blended approach of qualitative and quantitative techniques would be employed. Survey research would be used to gather information on the tasks which programmers think should be automated. It will also help understand the programmer's point of view towards code automation - whether programmers are inclined towards automating code-related tasks or prefer performing the task manually.

Sample Survey Questions:

Q1. Describe your role: Student, Programmer, Manager, or Other

Q2. Rate your programming skills on a scale of 1 to 4

Q3. What programming language are you comfortable with:

Java, C++, Python, .NET, Javascript, C, C#, Go, R, Swift, PHP, Perl, Ruby, Scala, Other

Q4. Time (in mins) you would take to code a simple Java program

Q5. Time (in mins) you would take to code a simple Python program

Q6. Time (in mins) you would take to code a simple C++ program

Q7. Time (in mins) you would take to code a medium-complexity program

Q8. Time (in mins) you would take to code a complex program

Q9. Select the reason for the slow development time

Q10. Would you prefer to write code from scratch compared to using templates?

Q11. Would you prefer to make use of an automatic code generation utility for code completion?

By using quantitative methods, some experiments will be performed in a controlled environment where I will study the impact of automating the code-related tasks using per-trained models. The results from the experiments will be analyzed to measure the amount of cost and effort savings we get as a result of this automation. Based on both the qualitative and quantitative research done, I will design a framework to evaluate the benefits of code automation and the kind of reskilling needed, if any.

3.2.2 Sample Selection

For my sample I would want to reach out to programmers with different levels of programming experience and who are willing to to take part in the study. The objective is to receive five thousand responses to the survey and have a small group of programmers who would be involved in the experiment. The setup of the experiment will be as follows:

1. For each programming language, there will be six programmers needed. These six programmers will be divided into three groups.
2. The first group will contain an expert programmer, the second group will have a programmer with medium programming skills and the third group will have a programmer with basic programming skills in the concerned programming

language. These programmers will be assigned a code-related task to be done manually without any automation.

3. Each of these groups will also contain a programmer with basic knowledge of the concerned programming language who will perform the same code-related task with the help of pre-trained language models like Codex, GPT-3, GPT-J, or any other such model.

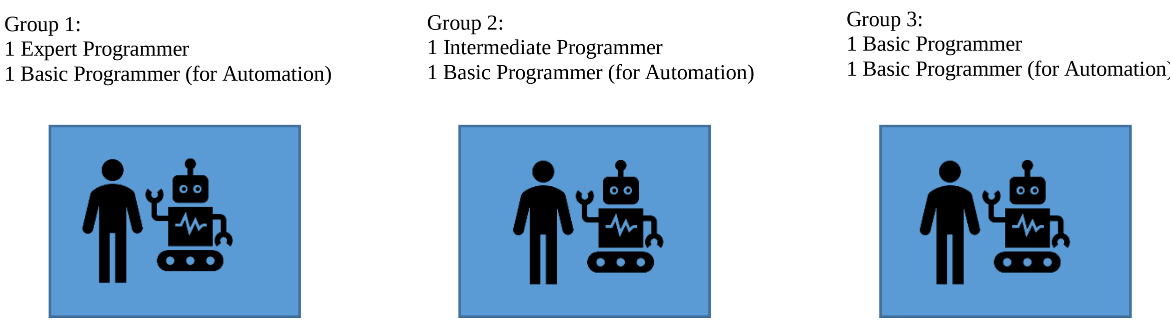


Figure 2
Experiment Setup

This setup will be repeated for different programming languages and different code-related tasks. The intention will be to analyze the effort saving we get by automating different code-related tasks. By assigning programmers with different levels of expertise and experience, we will also be able to analyze the cost-saving we can get by replacing one expensive resource with a cheaper resource and augmenting it with Artificial Intelligence based models. The data collected and analyzed in this manner will also help chart the skills needed to work with AI-based code automation models. Thus, it will help me answer my research questions.

3.2.3 Participant Selection

A wide range of developers will be considered for participation in the survey. This would range from freshers to experts in the said programming language. Since code automation can greatly impact the future of the software industry, a larger focus will be on the new generation who will be using these tools.

Fresh graduates/students pursuing bachelors/masters with programming knowledge will constitute around 60% of the participants. 20% of the participants will be programmers with between 2 to 5 years of programming experience. The remaining 20% of the participants will be software developers with 5+ years of programming experience.

For the experiment setup, participants with experience in different programming languages will be considered. Also, it will be ensured that programmers with different levels of expertise participate in this exercise.

3.2.4 Sources of Data and Data Collection Procedures

To validate different tasks that can be automated using large pre-trained models, there is a dependency on small datasets that can be used for few-shot learning or fine-tuning. For this purpose, free datasets for different tasks have been identified. A few of the sources are:

- Code-T5 dataset ("Code-T5 Dataset," no date) for code summarization, generating code from text, code translation, code defect detection, and so on.
- LeetCode solutions (kamyu, 2020) also has a repository of code solutions in C++, Python, and Java. Since these solutions are for the same coding task – in different programming languages, they also serve as parallel data that may be needed to validate the task at hand.
- Similarly, a dataset for Java code with documentation is available at http://leclair.tech/data/funcom/ (LeClair, A. and McMillan, C.).
- CodeSearchNet dataset (https://github.com/github/CodeSearchNet) contains around 2 million pairs of comments and equivalent code which can be utilized for fine-tuning and extracting samples for few-shot learning.

- CodeTrans for Java-C# translation (CodeXGLUE, 2020)

3.3 Data Analysis

Our first research question is **“**What are the software development tasks that can be automated using artificial intelligence-based language models?”

To answer this question, we will work with different pre-trained models like Codex, GPT-3, GPT-J, Code-T5, and CodeBERT. All these models are pre-trained on code datasets and capable of performing different code automation tasks. If possible, we will try to do a comparative study of these models for related tasks and present the results.

For example, Table 105 in Appendix B, shows the code generation capability of GPT-3, GPT-J, and Codex for a simple Python program. The instruction is given in natural language to take the name, age, and roll no as input from the user and write it to the file. Similarly, an example of a code documentation task that can be performed using the pre-trained Codex model is shown in Figure 79 in Appendix B. Similarly, we will try other tasks possible with Codex and other pre-trained models.

Our second research question is “How do large language models’ code automation capabilities compare against traditional code generation?”

Traditionally, code is written manually from scratch or by referring to various online resources like GitHub, StackOverflow, and so on. Also, traditional code translation tools are rules-based. To answer this question, we will show through some experiments, the quality of code generated or translated by large language models. We will also compare these capabilities with some standard IDE that are used by software developers to augment their coding tasks. We aim to show that such code generation, translation, or documentation capabilities provided by these large language models are better compared to the standard options available to a software programmer.

Our next research question is “How much effort-saving and cost-benefit will organizations get by augmenting these AI-based models with a software programmer?”

To answer this question, based on the survey conducted, we will select three programming languages and three tasks that the programmer community is interested in. Then we will perform experiments using the strategy mentioned under “Sample selection”.

The research methodology to be adopted to answer this question is showcased below with the help of an example:

1. Code automation task to be done - Code Translation (from Python to Java)
2. The pre-trained model identified – Codex
3. Code Complexity – Simple
4. Code objective - Add two numbers using a linked list
5. Experiment Group - One programmer with medium programming skills and another with basic skills (to auto-generate code using pre-trained models)

Figure 80 in Appendix B shows the Python program which needs to be translated to Java. The code shown in Figure 80 is fed to Codex which then translates it into equivalent Java code. Figure 81 in Appendix B shows the Java code which is generated using Codex. A few lines of code need to be added to the program as shown in Figure 82 in Appendix B to test the program. This program generated by Codex is a working program with no errors and correct logic.

Parallelly, a programmer with intermediate programming skills will be assigned to generate the equivalent Java program manually. The screenshot in Figure 83 in Appendix B shows the program written manually.

Analysis:

Table 2
Analysis of Code Translation using Codex

No. of lines of Python Code	37
No. of lines generated by Java Code	37
No. of lines added to test the program	8
Does translated program compile?	Yes
Does the translated program execute and give correct results?	Yes
Time taken to translate the program using Codex	30 sec
Time taken to copy-paste the code in Java editor and add the extra lines of code to test and run the program	5 min 30 sec
Total time for end-to-end translation of simple Python program to Java	6 min
Approximate time to translate code from scratch and execute it (with intermediate programming skills)	25 min
Effort saving	19 min
*Cost-saving	One expensive resource replaced with one cheaper resource
Skills needed	Basic programming and debugging

*In terms of cost-saving, based on project size, replacing three software developers (at an average rate of $100/hour) with one developer, can give a cost-saving of almost 60%.

The same process will be repeated with the other two groups with one programmer of different programming expertise and another with basic programming skills for auto-generation of code. Similarly, the task will also be repeated using other pre-trained models like GPT-3, GPT-J, CodeBERT, and Code-T5. Post the experiment, we will conduct another survey to study the effectiveness of these models for code automation.

We will develop a framework to measure the cost-benefit one can get by utilizing these models. An example of measuring the benefits of utilizing automation in code translation for a large program with 10 million lines of code is shown below (code is fed by breaking it down into smaller blocks):

Table 3
Example of Cost-Benefit Analysis for Code Translation

	Manual Translation	**Translation using AI model**	**Saving**
LOC per block	1000	1000	
Total no. of blocks of code	10,000	10,000	
% Effort saving using AI (Minimum assured)		40%	40%
Developer's translation effort (hours per LOC block)	40	24	
Developer's rate ($ per hour)	40	40	
Developer's cost to client ($) / block of code	1600	960	
Developer's cost to client ($) for 10 million lines of code	16,000,000	9,600,000	6,400,000
Cost of manual effort (correction/validation) needed with AI model		600,000	
Total cost to client (per block)	16,000,000	10,200,000	36%

*Saving to the client is near ~ 6 million dollars or 36% of the original cost

The answer to the next two research questions "What types of jobs will be completely replaced, if any, due to adoption of AI in the software industry?" and "What are the new set of skills that will emerge as AI is increasingly adopted across the organization for software development?" will depend on the exhaustive experiments conducted using different models.

A preliminary analysis suggests that for complex programs, for which one needs to refer to multiple resources, large models like Codex and GPT-J can help generate template code. Manual intervention will however be needed to check for logical errors.

The current analysis also suggests that strong programming fundamentals with good debugging skills will be needed to work with these models. However, expert programming skills may not be needed as the model can help generate at least 60 percent of the code.

3.4 Research Hypothesis

Based on the research questions, we formulate the following hypotheses:

1. *The code generation/translation capabilities of large language models are better than traditional code generation and rule-based tools.*

We have seen in the literature review that traditional tools used for code automation either help with a single line of code/method completion or use rule-based methods for code translation. Hence, they do not provide an optimal way of code automation. When it comes to code generation, large language models do not suggest a single line or method completion but are capable of completing an entire piece of code.

Also, regarding code translation, large language models, do not perform a one-one translation of code from source to target, as is the case with traditional code translation tools. They, on the contrary, rely on understanding the semantics of the program and suggest an optimal translation of the given piece of code.

We aim to prove this hypothesis by performing a few experiments outlined in the sections above.

2. *If large language models are to be used optimally, then a human-in-the-loop is necessary for validation of the result generated by the model.*

Even though large pre-trained language models can generate code, the generated output, depending on the complexity of the task, may not be completely accurate. We hypothesize that manual intervention will remain a part of these techniques of code generation, where the code generated by the model will be validated, corrected, and then implemented by the software developer as part of the bigger project.

3. *If code automation using large language models is leveraged in the industry, it can improve software quality by reducing errors and increasing consistency.*

Manual code generation is prone to errors. Also, with different programmers following different programming styles, code inconsistency is bound to rise. We hypothesize that if we make use of large language models for automatic code generation, these inconsistencies will reduce since the code generated by the model will be consistent and can be directed to follow best practices.

4. *If large language models are used for code automation, it can improve software development efficiency by reducing the time needed to develop code and provide significant cost and effort benefits to software organizations.*

We hypothesize that large language models like Codex can greatly help reduce the time needed to develop code with a good amount of accuracy. This can, in turn, help a company decrease the time to market its software products. We will evaluate these models for the time and effort spent on code generation. We plan to prove this hypothesis by performing various experiments and thus show that they can provide significant cost and effort benefits to the organization.

5. *If code automation using large language models is adopted in the software industry, it can improve software development flexibility by allowing developers to focus on higher-level tasks and by providing more options for code generation.*

With AI performing repetitive and tedious coding tasks, we hypothesize that the time and energy of software professionals can be directed toward higher-level tasks and problem-solving.

6. *If code automation using large language models is adopted in the industry, then reskilling software professionals is necessary.*

We aim to show that AI models will provide significant benefits to organizations in terms of both cost and effort. At the same time, we hypothesize that the role of a software developer will not diminish. Rather, there will be a shift where automation will take over the task of performing all the trivial tasks, and software engineers can concentrate on high-level aspects. As code automation tools become more sophisticated, developers may find themselves doing less coding and more debugging, testing, and troubleshooting.

Prompt engineering is another skill that will need a developer's attention. Providing the right prompts for guiding the model to generate the best output is a skill that software developers will have to learn.

7. *If code automation continues to increase in popularity and effectiveness, it will have a significant impact on the software industry and software professionals.*

We hypothesize that if code automation using large language models proves to be effective and becomes popular, it is likely to impact the software industry in many ways. First, as suggested in the earlier hypothesis, the role of software developers may change. Second, the demand for software developers may change. As code automation becomes more common, businesses may be less likely to need as many software developers on staff. Instead, they may contract with code automation service providers to handle their coding needs. Third, the nature of software development may change. As code automation tools become more prevalent, the software development process may become more streamlined and efficient. The cost of software development may change. As code automation tools become more common, the cost of developing software may decrease. This could make it more affordable for small businesses and startups to develop their software applications. Finally, it will allow for the rapid creation of new software applications and the ability to rapidly deploy them. This will lead to a more competitive software industry and increased pressure on software professionals to keep up with the latest trends.

3.5 Research Design Limitations

The approach described above can be applied to extensively understand the benefits of three major code automation tasks. However, under code automation, many tasks can be proposed to be automated. The limitation of this approach is in analyzing the benefits of code automation for all possible tasks.

Also, every language model has a limit on the total number of tokens. For example, GPT-J and GPT-3 have a limit of 2048 tokens and Codex has a limit of 4096 tokens. Hence, a single large program cannot be fed to these models. We will be limiting ourselves to programs that are not very big and do not exceed the required number of tokens.

CHAPTER IV:

RESULTS

4.1 Survey Results

A survey was conducted to gather information on the most used programming languages, estimate the time taken to code manually from scratch, and the preference of the audience towards automating code-related tasks. Survey results are presented in Appendix A. The survey results indicate that the most preferred programming languages are Java, Python, and C++. We also observe that as the complexity of the program increases, the effort needed for the same also increases. The unavailability of sufficient skilled resources and the rapid technology change appear as the top two reasons behind the slow development time. The survey also indicates that the community is open to the use of automation in the area of software development.

We then proceeded with validating the hypothesis and arriving at the answers to our research questions.

Research Question Findings

4.2 Research Question One

"What are the software development tasks that can be automated using artificial intelligence-based language models?"

There are many different tasks involved in software development, but some of the most common include planning and requirements gathering, design, coding, testing, and documentation. AI-based models can be used to automate all of these tasks, making the software development process more efficient and effective.

4.2.1 Planning and requirements gathering

One of the most important aspects of software development is planning and gathering the requirements. This entails understanding what the software needs to do, and what the user wants it to do. Software development teams often expect business customers to articulate their requirements in a clear and concise manner, while business customers anticipate development teams to provide a solution based on ambiguous and unstated, or unknown requirements. However, these expectations are often unrealistic. Therefore, it is essential to explicitly document the requirements in a consolidated document that can serve as a reference during the software development process.

Requirements gathering is largely done manually. However, there are a few areas where AI can help augment this phase of software development.

4.2.1.1 Software requirement classification

This helps in understanding the requirements better and also helps in scoping the project. Manually, this is a time-consuming task, but AI can help automate it to some extent. Software requirements broadly fall into two major categories: functional requirements and non-functional requirements. Non-functional requirements can further be classified into other categories like performance, security, usability, maintainability, scalability, and so on. Software requirement classification can be done by analyzing the natural language requirements documents using NLP techniques. This will help identify the different types of requirements, including functional, non-functional, design, etc.

We tested the capability of a large model like GPT-3 to perform this task. The model was first asked some questions to check its knowledge on functional and non-functional requirements. Then, the model was given the task to classify a given requirement into one of the non-functional requirements. Figure 3 and Figure 4 below show the capability of the model in understanding software requirements.

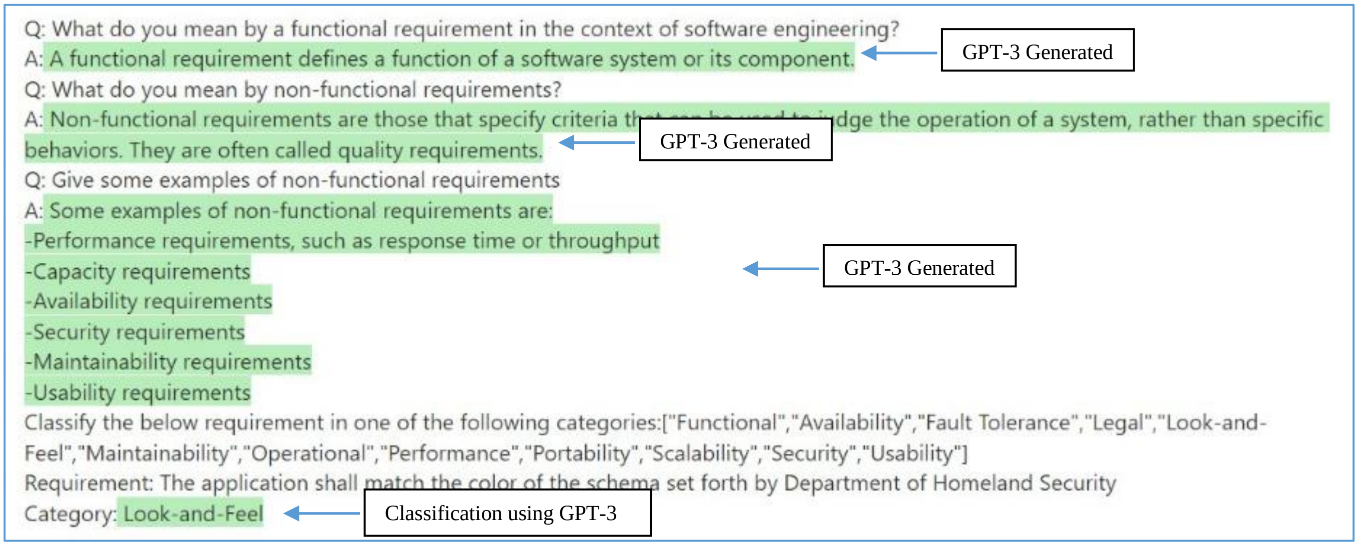


Figure 3
GPT-3 understanding about software requirements – Look and Feel

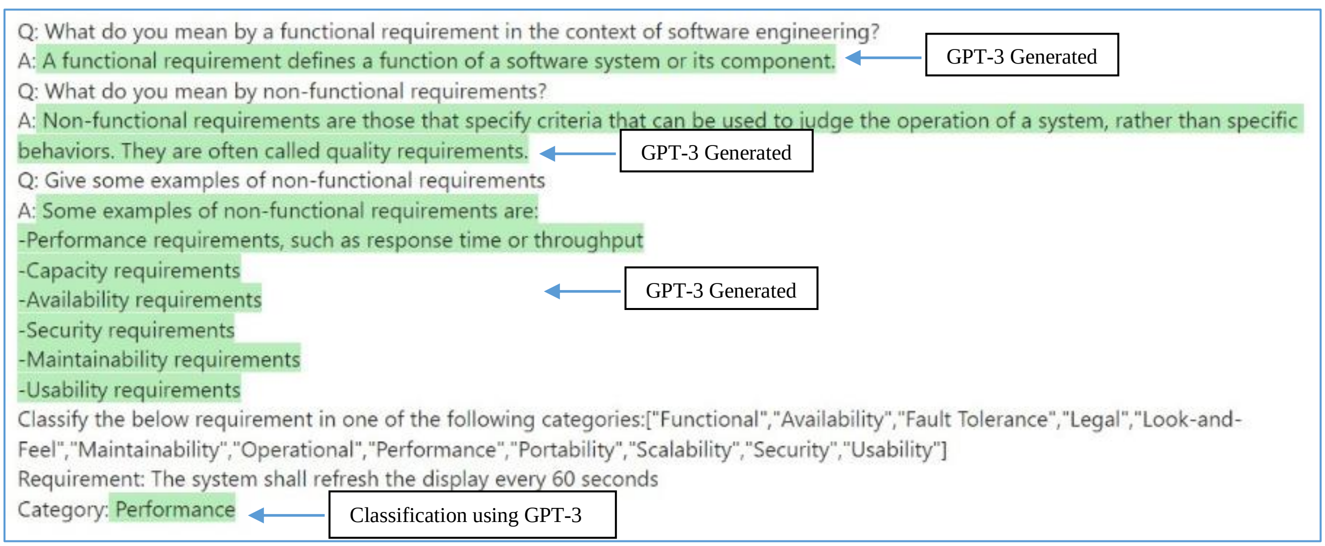


Figure 4
GPT-3 understanding about software requirements – Performance

The model was further tested on the Promise NFR (PROMISE Software Engineering Repository, no date) dataset to measure its capability against other traditional methods and other ML models. The detailed approach and results are provided in the next section 4.3.1.

4.2.1.2 Software requirement clustering

Software requirements clustering, like, software requirements classification, is another approach for organizing and analyzing requirements in the software development process. While both approaches can be useful for understanding and prioritizing user needs, there are some key differences between the two:

- Purpose: Grouping similar requirements is the primary aim of software requirements clustering, whereas the main goal of software requirements classification is to assign requirements to predefined categories or labels.
- Granularity: Software requirements clustering tends to be more granular than classification, as it involves grouping similar requirements rather than assigning them to broad categories. This can help developers to identify specific themes and patterns in user needs and to understand the context and implications of each requirement more fully.
- Benefits: Software requirements clustering can help developers to more efficiently and effectively understand and prioritize user needs, as it allows them to identify specific themes and patterns in the requirements. Classification, on the other hand, can be useful for organizing and tracking requirements, as it allows developers to assign requirements to specific categories or labels.

Overall, software requirements clustering and classification are both useful tools for understanding and prioritizing user needs in the software development process. Since large language models like GPT-3 have natural language understanding capabilities, they can be utilized for this task of clustering software requirements. We tested the capability of GPT-3 to cluster a set of software requirements.

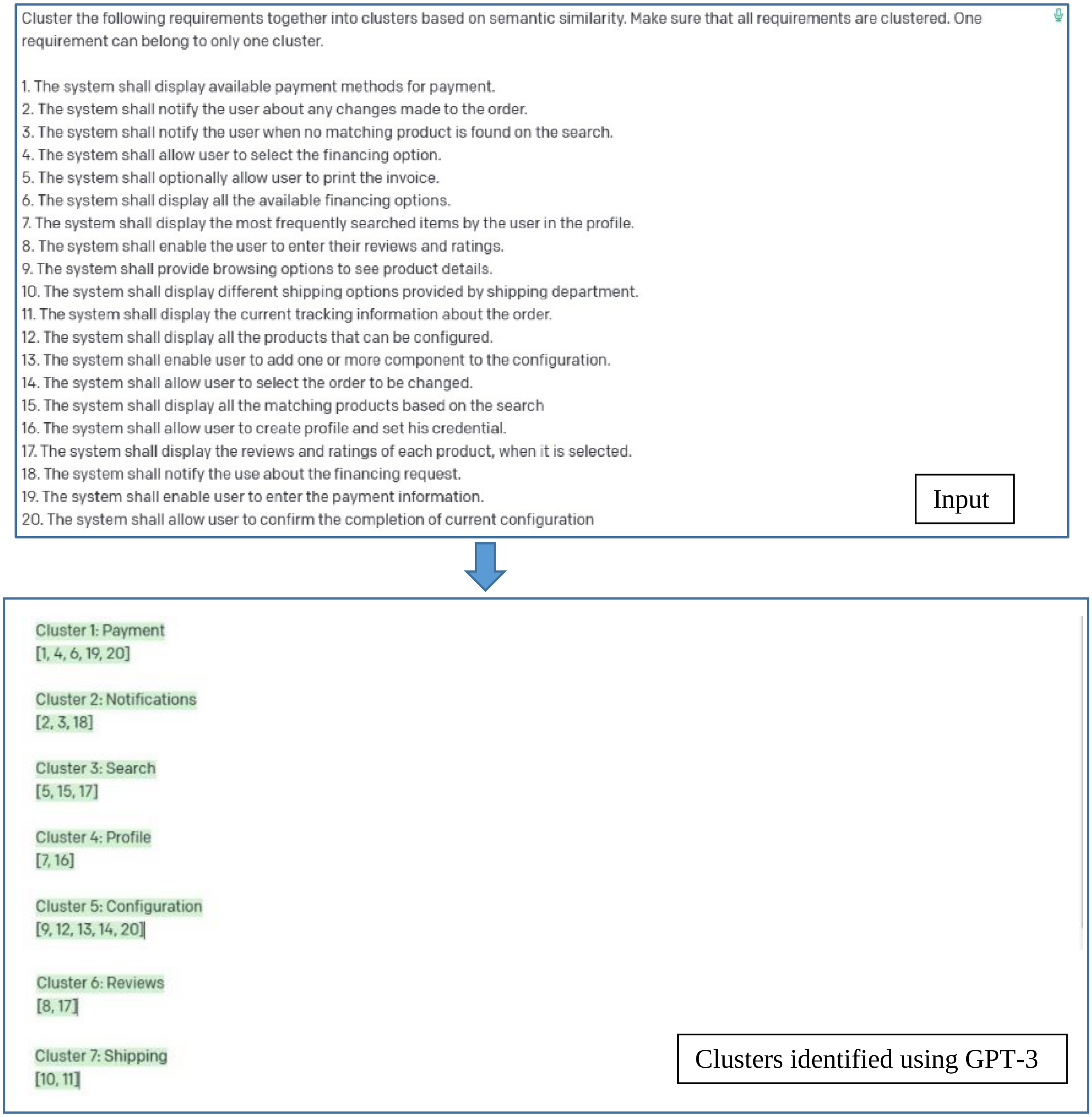


Figure 5
Clustering using GPT-3

As we can see in Figure 5, GPT-3 model can form logical clusters given a set of software requirements, without any fine-tuning or few-shot learning.

4.2.1.3 Automated requirement tracing

Automated requirement tracing is the process of establishing and maintaining traceability between software requirements and other artifacts in the development process, such as code, test cases, and design documents. This helps developers verify all requirements are fully addressed and tested, and to determine any gaps or inconsistencies in the requirements.

Large language models like GPT-3 and BERT are powerful models for various natural language processing (NLP) tasks. They can be used to automate the requirement tracing process, by analyzing the content of requirements and other artifacts and identifying relationships between them. To use a large language model for automated requirement tracing, developers can input the requirements and other artifacts into the model and train it to recognize specific phrases or patterns that indicate a relationship between them. For example, the model might be trained to recognize phrases such as "This requirement is related to" or "This design document addresses". The model could then extract these phrases and generate a list of relationships between the requirements and other artifacts.

Here is an example of how a model like GPT-3 can be used for this purpose: Consider a software development project where the requirement's gathering phase has yielded the following requirement:

Input - Requirement (R1): "The system shall support multiple languages, including English, Spanish, and French."

During the design, implementation, and testing phases, various artifacts related to this requirement are created. However, the implementation has an issue that needs to be addressed.

Example Artifacts:

Design Artifact (D1): A detailed design document that includes the following excerpt:

The system will use a localization library, such as i18next, to provide internationalization support for English, Spanish, and French languages. Language files containing translations for each supported language will be stored in the /locales directory.

Code Artifact (C1): A snippet of code implementing the language support, but mistakenly only including English and Spanish.

```
import i18next from "i18next";
i18next.init({
  resources: {
    en: { translation: require("./locales/en.json") },
    es: { translation: require("./locales/es.json") },
  },
});
```

Test Artifact (T1): A test case verifying that the system works as expected in English and Spanish, but missing test cases for French language support.

```
describe("Language support", () => {
  it("supports English language", () => {
    i18next.changeLanguage("en");
    expect(i18next.t("welcome_message")).toBe("Welcome!");
  });
it("supports Spanish language", () => {
    i18next.changeLanguage("es");
    expect(i18next.t("welcome_message")).toBe("¡Bienvenido!");
  });
});
```

Output - Traceability Matrix (produced by GPT-3):

Table 4
Traceability Matrix

Requirement	Design Artifact	Code Artifact	Test Artifact
The system shall support multiple languages, including...	Excerpt from design document (D1) describing...	Sample code (C1)	Test cases (T1)

Upon reviewing the traceability matrix and examining the artifacts, it becomes apparent that both the code artifact (C1) and the test artifact (T1) are missing support for the French language, which is part of the original requirement (R1). The development team can then address this issue by updating the implementation and test cases accordingly.

In this example, the traceability matrix helps identify an implementation issue by highlighting the discrepancy between the original requirement and the related artifacts. By examining the actual artifacts in detail, the team can detect and correct the issue. Similarly, another example of how a model like GPT-3 can be used for this purpose:

- Input the requirements and other artifacts into GPT-3: The developers input the requirements and other artifacts (such as design documents and test cases) into GPT-3 as text data.
- Train the model to recognize relationships: The model can then be fine-tuned or used to recognize specific phrases or patterns that indicate a relationship between the requirements and other artifacts using prompt-engineering techniques. For example, the model can be used to recognize the phrase "This requirement is related to" as an indication that the requirement is linked to another artifact.

- Extract the relationships: The model analyzes the text data and extracts the relationships between the requirements and other artifacts based on the phrases and patterns it has learned to recognize.
- Track changes and identify conflicts: The developers can use the model to track changes to the requirements and other artifacts over time, ensuring that traceability is maintained as the project progresses. The model can also be used to identify any potential conflicts or inconsistencies between the requirements and other artifacts, helping developers to identify and resolve any issues.

There are many potential applications for using large language models for automated requirement tracing in the industry. For example, a software development team could use the model to establish traceability between requirements and design documents, helping to ensure that all requirements are fully addressed in the design. A quality assurance (QA) team could use the model to trace requirements to test cases, helping to ensure that all requirements are fully tested. And a project manager could use the model to identify dependencies and constraints in the requirements, helping to mitigate risks and ensure the project stays on track.

Overall, the use of large language models for automated requirement tracing can significantly improve the efficiency and accuracy of the requirement tracing process, helping developers to more fully understand and address user needs and to deliver software that meets those needs.

4.2.1.4 Similarity detection

Similarity detection in software requirements refers to the process of identifying similar requirements across different projects or within a single project. This can be useful for many reasons, including reducing the effort required for requirement gathering and helping to automate team formation.

One way to detect similarities in software requirements is by using natural language processing (NLP) techniques, such as those offered by large language models. By analyzing the text of the requirements and identifying common words and phrases, LLMs can identify requirements that are similar in content or theme.

For example, consider two software projects that both require a feature for filtering and sorting products. By analyzing the text of the requirements for these projects, an LLM might identify the common theme of "filtering and sorting products" and group the requirements together as similar. Figure 6 below shows how GPT-3 analyzes such requirements which are similar and generates one comprehensive requirement.

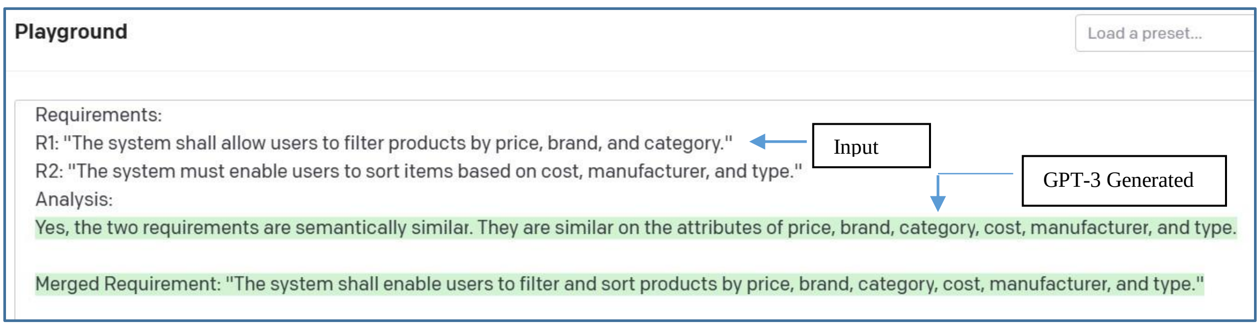


Figure 6
Similarity detection

Another approach to similarity detection in software requirements is to use machine learning algorithms to analyze the requirements and identify patterns or trends. For example, a machine learning algorithm could be trained on a dataset of software requirements and learn to identify common themes or patterns based on the words and phrases used in the requirements. The algorithm could then be used to identify similar requirements in new projects.

Similarity detection in software requirements can be useful for several purposes, such as reducing the effort required for requirement gathering and helping to automate team formation. By identifying similar requirements in different projects, developers can more easily reuse or adapt existing requirements, reducing the need to start from scratch and saving time and resources. Similarly, by identifying similar requirements within a single project, developers can more easily identify common themes and patterns, helping to inform the development roadmap and team formation.

Overall, similarity detection in software requirements can significantly help to improve the efficiency and accuracy of the requirement-gathering process. It can help developers to gain deeper comprehension of the requirements, address the needs of the user and to deliver software that meets those needs.

4.2.1.5 Automated requirement elicitation

This can be done by extracting requirements from unstructured sources, such as emails, chat logs, etc. For example, let's say a software development team is working on a new mobile app for booking flights. The team has received a large amount of customer feedback on the app, including comments and suggestions from users. The team can use a large pre-trained language model like GPT-3 to analyze this customer feedback and extract specific requirements or feature requests.

We tested the capability of GPT-3 to perform this task of eliciting software requirements from customer reviews. We used the app_reviews dataset on huggingface (Grano et al., 2017) for this purpose. GPT-3 was able to successfully identify the possible set of software requirements from a given set of customer reviews as shown in Figure 7. This indicates the usefulness of these large language models in eliciting software requirements from unstructured text.

Reviews:
1. Great app! The new version now works on my Bravia Android TV which is great as it's right by my rooftop aerial cable. The scan feature would be useful...any ETA on when this will be available? Also the option to import a list of bookmarks e.g. from a simple properties file would be useful.
2. Great It's not fully optimised and has some issues with crashing but still a nice app especially considering the price and it's open source.
3. Works on a Nexus 6p I'm still messing around with my hackrf but it works with my Nexus 6p Trond usb-c to usb host adapter. Thanks!
4. The bandwidth seemed to be limited to maximum 2 MHz or so. I tried to increase the bandwidth but not possible. I purchased this is because one of the pictures in the advertisement showed the 2.4GHz band with around 10MHz or more bandwidth. Is it not possible to increase the bandwidth? If not it is just the same performance as other free APPs.
5. Works well with my Hackrf Hopefully new updates will arrive for extra functions
6. Good job Tried a few . This has to be the most stable and best to my liking . And a whole world of settings to play with. However we are quite limited on frequency band width id like to see it lowered to 14.mhz if possible
7. Working great on Nexus6
8. Works with RTL and Nextbook Aries 8. Demod stops working if the scan width is changed requiring restart.
9. Works with RTL SDR Works but no audio when demodulating
10. Awsome App! Easy to use works great on Notes w / Realtek dongle.

Input

Requirements for next release:
1. Scan feature
2. Option to import a list of bookmarks from a simple properties file
3. Optimisation and bug fixes
4. Increase bandwidth to 2.4GHz with 10MHz or more bandwidth
5. Lower frequency bandwidth to 14MHz
6. Audio when demodulating
7. Compatibility with Nexus 6p, RTL SDR, and Nextbook Aries 8

GPT-3 Generated

Figure 7
Software requirements elicitation

4.2.1.6 Detect ambiguity in SRS documents

Ambiguity in software requirements specification (SRS) documents can be a major issue as it can lead to misunderstandings, confusion, and ultimately, project failures. Therefore, it is important to detect and resolve ambiguities in SRS documents as soon as feasible during the process of software development.

One way to detect ambiguity in SRS documents is to use LLMs. These transformer models, since trained on a vast amount of data, understand the nuances of natural language, making them well-suited for identifying ambiguities in written text.

For example, consider the following two requirements:

- The system shall allow users to search for products by name or description.
- The system shall allow users to search for products by name, description, or both.

At first glance, these requirements may seem similar, but they have different intentions. The first requirement specifies that users can search for products using either the name or the description, while the second requirement specifies that users can search using either the name, the description, or both.

A large language model could potentially flag these requirements as ambiguous because they are similar but have different intentions. This would allow the development team to clarify the requirements and ensure that they are understood correctly before proceeding with the development process. Some more examples of software functional requirements that are similar but have different intent and could potentially lead to ambiguity:

- "The system shall allow users to view their account balance."
- "The system shall allow users to view the balance of any account."

The first requirement specifies that users can view their account balance, while the second requirement specifies that users can view the balance of any account. This could potentially lead to ambiguity if it is not clear which of these requirements should take precedence, or if it is not clear how the system should handle conflicting requirements. Moreover, such a conflict could potentially have significant implications for the system's security and access controls. Hence, such ambiguity must be resolved. Another example:

- "The system shall display a list of all available products when the user selects the 'Browse Products' option."
- "The system shall display a list of all available products to all users."

The first requirement specifies that a list of available products should be displayed when the user selects the 'Browse Products' option, while the second requirement specifies that a list of available products should be displayed to all users at all times. This could potentially lead to ambiguity if it is not clear how the two requirements should be reconciled, or if it is not clear which of these requirements should take precedence. Also, such conflict could have significant implications for the system's user interface and the way that information is presented to users. Hence, resolving such ambiguities and conflict is critical. Large transformer models like GPT-3 and BERT can be used for the same.

It is important to identify and resolve ambiguities in software functional requirements, early on in the development process, to avoid misunderstandings and confusion and to ensure that the system is developed per the intended requirements. Another way to detect ambiguity in SRS documents is to use the model to specifically check for ambiguities. The model can be used to identify phrases or words that may be open to interpretation and flag them for further review. For example, consider the following requirement:

"*The system shall display a warning when the user attempts to delete a file that is currently in use.*"

This requirement could potentially be ambiguous because the phrase "currently in use" could be interpreted in different ways. For example, it could mean that the file is being accessed by another user, or it could mean that the file is being edited by the current user.

A language model can be used to check for ambiguities and potentially flag this requirement as ambiguous and prompt the development team to clarify the meaning of "currently in use" before proceeding with the development process. Some additional examples of ambiguous requirements are:

- “*The system shall allow users to enter their login credentials.*”

This requirement does not specify what the login credentials consist of (e.g., username and password, email, and password, etc.), which could lead to confusion and misunderstandings.

- "*The system shall display an error message when the user inputs an invalid value.*"
  This requirement does not specify what qualifies as an "invalid value," leaving it open to interpretation. It could refer to values that are outside a certain range, values that do not match a specific format, or any other type of value deemed "invalid" by the system.
- "*The system shall send a notification to the user's email address when a certain event occurs.*"
  This requirement does not specify which email address the notification should be sent to, or how the system should determine the correct email address to use. This could lead to confusion if the system has access to multiple email addresses for a given user.

In summary, detecting ambiguity in SRS documents is essential to a software development project's success. Large language models and tools that check for ambiguities can be useful in identifying and resolving ambiguities in written text, helping to ensure that requirements are understood and implemented correctly.

4.2.2 Design

Design is the process of translating the requirements into a representation of the software that can be used to guide its implementation. It is the process of creating a plan or blueprint for the software. This plan includes the overall structure of the software, the interfaces between the various components, and the algorithms that will be used to implement the functionality.

In most cases, the design is expressed as a diagram or a set of diagrams. Alternatively, the design could be described as a set of rules or constraints. The design phase is important because it is during this phase that the team decides how the software will be organized and how it will work. Multiple design tasks are done manually, like creating UML diagrams, website design, identifying the correct class hierarchy, etc. AI can help automate some of these tasks. The design phase can be automated to some extent using AI-based models.

4.2.2.1 Action extraction from requirements document

Action extraction from software requirements is an important step in the design phase of software development, as it helps to identify and understand the specific tasks and behaviors that the software should perform. Action recognition and actor recognition are important tasks in the process of action extraction from software requirements. Action recognition involves identifying the specific tasks and behaviors that the software should perform, while actor recognition involves identifying the entities that will perform these actions.

Here are some examples of action and actor recognition using large language models:

- Identifying specific actions: A large language model can analyze a requirement document and identify specific actions that the software should perform. For example, "The software should be able to search for a particular product by name" would be recognized as the action "search for a product by name."
- Identifying actors: The language model can also identify the entities that will perform the actions. In the example above, the actor might be "the user" or "the system."
- Extracting action details: The language model can also extract additional details about the actions, such as the input and output for each action. For example, "The software

should be able to search for a particular product by name and display a list of matching products" would be recognized as the action "search for a product by name" with input "product name" and output "list of matching products."

- Identifying conditional actions: The language model can also recognize conditional actions, such as "If the search returns no results, display a message saying 'No products found.'" In this case, the action is "display a message" and the condition is "if the search returns no results."

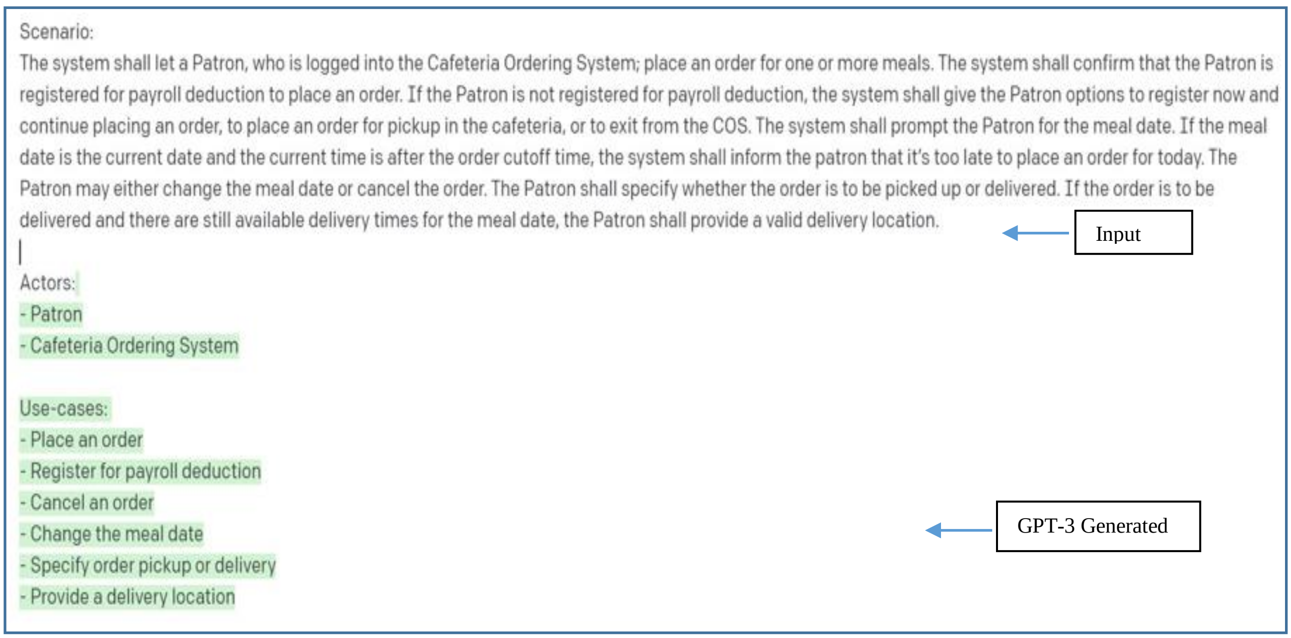
Use-case diagrams are a common way to represent the interactions between different actors and systems in software development. They can be used to describe how the system functions and the various ways in which it can be used.

Action and actor recognition from software requirements can be useful in creating use-case diagrams because it allows you to identify the various actions or use-cases that the system needs to support and the actors that will be interacting with the system. For example, for a given set of software requirements that describe the functionality of an e-commerce system, we can use action and actor recognition to identify the different actions that the system needs to support, such as searching for products, placing orders, and processing payments. We can also identify the actors that will be interacting with the system, such as customers, administrators, and payment processors.

Using this information, we can then create a use-case diagram that shows the different actions that the system needs to support and the actors that will be interacting with the system. We can also use the use-case diagram to understand the relationships and dependencies between different components of the system and how they interact with each other to support the various actions and actors.

Creating a use-case diagram is just one step in the software development process, and there may be additional steps involved in fully understanding and implementing the interactions between different components of the system. However, action and actor recognition can be a useful starting point for understanding the functionality of a system and identifying the various actors and actions that it needs to support.

We evaluated the capabilities of GPT-3 for the task of actor and use-case recognition. As we can see in Figure 8 below, GPT-3 is capable of successfully recognizing the actors and use-cases given a problem description. Here, we used one-shot, meaning we provided one sample for GPT-3 to understand the task needed to be performed.



Scenario:
The system shall let a Patron, who is logged into the Cafeteria Ordering System; place an order for one or more meals. The system shall confirm that the Patron is registered for payroll deduction to place an order. If the Patron is not registered for payroll deduction, the system shall give the Patron options to register now and continue placing an order, to place an order for pickup in the cafeteria, or to exit from the COS. The system shall prompt the Patron for the meal date. If the meal date is the current date and the current time is after the order cutoff time, the system shall inform the patron that it's too late to place an order for today. The Patron may either change the meal date or cancel the order. The Patron shall specify whether the order is to be picked up or delivered. If the order is to be delivered and there are still available delivery times for the meal date, the Patron shall provide a valid delivery location.

Actors:
- Patron
- Cafeteria Ordering System

Use-cases:
- Place an order
- Register for payroll deduction
- Cancel an order
- Change the meal date
- Specify order pickup or delivery
- Provide a delivery location

Figure 8
Actor and use-case recognition using GPT-3

4.2.2.2 Design pattern recognition and classification

In the design phase of software development, it is important to accurately identify and classify design patterns to effectively structure and organize the software. Design patterns are reusable solutions to common design problems, and they can help to improve the flexibility, maintainability, and scalability of the software.

Large language models can play a valuable role in the recognition and classification of design patterns by using natural language processing techniques to analyze and understand written text, including design documentation and source code. These models can identify the specific design patterns employed in the software and classify them based on their characteristics and features.

Below are some concrete examples of how large language models can support design pattern recognition and classification:

- Identifying design patterns in documentation

  A large language model can analyze design documentation, such as requirement documents and design specifications, to identify the design patterns that are used in the software. For example, if the documentation describes the use of a factory pattern to create objects, the language model could recognize this and classify it as a factory pattern.

- Identifying design patterns in source code

  The model can also analyze source code to identify and classify design patterns. For example, if the source code includes the use of a decorator pattern to add new functionality to an object, the model could recognize this and classify it as a decorator pattern.

- Classifying design patterns

  In addition to identifying the specific design patterns that are used in the software, the model can also classify the patterns based on their characteristics and features. For example, the model could classify a factory pattern as a creational pattern, which deals with the creation of objects, or a decorator pattern as a structural pattern, which deals with the composition of objects.

- Providing context and explanations

The model can also provide context and explanations for the identified design patterns, such as the benefits and drawbacks of each pattern and the situations in which they are most appropriate. This can help software developers to understand and effectively apply the design patterns in their work.

Overall, large language models can play a valuable role in the recognition and classification of design patterns in the design phase of software development. By using NLP techniques to analyze written text and source code, these models can accurately identify and classify the design patterns employed in the software, providing valuable context and explanations for software developers.

We evaluated GPT-3 for the task of design pattern recognition and classification. Figure 9 below shows, that given a set of problem descriptions, how GPT-3 can correctly identify the design pattern applicable to a given statement.

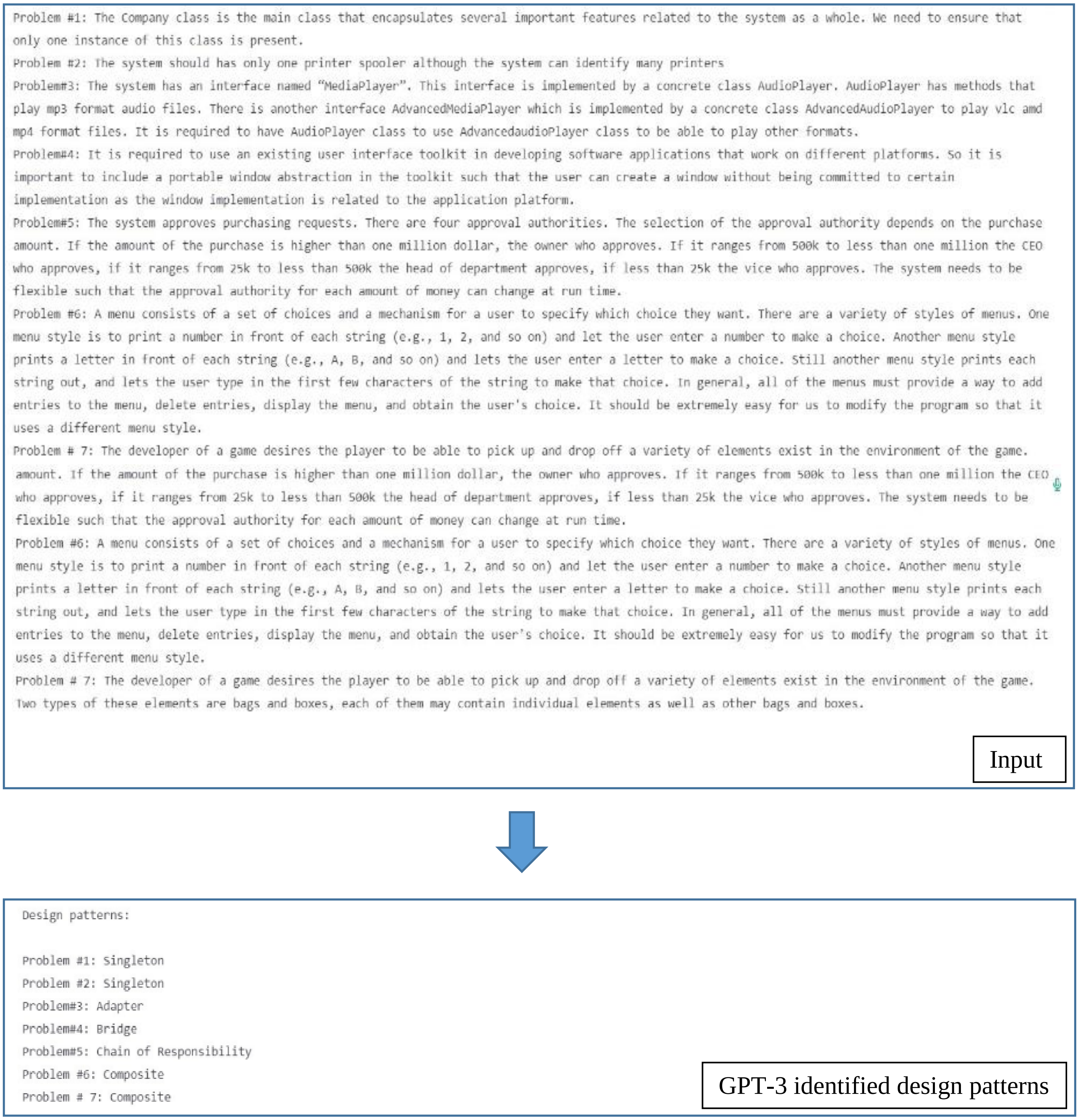


Figure 9

Design pattern recognition from problem description

Similarly, Figure 10 shows how a model like GPT-3 can be used to identify the correct design pattern in code.

```
Code:
public abstract class Game {
  abstract void initialize();
  abstract void startPlay();
  abstract void endPlay();

  public final void play(){

    initialize();

    startPlay();

    endPlay();
  }
}

public class Cricket extends Game {

  @Override
  void endPlay() {
    System.out.println("Cricket Game Finished!");
  }

  @Override
  void initialize() {
    System.out.println("Cricket Game Initialized! Start playing.");
  }

  @Override
  void startPlay() {
    System.out.println("Cricket Game Started. Enjoy the game!");
  }
}

public class Football extends Game {

  @Override
  void endPlay() {
    System.out.println("Football Game Finished!");
  }

  @Override
  void initialize() {
    System.out.println("Football Game Initialized! Start playing.");
  }

  @Override
  void startPlay() {
    System.out.println("Football Game Started. Enjoy the game!");
  }
}
```

Input

Design pattern: Template Method Pattern

Reasons: The Template Method Pattern is a behavioral design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses. The code example provided follows this pattern as the abstract class Game defines the basic algorithm (the play() method) and the subclasses (Cricket and Football) can override the methods initialize(), startPlay(), and endPlay() to provide the specific implementation of each step.

Design pattern recognized using GPT-3

Figure 10

Design pattern recognition in code

design pattern extraction on the code fragments from the book Design Patterns: Elements of reusable object-oriented software (Gamma et al., 1994). In the next section, we present our approach and results on the same.

4.2.2.3 Class extraction for given problem statement

Classes are a fundamental concept in object-oriented programming, and they represent the entities that make up the software system. In the design phase of software development, it is important to accurately understand and model these classes. Class extraction involves identifying and defining the classes that are needed to achieve the desired behavior and structure of the software, as specified in the requirement documents.

Large language models can play a valuable role in class extraction by using natural language processing (NLP) techniques to analyze the text of the requirement documents and identify relevant information. These models can extract key nouns and verbs from the requirements and use them to define the classes and their characteristics and behaviors. For example, consider the following requirement:

"*The software should be able to search for a particular product by name and display a list of matching products*."

A large language model can identify the key nouns "software," "product," and "list" and the verb "search," and use this information to define the classes in the software. The model might suggest defining a "Product" class with a "search" method, a "ProductList" class to represent the list of matching products, and a "Software" class to represent the overall software system.

In addition to extracting information about the classes, the model can also provide context and explanations for the identified classes and their characteristics and behaviors. For example, it might provide information about the attributes and methods that should be included in each class and the relationships between the classes.

We tested the capability of the GPT-3 model to perform this task of class extraction. Figure 11 below, shows how the model is successfully able to identify the classes, the attributes, the methods, and the relationships between the classes, given a problem statement.

```
Problem Statement:
A library issues loan items to customers. Each customer is known as a member and is issued a membership card that shows a unique member number. Along
with the membership number, other details on a customer must be kept such as a name, address, and date of birth. The library is made up of a number of
subject sections. Each section is denoted by a classification mark. A loan item is uniquely identified by a bar code. There are two types of loan items,
language tapes, and books. A language tape has a title language (e.g.French), and level (e.g.beginner). A book has a title, and author(s). A customer may
borrow up to a maximum of 8 items. An item can be borrowed, reserved or renewed to extend a current loan. When an item is issued the customer's
membership number is
scanned via a bar code reader or entered manually. If the membership is still valid and the number of items on loan less than 8, the book bar code is
read, either via the bar code reader or entered manually. If the item can be issued (e.g., not reserved) the item is stamped and then issued. The library
must support the facility for an item to be searched and for a daily update of records.
```

```
Classes:

Customer
Member
LoanItem
LanguageTape
Book
Section

Attributes:

Customer: name, address, date of birth
Member: member number
LoanItem: bar code
LanguageTape: title, language, level
Book: title, author
Section: classification mark

Methods:

Customer: getName(), getAddress(), getDateOfBirth()
Member: getMemberNumber()
LoanItem: getBarCode()
LanguageTape: getTitle(), getLanguage(), getLevel()
Book: getTitle(), getAuthor()
Section: getClassificationMark()

Relationships:

Customer has a Member
LoanItem has a LanguageTape
```

GPT-3 generated classes, methods, and relationships

Figure 11
Class extraction from problem statement using GPT-3

Machine learning algorithms can be applied to create custom domain-specific website designs for the user. This can save a lot of manual effort. AiDA (Artificial intelligence design assistant) makes use of proprietary machine learning algorithms to create custom domain-specific website designs for the user – saving a lot of effort. Based on the user inputs and requirements, the AI assistant provides the best suggestions to the user for their website.

Similarly, large pre-trained models like GPT-3 can be utilized to generate template design code for a given requirement. For example, consider a requirement below:

"A user wants to create a new website for an online store that sells eco-friendly products. The user provides some high-level descriptions and design requirements such as:

The website should have a clean and modern design. The color scheme should include green and white to reflect the eco-friendly theme. The website should have a prominent header section with a logo, navigation menu, and search bar. The website should have a hero section that showcases some of the featured products. The website should have a section for displaying categories of products. The website should have a footer section with links to social media and other relevant pages. Using these high-level descriptions and design requirements, generate a website design that meets the user's needs"

For this requirement, the machine generated code is added in the appendix. Figure 12 below shows the output generated by the code:

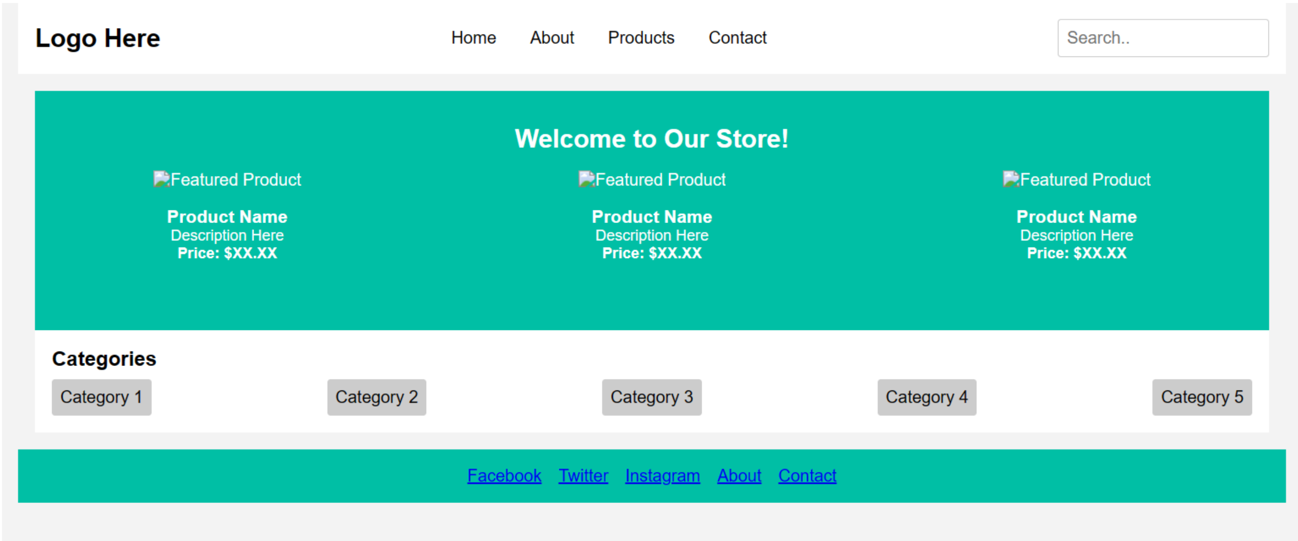


Figure 12
Eco-Friendly website

Similarly, Figure 13 below shows the output produced by the code generated by gpt3.5-turbo (ChatGPT)

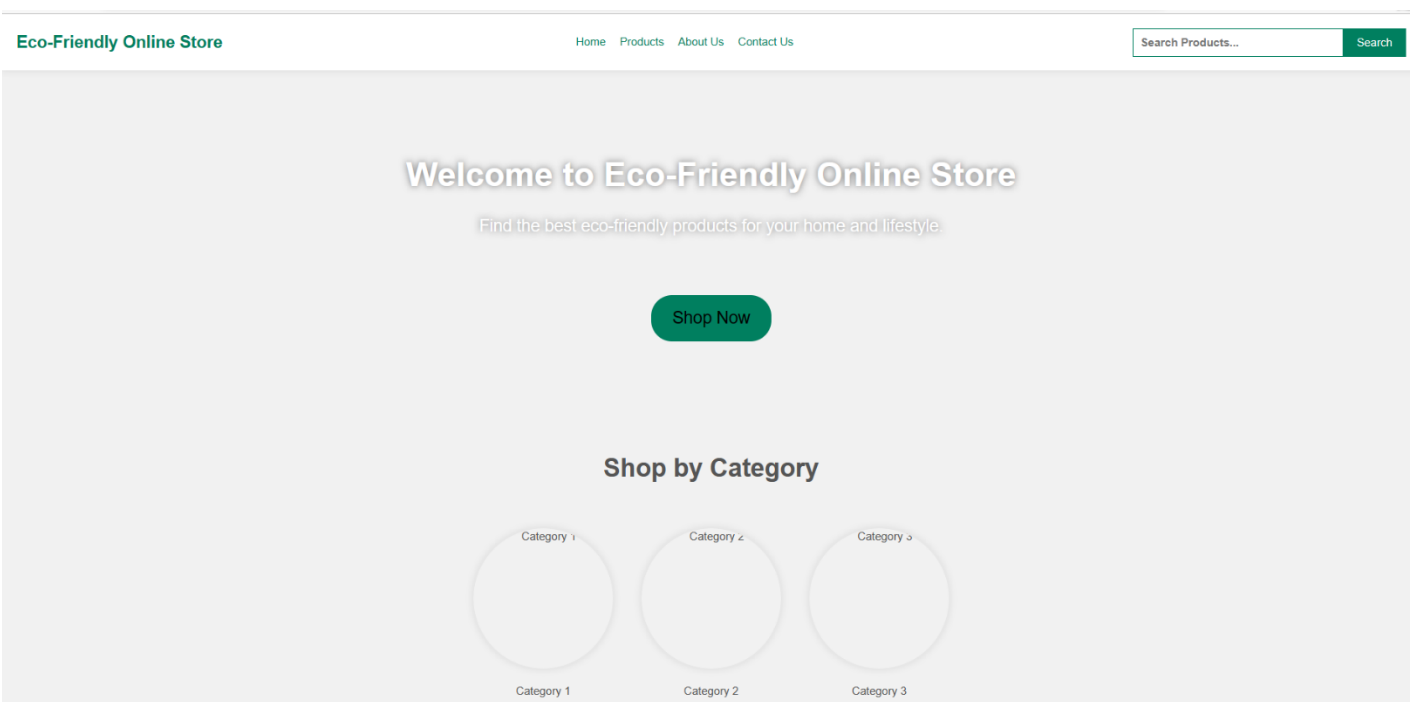


Figure 13
Eco-friendly website (ChatGPT)

4.2.3 Coding and Testing

Coding is the process of translating the design into a computer program. The code is written in a programming language and is typically organized into modules or classes. Testing ensures that the code compiles and meets the requirements. This is among the most crucial phases of software development, as errors in the code can lead to incorrect results. Also, there is a huge dependency on the coding standards followed by the team and the testing team. There are multiple tools available to help with coding and testing, like code coverage tools, static analysis tools, and so on. AI can help in automating the process of writing code as well as in testing the code.

4.2.3.1 Code completion and suggestion

Code completion, also known as statement completion or auto-completion, is a software feature that suggests possible syntax or code snippets as the user is typing, based on previous inputs or a predefined list of suggestions. Code suggestion, on the other hand, is a feature that suggests possible functions or method names based on the context of the code. Large pre-trained transformer models have been used to improve the accuracy and efficiency of code completion and suggestion systems. Large volumes of source code and text written in natural language can be processed and analyzed by these models, which then use this knowledge to provide relevant suggestions to the user.

Code completion is a feature that helps software developers by automatically completing the code they are writing. Large language models can be used to implement code completion systems that can suggest code snippets and complete entire code blocks. For example, a code completion system based on a large language model can suggest the correct code snippets for a given programming task, such as creating a database connection or implementing a specific algorithm. The system can also generate entire code blocks for common software components, such as error-handling routines or database access layers.

One example of a code completion tool that uses a large language model is Deep TabNine. This tool uses a language model based transformer architecture and trained on a dataset of more than two million GitHub repositories and can provide accurate code completion suggestions for a wide range of programming languages. Another example is the code suggestion feature in the PyCharm Integrated Development Environment (IDE). PyCharm uses a language model trained on a dataset of Python code to suggest function and method names as the user is typing. This can save time and improve the readability of the code by suggesting appropriate names for variables and functions.

Large language models have also been leveraged to enhance the precision of error detection and correction in code. For example, the language model-based tool Kite uses machine learning to analyze the context of the code and suggest corrections for syntax errors. Similarly, Figure 14 shows the capability of Codex to complete a piece of incomplete code.

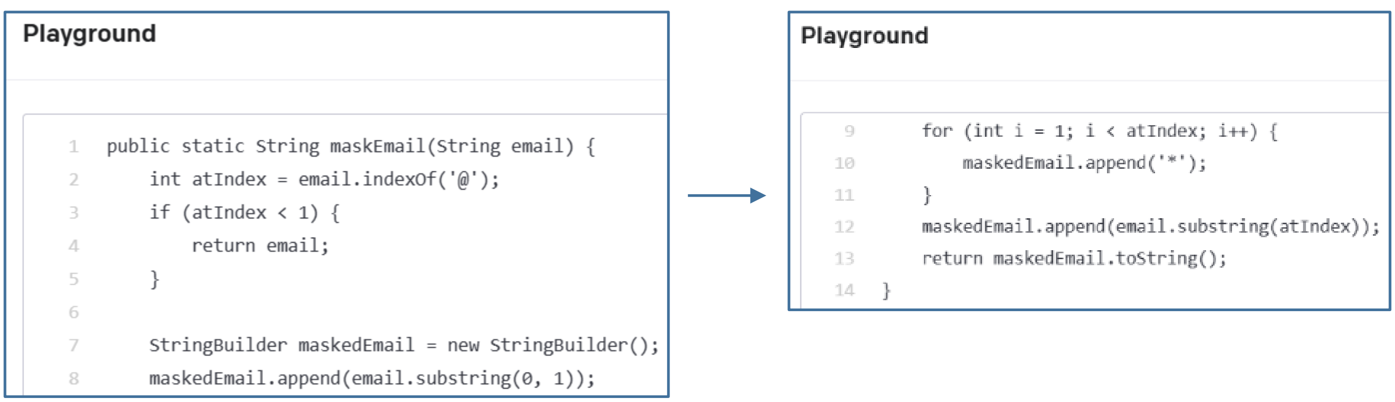


Figure 14
Code completion

In addition to the code-specific applications, these large transformer models have also been used to enhance the accuracy of natural language processing tasks in code documentation and comments. For example, a language model could be used to suggest appropriate tags for documentation or to generate summaries of code functionality based on comments.

Overall, the use of LLMs in code completion and suggestion has the potential to improve the efficiency and accuracy of coding tasks and to make programming more accessible to a wider range of users. In the next section, we present a detailed analysis and comparison of different models for performing this task.

4.2.3.2 Code generation

Code generation denotes the ability of large machine learning models to generate code based on instructions in natural language. This can be a useful tool for software developers, as it can help to automate the task of writing code. The use of LLMs in code generation has recently become very popular, since these LLMs can generate code that is not only correct but also easily understandable and maintainable. The following are some examples of how large language models are being used in code generation.

- Code Templates

  Code templates are pre-written code snippets that can be used to generate code quickly and easily. Large language models can be trained to generate code templates based on natural language instructions. For example, a software developer could describe a specific software component and the language model would generate a code template that implements that component. For example, Figure 15 below shows the template code generated by Codex for a class in Python that implements a stack data structure. This code implements a basic stack data structure in Python, with methods for pushing items onto the stack, popping items off the stack, checking if the stack is empty, and returning the size of the stack.

Playground

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

Template code generated using GPT-3

Figure 15
Template code generation

- Code Refactoring

Code refactoring involves making design changes to existing code without affecting its functionality. Pre-trained language models can be employed to automate the code refactoring process by identifying patterns in the existing code and generating new code snippets that are more efficient, readable, and maintainable. For example, a code refactoring system based on a large language model can analyze an existing codebase and generate new code snippets that follow modern programming practices and best coding standards. The system can also suggest code changes that will improve performance, reduce complexity, and enhance code quality. Figure 16 below shows an example of code refactored to improve performance.

```
def sum_of_cubes(numbers):
    sum = 0
    for num in numbers:
        sum += num ** 3
    return sum
```

Input

```
Refactored code:

def sum_of_cubes(numbers):
    n = len(numbers)
    return (n * (n + 1) // 2) ** 2
```

Refactored code using GPT-3

Figure 16
Code Refactoring

This code implements a function named ‘sum_of_cubes’ which accepts a list of numbers as input and computes a sum of the cubes of those numbers. However, this code can be improved by using a more efficient algorithm. One such algorithm is using the formula $[n(n + 1)/2]^2$ to calculate the sum of cubes of the first n natural numbers.

- API Generation

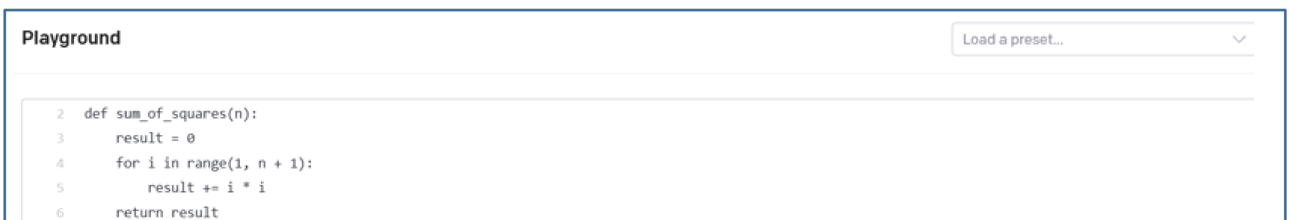
One of the prevalent use-cases of large transformer models in code generation is the generation of APIs. An API, short for Application Programming Interface is a collection of protocols, procedures, and tools used in the development of software and applications. An API specification contains information about the functions, data types, and structures that a software component provides. LLMs can be leveraged to generate code for APIs based on the input API specification. For example, a language model could generate code in multiple programming languages, such as Python, Java, and C++, for both the server-side and client-side components of an API.

This shows that as technology continues to advance, it is likely that large language models will play an increasingly important role in the future of software engineering. The results of different large language models on the tasks described above, are presented in the next section.

4.2.3.3 Code summarization and documentation

As the team of developers working on a piece of code changes over time, it becomes difficult for new resources to fully comprehend code that was written years ago. Code summarization involves creating a concise, meaningful summary of code by capturing its key features and functionality. Large language models can process the code, identify its key components, and generate a brief and informative summary. Figure 17 below shows the capability of a large model to summarize code.

Code documentation, like code summarization, is an important aspect of software development, as it helps developers understand the code and use it effectively. Large language models can generate documentation for given piece of code, making it easier for developers to understand code.



Input

```
Summary:

The function sum_of_squares takes a positive integer n as input and returns the sum of the squares of the first n positive integers.

The function uses a variable result to keep track of the sum of the squares. The function starts with result set to 0.

The function uses a for loop to iterate over the integers from 1 to n. On each iteration, the function adds the square of the current value of i to result.

After the loop completes, the function returns result.

The function is called with the value 5. The function returns 55.

The function is called with the value 10. The function returns 385.
```

GPT-3 Generated Code summary

Figure 17
Code summarization

Here are a few examples of how these models are used in code documentation.

- Auto-Generated Documentation

  Large language models can be leveraged to generate code documentation automatically, by analyzing the code and extracting relevant information. This helps developers understand the code easily, as they do not have to manually go through the code and create documentation.

```
import java.util.Scanner;
public class Commissions
{
  public void printCommision(int unitsSold)
  {
    if (unitsSold >= 0 && unitsSold <= 10)
    {
      System.out.println("Commission is 10% and this is under the quota.");
    }
    else if (unitsSold >= 11 && unitsSold <= 30)
    {
      System.out.println("Commission is 15% and this meets the quota.");
    }
    else if (unitsSold >= 31 && unitsSold <= 99)
    {
      System.out.println("Commission is 20% and this is over the quota.");
    }
    else
    {
      System.out.println("This is the default");
    }
  }

    public static void main(String[] args)
    {
      Scanner keyboard = new Scanner(System.in);
      int unitsSold;
      System.out.print("Enter the number of units sold: ");
      unitsSold = keyboard.nextInt();
      printCommision(unitsSold);
    }
  }
```

Input given to GPT-3

```
Code with documentation:
  import java.util.Scanner;
  /**
  * This program will calculate the commission of a salesperson based on the number of units sold.
  *
  * @author (your name)
  * @version (a version number or a date)
  */
```

Inline documentation generated using GPT-3

```
  public class Commissions
  {
     /**
     * This method will print the commission of a salesperson based on the number of units sold.
     *
     * @param unitsSold The number of units sold by the salesperson.
     */
```

Inline documentation generated using GPT-3

```
     public void printCommision(int unitsSold)
     {
        if (unitsSold >= 0 && unitsSold <= 10)
        {
           System.out.println("Commission is 10% and this is under the quota.");
        }
        else if (unitsSold >= 11 && unitsSold <= 30)
        {
           System.out.println("Commission is 15% and this meets the quota.");
        }
        else if (unitsSold >= 31 && unitsSold <= 99)
        {
           System.out.println("Commission is 20% and this is over the quota.");
        }
        else
        {
           System.out.println("This is the default");
        }
     }

     /**
     * This is the main method.
     *
     * @param args The command line arguments.
     */
```

Inline documentation generated using GPT-3

```
     public static void main(String[] args)
     {
        Scanner keyboard = new Scanner(System.in);
        int unitsSold;
        System.out.print("Enter the number of units sold: ");
        unitsSold = keyboard.nextInt();
        printCommision(unitsSold);
     }
  }
*/
```

Figure 18
Code Documentation

- Code Annotation

  Another important use of large language models in code documentation is code annotation. These models can add comments to the code, explaining its functionality and purpose. This helps developers easily comprehend the code since they can review the comments as they go through the code.

Playground

```
def get_even_numbers(n):
    even_numbers = []
    for i in range(2, n + 1, 2):
        even_numbers.append(i)
    return even_numbers
```

```
Code with annotations and documentation:

def get_even_numbers(n):
    """
    This function returns a list of even numbers up to n
    """
    # Initialize an empty list to store even numbers
    even_numbers = []
    # Loop through the numbers from 2 to n, incrementing by 2 to get only even numbers
    for i in range(2, n + 1, 2):
        # Add the current number to the list of even numbers
        even_numbers.append(i)
    # Return the list of even numbers
    return even_numbers
```

Figure 19
Code annotation

- Code Examples

  Large ML models can also be leveraged to generate code examples, to help developers understand how to use the code effectively. These models can analyze the code and generate examples that demonstrate how the code can be used in different scenarios.

4.2.3.4 Code translation

As technology evolves rapidly, the programming languages that are currently relevant may become obsolete in a few years. This makes the code difficult to maintain, and it may need to be migrated or translated to a newer language. Code translation involves converting a piece of code from one programming language to another. This can be challenging since it demands an understanding of the syntax, semantics, and structure of both target as well as source language, as well as the underlying logic and functionality of the code.

Leveraging large language models for code translation has made this task easier and more accurate. The models have learned the patterns and structures of multiple programming languages since they were trained on a vast amount of code. They can then use this knowledge to translate the code into a new language while preserving its functionality and logic. Here are a few examples of how large language models are being used in code translation:

- Source-to-Target Translation

  This type of code translation involves converting source code written in one language into another language. For example, translating Python code into Java code. In this case, the model needs to understand the syntax, structure, and semantics of both languages to ensure a successful translation.

- Framework Conversion

  This type of code translation involves converting code written for one framework into another framework. For example, converting code written for the Django framework in Python into the Ruby on Rails framework. In this case, the model needs to understand the specific syntax, structure, and functionality of each framework to ensure the translated code works as expected.

- Legacy Code Migration

This type of code translation involves converting old, outdated code into modern, up-to-date code. For example, converting code written in COBOL into Java. In this case, the model must comprehend the syntax, structure, and functionality of both the old and new languages to ensure the translated code works as expected.

These are just a few examples of how large language models are being used in code translation. The use of these models has made the task of code translation faster, more accurate, and more efficient, enabling developers to concentrate on other areas of the development process. It is noteworthy to highlight that while large pre-trained models have the potential to greatly aid in code translation, they are not a silver bullet solution. The generated code still needs to be reviewed and tested by a human developer to ensure that it meets the desired specifications and standards. While these models are not a complete solution, they can save time and effort for developers and make code translation a more efficient and streamlined process.

4.2.3.5 Code review

Code review is the process of examining the code written by a software developer for errors, bugs, and potential improvements. It is a necessary step in the process of software development since it ensures that the code adheres to the desired levels of quality and is ready to be deployed. However, when done manually, this can be a tedious and time-consuming process.

AI-based models such as GPT-3 and Codex can play a significant role. By utilizing the strength of Deep Learning and Natural Language Processing, GPT-3 can assist in code review in several ways:

- Automated Bug Detection

GPT-3 can be trained to identify common bugs and security vulnerabilities in code. For example, if the code has an unhandled exception, GPT-3 can detect it and suggest an appropriate solution.

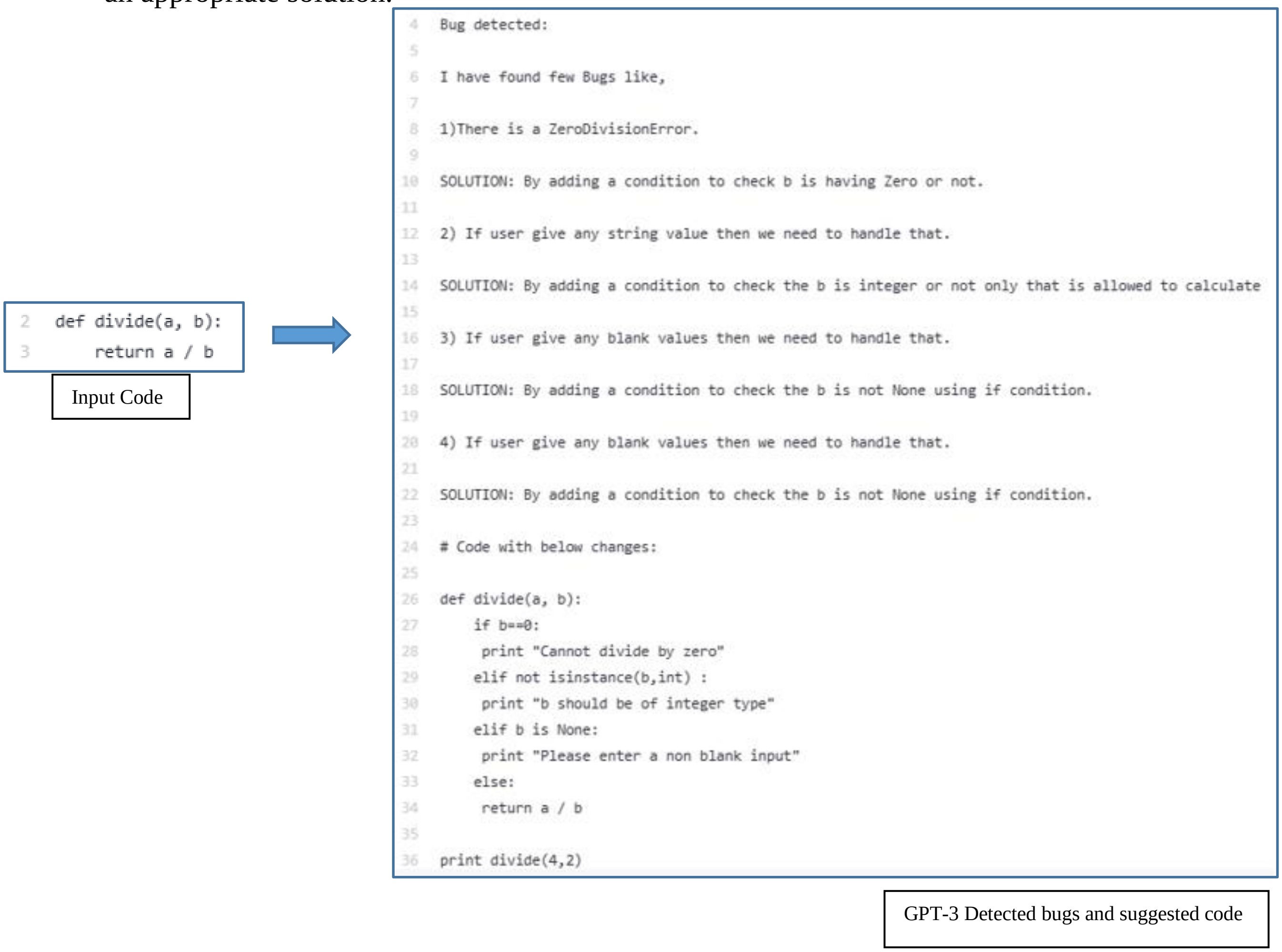


Figure 20
Bug Detection

- Code Style and Formatting

Large language models like Codex can assist in ensuring that the code follows the desired coding standards and conventions. For example, if the code is not formatted according to the company's coding standards, Codex can suggest appropriate changes.

```
def hello(name):
print("Hello, "+name+"!")

people = ["Alice","Bob","Charlie"]

for person in people:
hello(person)
```

Input

```
Code conforming to PEP8 standards:

def hello(name):
   print("Hello, " + name + "!")

people = ["Alice", "Bob", "Charlie"]

for person in people:
   hello(person)

PEP8 violations:
- The function name is not in lowercase with words separated by underscore (snake_case).
- There is no space after the comma in the print statement.
- The list of people should have a space after each comma.
- There is no indentation for the print statement within the hello function.
- There is no indentation for the for loop.
```

GPT-3 suggested PEP8 violations and code

Figure 21
Code style and formatting

- Code Optimization and suggesting alternatives

  A large language model can help identify areas of the code that can be optimized for better performance. For example, if a particular piece of code is taking too long to execute, LLM like Codex can suggest alternative solutions that can improve the performance. If a developer is using an inefficient algorithm, Codex can suggest a more efficient one. Figure 22 below shows one such example.

```
def sort(arr):
    n = len(arr)
    # optimize code, so if the array is already sorted, it doesn't need
    # to go through the entire process
    swapped = False
    # Traverse through all array elements
    for i in range(n-1):
        # range(n) also work but outer loop will
        # repeat one time more than needed.
        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            # if we haven't needed to make a single swap, we
            # can just exit the main loop.
            return
```

Input

```
Optimal code:
 def sort(arr):
     if len(arr) <= 1:
         return arr
     pivot = arr[len(arr) // 2]
     left = [x for x in arr if x < pivot]
     middle = [x for x in arr if x == pivot]
     right = [x for x in arr if x > pivot]
     return sort(left) + middle + sort(right)

 print(sort(arr))
```

GPT-3 generated optimal code

Figure 22
Code optimization

- Duplicate Code Detection

  Large language models can assist in identifying duplicated code within a codebase, making it easier to eliminate and maintain the codebase's readability and maintainability. Thus, pre-trained transformer models such as Codex and GPT-3 have the potential to revolutionize the code review process. By automating some of the manual tasks involved in code review, these models can help software developers to save time, improve the quality of their code, and make the software development process more efficient. While there is still room for improvement, the future looks bright for AI-based models in code review, and it will be interesting to see how they evolve and become even more useful in the years to come.

4.2.3.6 Test case generation

Test case generation is the process of automatically generating test cases for a piece of code. To test the functionality of the code, test cases need to be generated. This is a time-consuming process that can be automated using AI-based models. For example, given a piece of code that implements a function, the model can generate a set of inputs that would exercise different scenarios and edge cases of the function. The model can also generate assertions or expected outputs for each test case, making the test cases self-verifying as shown in Figure 23 below.

Similarly, large language models can be used for test-driven development (TDD). Developing automated tests before writing code is a key component of the test-driven development technique. The purpose of TDD is to make sure that the code meets the required specifications and that the software behaves as expected. Large language models can be leveraged in TDD to generate test cases and validate the code's behavior.

```
def find_largest_number(numbers):
    largest = None
    for num in numbers:
        if largest is None or num > largest:
            largest = num
    return largest
```

Input

```
Test case 1:
numbers = [1, 2, 3, 4]
Expected output: 4

Test case 2:
numbers = []
Expected output: None

Test case 3:
numbers = [0, 0, 0]
Expected output: 0

Test case 4:
numbers = [1, -2, 3]
Expected output: 3

Test case 5:
numbers = [2.5, 3.2, 1.9]
Expected output: 3.2
```

GPT-3 generated test-cases

Figure 23
Test case generation

4.3 Research Question Two

How do large language models' code automation capabilities compare against traditional code generation?

Our first hypothesis states that "The code generation/translation capabilities of large language models are better than traditional code generation and rule-based tools". We extend this hypothesis to different phases of the software development life cycle.

In our findings to RQ1, we found several tasks where AI-based large language models can help automate different tasks. In this question, we aim to compare how these models fare when compared to the traditional techniques most widely used to perform the same task.

4.3.1 Planning and requirements gathering

Software requirement classification

Software development teams must learn about the needs and requirements of their potential clients and users before launching a new product. This process is called requirements elicitation and can take a long time to complete. A software requirements specification (SRS) document, which outlines and defines all of the criteria that must be satisfied for the product to be considered complete, is the final result of the requirements elicitation process. The requirements elicitation method takes into account a variety of requirements, including functional requirements (FRs) and non-functional requirements (NFRs). As opposed to NFRs, which evaluate criteria like performance, scalability, and use cases, FRs describe the specific behaviors and functionality of the product.

The SRS is more transparent and well-organized when requirements are classified correctly. However, manual classification can be both time- and resource- consuming and can lead to teams inaccurately assessing the non-functional requirements. Automating requirements classification might reduce the demand for subject-matter specialists and enable the software industry in adopting this technique.

However, it is still difficult to automatically categorize requirements stated in natural language into both functional as well as non-functional and also the subcategories of non-functional requirements (Núñez, no date). This is even though software requirements are well understood and thoroughly documented. This is particularly because stakeholders and requirements engineers define the same type of demand using various terminology and grammar, according to Abad et al., 2017. Hence, finding the best techniques to implement an accurate automated classification is a challenge.

We have seen in the discussion about RQ1, that large language models support the task of requirements classification. Here, we do a detailed study on the same and compare their performance to other tools and techniques used for the same.

Dataset:

We used the PROMISE NFR dataset. The dataset consists of a set of functional and non-functional requirements for a range of software products. The dataset has a total of 624 requirements. Figure 24 below shows the distribution of functional vs non-functional requirements.

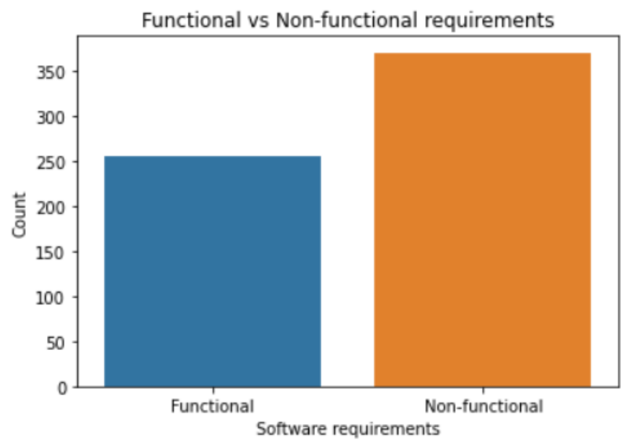


Figure 24
Distribution of functional vs non-functional requirements

The non-functional requirements further belong to different types like Usability (US); Security (SE); Scalability (SC); Portability (PO); Performance (PE); Operational (O); Maintainability (MN); Look and Feel (LF); Legal (L); Fault Tolerance (FT); Availability (A)

Table 5 below shows the distribution of each of these NFRs in the dataset.

Table 5
Non-functional requirements

Sr. No.	**NFR**	**Count**
1	A	21
2	FT	10
3	L	13

4	LF	38
5	MN	17
6	O	62
7	PE	54
8	PO	1
9	SC	21
10	SE	66
11	US	67

The same is depicted in the graph below.

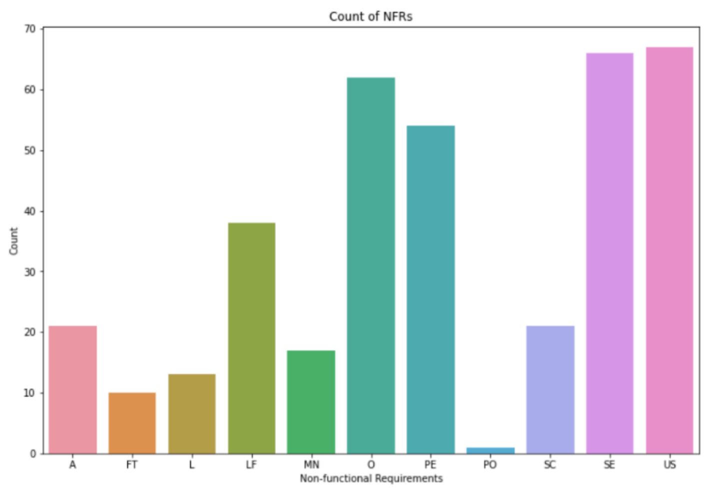


Figure 25
Distribution of non-functional requirements

Task

The evaluation is carried out on two tasks related to requirement classification:

1. Binary classification – classify a given requirement into functional vs non-functional
2. Multi-class classification – for the set of non-functional requirements, classify the requirement into one of the eleven categories mentioned above.

Manual techniques

Prior to the application of machine learning techniques to the task of categorizing software requirements, this task was majorly done manually. As mentioned earlier, manual classification is error-prone, resource-intensive, and time-consuming. There is a dependency on domain experts to correctly identify and capture the non-functional requirements. At the same time, the availability of domain experts has always remained a challenge. Daniel Ott (Ott, 2013) explored the categorization of requirement specifications for the automotive sector. He evaluated fully-automated, semi-automatic, and manual methods for requirements classification in an industrial context with a team of ten practitioners. According to his findings, a semi-automatic technique provides the best quality-to-effort ratio and the best learning outcomes.

To develop a comparison of how traditional manual techniques fare against large language models, the task of requirements classification is done manually. As discussed in the research design section, this task is assigned to three candidates with different levels of competence - expert, intermediate, and novice. These candidates, based on their expertise, domain knowledge, and prior experience manually classify the requirements for both binary classification as well as multi-class classification.

The following are the results obtained:

Binary classification

Table 6
Binary Classification - Results

Expertise Level	**# of correctly classified requirement**	**Total # of requirements**	**% Accuracy**
Novice	439	625	70
Intermediate	506	625	81
Expert	531	625	85

Multi-class classification

Table 7
Multi-class Classification - Results

Expertise Level	**# of correctly classified requirement**	**Total # of requirements**	**% Accuracy**
Novice	229	370	62
Intermediate	263	370	71
Expert	296	370	80

Traditional Machine Learning

As machine learning algorithms have become more sophisticated, researchers have begun to apply them to the task of automatically classifying software requirements. Dias Canedo and Cordeiro Mendes, 2020 discuss how machine learning methods can be used to perform this task effectively. The authors used PROMISE dataset as their primary data source. They applied different pre-processing and cleaning techniques and BoW, TF-IDF, and CHI2 were used, respectively, for feature extraction and feature selection. Logistic Regression (LR), k-Nearest Neighbors (kNN), Support Vector Machine (SVM) and Multinomial Naive Bayes (MNB) were different algorithms they used for classifying the requirements. The study found that employing TF-IDF followed by LR was more efficient with F-measure scores in binary classification, NF classification and general classification of nearly 0.9 (SVM gave the same score), 0.74 and 0.78 respectively.

Similarly, Navarro-Almanza, Juarez-Ramirez and Licea, 2017 apply a CNN-based deep learning approach for the task of multi-class classification of software requirements. With their approach, they got average precision, recall, and F-measure scores of 0.80, 0.79 and 0.77 respectively. In another study, Kurtanovic and Maalej, 2017 evaluates different supervised classifiers for automatic classification of requirements as functional / non-functional, with a focus on different types of NFRs. They found that the most informative features were part of speech tags with the cardinal number (POS tag CD) being the single most informative feature. In their study, they find that with manual selection of features followed by different pre-processing techniques, they achieve precision and recall of almost 92%. By simply using word features and no feature selection, they achieved precision and recall rates of 70% to 90% for classifying NFRs.

Supervised classification techniques require a substantial volume of labeled data. Casamayor, Godoy and Campo, 2010 employed semi-supervised learning techniques for NFR classification and demonstrated that this approach performs better as compared to the supervised classification techniques. Similarly, Fong, 2019 presents his findings on the use of word embeddings and convolutional neural networks for classifying software requirements. All the above techniques involve training a custom model using different ML algorithms. Since there are multiple studies and metrics which already suggest the accuracy of the various techniques and models, we do not repeat the same exercise here.

As mentioned in the above sections, large language models like GPT-3 and BERT are already pre-trained on large datasets and they are already capable of performing the task of classifying text. We explored these models to answer RQ1 for the specific task of software requirement classification.

Large language models

We observed during RQ1, the capability of GPT-3 to perform the task of software requirements classification using zero-shot prompt engineering. We now evaluate it further on the PROMISE NFR dataset.

Approach:

We experiment with GPT-3 on the same two tasks related to requirement classification, as done in the manual section.

Dataset:

The dataset that we have is small with a total of only 624 requirements. Traditional ML-model which are built using techniques like word2vec embeddings generally require larger datasets to avoid overfitting and for the model to perform with decent accuracy. However, since models like GPT-3 are already pre-trained, such models can leverage their in-built knowledge and work well with small datasets too. In the next section, we tested the performance of GPT-3 on both of the above tasks.

Approach 1 – Zero-shot

Binary classification

The first approach we tried was using zero-shot learning. Here, we provide a set of instructions that covers the larger context, for the model to understand the task it is required to perform. This is called as prompt-engineering. When we have a good prompt, which we first test on a small number of requirements, we then use this prompt to classify a larger set of requirements. With this approach, GPT-3 model gave an accuracy of 81% on the task of binary classification. Figure 26 below shows the precision and recall values for functional (1.0) and non-functional (0.0) requirements.

	precision	recall	f1-score	support
0.0	0.88	0.79	0.83	370
1.0	0.74	0.84	0.78	255
accuracy			0.81	625
macro avg	0.81	0.82	0.81	625
weighted avg	0.82	0.81	0.81	625

Figure 26
Classification report – binary classification (zero-shot)

Multi-class classification

Similarly, we prepare a prompt that works well on a set of requirements, for categorizing the non-functional requirements into one of the eleven categories. We then evaluate the model accuracy on classifying larger set of requirements using zero-shot learning. GPT-3 model gave an accuracy of 71% on the task of multi-class classification.

Figure 27 below shows the recall and precision values for various classes using this approach.

	precision	recall	f1-score	support
A	0.61	0.90	0.73	21
FT	0.57	0.40	0.47	10
L	0.50	0.69	0.58	13
LF	0.89	0.87	0.88	38
MN	0.30	0.47	0.36	17
O	0.70	0.48	0.57	62
PE	0.79	0.81	0.80	54
PO	0.05	1.00	0.09	1
SC	0.69	0.43	0.53	21
SE	0.98	0.83	0.90	66
US	0.87	0.78	0.82	67
accuracy			0.71	370
macro avg	0.63	0.70	0.61	370
weighted avg	0.78	0.71	0.73	370

Figure 27
Classification report - multiclass classification (zero-shot)

Approach 2 – Embeddings based classification

Binary classification

In the second approach, we use GPT-3 embeddings and build a classifier model using it. Since GPT-3 is a massive 175-billion-parameter model, its embeddings are very powerful as they capture complex relationships between words, phrases, and sentences. We use the text-similarity-davinci-001 model to generate the embeddings for the requirements. These embeddings are very high in dimension (12288 dimensions).

We tried different algorithms, performed cross-validation, and applied hyperparameter tuning to arrive at the best-performing model. The results are shown below.

Table 8
Results from different experiments

Sr. No.	Model	Accuracy	Precision (1)	Recall (1)	Precision (0)	Recall (0)
1	Random forest	88	0.85	0.81	0.89	0.92
2	XG Boost	91	0.84	0.92	0.96	0.90
3	XGB Random Forest	83	0.76	0.81	0.88	0.85
4	**Random forest (hyper-tuned)**	93	0.87	0.92	0.96	0.93

As observed above, for binary classification, random forest classifier gave the best accuracy of 92%.

Table 9
Results from Random-Forest Classifier

	Precision	**Recall**	**F1-score**	**Support**
0	0.96	0.92	0.94	122
1	0.86	0.92	0.89	66
Accuracy			0.92	188
Macro avg	0.91	0.92	0.91	188
Weighted avg	0.92	0.92	0.92	188

Multiclass-classification

Similarly, experiments were performed for multi-class classification using an embeddings-based approach. We trained different models for this task using different algorithms. The dataset for the non-functional requirements is imbalanced. First, we trained the models without applying any class-balancing techniques. The performance of the trained models was observed and the metrics were recorded on the test dataset. The results are as shown below:

A. OneVsRestClassifier (SVC)

Table 10
OneVsRest Classifier (NFR classification)

	precision	recall	f1-score	support
0	0.89	1.00	0.94	8
1	0.00	0.00	0.00	3
2	0.50	1.00	0.67	1
3	0.86	0.60	0.71	10
4	0.00	0.00	0.00	3
5	0.65	0.94	0.77	16
6	1.00	0.83	0.90	23
7	0.00	0.00	0.00	1
8	0.75	0.86	0.80	7
9	0.83	0.95	0.88	20
10	0.79	0.79	0.79	19
accuracy			0.80	111
macro avg	0.57	0.63	0.59	111
weighted avg	0.78	0.80	0.78	111

B. K-neighbors classifier

Table 11
K-neighbors Classifier (NFR classification)

	precision	recall	f1-score	support
0	0.78	0.88	0.82	8
1	1.00	0.33	0.50	3
2	0.33	1.00	0.50	1
3	0.78	0.70	0.74	10
4	0.00	0.00	0.00	3
5	0.58	0.88	0.70	16
6	0.75	0.78	0.77	23

7	0.00	0.00	0.00	1
8	0.80	0.57	0.67	7
9	0.86	0.95	0.90	20
10	0.92	0.58	0.71	19
accuracy			0.74	111
macro avg	0.62	0.61	0.57	111
weighted avg	0.76	0.74	0.73	111

C. SVC (an optimized model after hyper-tuning)

Table 12
SVC Classifier (NFR Classification)

	precision	recall	f1-score	support
0	0.88	0.88	0.88	8
1	0.00	0.00	0.00	3
2	0.50	1.00	0.67	1
3	0.89	0.80	0.84	10
4	0.00	0.00	0.00	3
5	0.58	0.94	0.71	16
6	1.00	0.87	0.93	23
7	0.00	0.00	0.00	1
8	0.86	0.86	0.86	7
9	0.95	0.90	0.92	20
10	0.83	0.79	0.81	19
accuracy			0.81	111
macro avg	0.59	0.64	0.60	111
weighted avg	0.81	0.81	0.80	111

As mentioned earlier, the dataset for non-functional requirements is a highly imbalanced dataset with only one example for the category Portability (PO) vs 67 examples for the category Usability (US). To handle this, first, a few additional samples were included in the dataset related to the Portability requirement. Then, class over-sampling techniques using SMOTE were applied to balance the dataset.

Different experiments were conducted on the balanced dataset. Based on the results obtained from these experiments, the model trained using SVC was selected as the best-performing model for the task of multi-class classification.

Table 13
SVC Classifier (NFR Classification on the balanced dataset)

	precision	recall	f1-score	support
0	0.88	0.88	0.88	8
1	0.00	0.00	0.00	3
2	0.50	1.00	0.67	1
3	0.89	0.80	0.84	10
4	0.00	0.00	0.00	3
5	0.60	0.94	0.73	16
6	1.00	0.87	0.93	23
7	0.00	0.00	0.00	1
8	0.86	0.86	0.86	7
9	0.95	0.90	0.92	20
10	0.83	0.79	0.81	19
accuracy			0.81	111
macro avg	0.59	0.64	0.60	111
weighted avg	0.81	0.81	0.80	111

4.3.2 Coding and Testing

Manual Techniques

Traditionally, code is written manually by programmers, requiring a lot of time and effort. Errors get easily introduced by typos, misplaced characters, or incorrect syntax, resulting in bugs and glitches that need to be fixed later. Software developers make use of specialized tools to perform several code-related tasks:

1. Writing code
2. Debugging code
3. Refactoring code
4. Managing code repositories
5. Deploying code

Some of such specialized tools are:

1. Integrated Development Environments (IDEs) like Eclipse, IntelliJ IDEA, Visual Studio, etc.
2. Code Editors like Emacs, Vim, Visual Studio Code, etc.
3. Code Repository and Version Control Systems like Git, SVN, CVS, etc.
4. Code Quality and Static Analysis tools like PMD, Checkstyle, FindBugs, etc.
5. Project Management tools like Jira, Trello, Asana, etc.

Rule-Based Tools and Techniques

To further simplify the coding process and reduce errors, rule-based techniques were introduced. Rule-based systems are logical programs that use predefined rules to make deductions and choices to perform automated actions. These rules which provide triggers and outline corresponding actions, are mostly expressed as "if statements". For example, an email containing the word "invoice" can be a trigger, with the action being to forward the email to the finance team. While rule-based systems mimic human intelligence, they are not AI and they do not learn from mistakes. Instead, they follow rules laid out by humans.

There are rule-based linting tools (Wilson, no date) that can be used to automate some tasks like code quality checks. Another rule-based tool that can be used for translating code (for example C++ to Java) is SWIG (Simplified Wrapper and Interface Generator).

However, these tools have some drawbacks:

1. They are not very intelligent and require a lot of manual input from developers.
2. They are not very good at understanding the context of the code, so they can only do very basic automation.

While manual techniques are time-consuming and error-prone, rule-based techniques have helped to simplify the coding process and reduce errors. However, rule-based systems are limited by the rules laid out by humans.

Large Language Model (LLM) Techniques

Recently, the development of Large Language Models (LLMs) has revolutionized the way we approach coding and testing. LLMs are founded on artificial intelligence and machine learning principles, which means that they can analyze and understand large amounts of natural language data to generate accurate and efficient code. Unlike rule-based techniques, artificial intelligence and machine learning-based techniques can learn and adapt. With LLMs, we can expect to see a significant reduction in the amount of time and resources required for coding and testing, while also improving the quality and reliability of the code being produced.

In RQ1, we identified various programming language tasks that can be automated using large models like Codex. For example, automatically generate code from natural language specifications, or generate test cases based on code. AI-based language models like Codex are very good at understanding the context of the code and can do more advanced automation. Many programming tools now integrate large language models pre-trained on code, which can be used for software development tasks. Some of these tools are:

1. Code completion tools like Kite
2. AI-powered code review tools like DeepCode

3. Bug detection tools like DeepBug

4. Test generation tools like Appvance IQ

These AI-based tools are very intelligent and can automate tasks that would normally require a lot of manual input from developers. They are very good at understanding the context of the code, so they can do more advanced automation. However, these tools are still in their early stages of development and might not be as reliable as traditional tools. They might also be more expensive than traditional tools.

In addition to different tools like Kite which are AI-based, different pre-trained models like Codex, CodeT5, CodeBERT and many more can be extremely useful for automating various code-related tasks. While AI-based tools like Kite are customized to perform a single task like code completion, pre-trained language models like Codex can perform multiple tasks like code generation, code translation, unit test generation, and more and can be incorporated into different workflows. Also, Kite or TabNine provide completion at line or function level while the pre-trained models are capable of completing whole blocks of code. Pretrained models also can generate code by providing instructions in natural language, which is a very powerful feature.

Pretrained models which are open-source like GPT-J and CodeT5 can be further improved through continuous training. They can be finetuned on project code for specific tasks and hence can be customized to user needs even when there are restrictions with data due to data being proprietary. This makes open-source pre-trained models extremely useful.

We explored various pre-trained models to understand different code automation tasks that can be performed using them.

Dataset:

CodeXGLUE (CodeXGLUE, 2020) is a compilation of datasets for the purpos of assessing the performance of machine learning models in various code-related tasks. These tasks include clone identification, detection of defects, cloze testing, completing code, translating code, searching for code, refining code, generating code from text, summarizing code, and generating documentation.

The datasets include both existing ones, such as CodeSearchNet, Defects4J, CONCODE, POJ-104, Bugs2Fix and BigCloneBench, as well as recently introduced ones. The following tables show the scores of different models on various code automation tasks. These scores are calculated on different datasets for different tasks. In the later section, the efficiency of the Codex model is shown for different code automation tasks, when performed in an experimental setup to solve different problem statements.

Code Completion

As we observed in the above sections, large pre-trained models can be used effectively for the task of code completion. The following table shows the token-level and line-level accuracy of different models on completion in Python and Java. As can be observed from the table below, GPT-Neox (250M parameter) model has the highest “Edit-Sim” score on the task of Java completion.

"Edit-sim" score is a similarity metric used to measure the similarity between two pieces of text based on the number of edits required to transform one text into the other. The score is determined by dividing the count of shared edit operations (insertions, deletions, and substitutions) by the total number of edit operations needed to convert both texts into a common form. A higher edit-sim score indicates a greater similarity between the two pieces of text. Amongst the first three models listed in the table 14 below, “CodeGPT-Adapted” has the highest overall score when considering both Python completion scores and Java completion scores.

Table 14
Code Completion (Code to Code)

Code Completion (Code to Code)		
Benchmark	**Dataset**	**Languages**
CodeXGlue	PY150	Python
	Github Java Corpus	Java

Score										
				Python			Java			
				token-level	line-level		token-level	line-level		Overall
Model	Size	Layers	Opensource	Accuracy	EM	Edit-Sim	Accuracy	EM	Edit-Sim	
GPT-2	117M	12	Yes	74.22	38.55	68.94	74.89	24.3	60.70	**69.69**
CodeGPT	124M	12	Yes	74.93	39.11	69.69	76.45	25.3	61.54	**70.65**
CodeGPT-Adapted	124M	12	Yes	75.11	39.65	69.84	77.13	26.43	63.03	**71.28**
GPT-Neox	125M	12	Yes	-	-	-	-	36.57	73.42	-
GPT-Neox	250M	24	Yes	-	-	-	-	41.30	76.74	-

Code Translation

Java and C#

Similarly, the following table compares the accuracy and BLEU score of different models on the task of code translation for Java to C# and C# to Java, using different pre-trained models. The StructCoder model has the highest CodeBLEU score on the CodeTrans dataset for both Java to C# and C# to Java. It also has the highest accuracy on both of these tasks.

Table 15
Code Translation (Code to Code)

Code Translation (Code to Code)									
Benchmark				**Dataset**		**Languages**			
CodeXGlue				CodeTrans		Java, C#			
				Score					
				Java to C#			C# to Java		
Model	Size	Layers	Open-source	BLEU	Accuracy	CodeBLEU	BLEU	Accuracy	CodeBLEU
RoBERTa(code)	125M	12	Yes	77.46	56.10	83.07	71.99	57.90	80.18
CodeBERT	125M	12	Yes	79.92	59.00	85.10	72.14	58.80	79.41
PLBART	140M	6	Yes	83.02	64.60	87.92	78.35	65.00	85.27
CodeT5	220M	12	Yes	84.03	65.90	87.38	79.87	66.90	85.51
StructCoder	220M	12	Yes	85.03	66.60	88.41	80.73	67.70	86.10

Multi-lingual code translation

The following table demonstrates the capability of different large language models like InCoder, CodeGen-Multi, and CodeGeeX. These models support translation between various programming languages.

In the context of the HumanEval dataset, @1, @10, and @100 are evaluation metrics that measure the accuracy of a model's response to a given prompt when compared to human responses.

- @1 represents the accuracy of the model's top-ranked response. It measures how often the model's highest-ranked response is the same as the human-selected response.

- @10 represents the accuracy of the model's top 10 responses. It measures how often the correct response is within the top 10 responses generated by the model.
- @100 represents the accuracy of the model's top 100 responses. It measures how often the correct response is within the top 100 responses generated by the model.

As can be observed, CodeGeeX-FT (the fine-tuned version of CodeGeeX) has the highest @1 score for most translation cases. CodeGen-Multi has a higher score for the task of translation from Python to C++.

Table 16
Multi-lingual Code Translation

Dataset	**Languages**
HumanEval	Python, C++, Java, JavaScript, Go

			Score								
Model	Size	Target / Source	Python			C++			Java		
			@1	@10	@100	@1	@10	@100	@1	@10	@100
InCoder	6.7B	Python				26.11	41.00	54.25	26.74	42.66	61.20
CodeGen-Multi	16B	Python				**35.94**	47.81	59.37	29.27	45.70	64.45
CodeGeeX	13B	Python				26.54	43.56	56.48	25.84	41.52	59.72
CodeGeeX-FT	13B	Python				34.16	46.86	61.22	**41.98**	58.17	72.78
InCoder	6.7B	C++	34.37	58.41	78.57				34.04	57.02	68.70
CodeGen-Multi	16B	C++	33.83	55.37	76.64				43.20	69.84	88.82
CodeGeeX	13B	C++	27.18	49.02	67.69				22.56	40.91	64.08
CodeGeeX-FT	13B	C++	**62.79**	80.39	87.10				**71.68**	81.62	85.84

InCoder	6.7B	Java	42.76	65.55	80.43	40.01	55.17	70.39			
CodeGen-Multi	16B		52.73	69.30	82.74	41.42	54.68	65.50			
CodeGeeX	13B		43.41	68.46	84.03	39.33	58.48	72.36			
CodeGeeX-FT	13B		**75.03**	87.71	95.13	**49.67**	65.65	75.40			

Code Generation

The following table shows different scores on the CONCODE dataset, for the task of producing Java code when given natural language instructions. As can be observed, StructCoder gives the highest score for this task.

Table 17
Code Generation (Text to Code)

Code Generation (Text to Code)						
Benchmark		**Dataset**		**Languages**		
CodeXGlue		CONCODE		Java		
				Score		
Model	**Size**	**Layers**	**Open-source**	**EM**	**BLEU**	**CodeBLEU**
GPT-2	117M	12	Yes	17.35	25.37	29.69
CodeGPT	124M	12	Yes	18.25	28.69	32.71
CodeGPT-adapted	124M	12	Yes	20.10	32.79	35.98
PLBART	140M	6	Yes	18.75	36.69	38.52
CoTexT	220M	12	Yes	20.1	37.4	40.14
CodeT5	220M	12	Yes	22.30	40.73	43.20
StructCoder	220M	12	Yes	22.35	40.91	**44.76**

The table below shows the scores of different LLMs on the HumanEval dataset for the purpose of generating Python code when given instructions in natural language. The Pass - @1, @10 and @100 scores (Xu et al., 2022) are listed. As can be observed, in the open-source category, CodeGen-Mono (16 billion parameters) model has the highest Pass@1 score of 29.28%. This model has been pre-trained on BigPython dataset and has 34 layers. Codex (where the training data is not revealed), is a 12-billion parameter model and has the next highest score of 28.81%. Codex has been fine-tuned on 179 GB of Python files. Similarly, GPT-Neox, which is a 20-billion parameter model trained on the Pile dataset, has a Pass@1 score of 15.4%. This transformer-based model has a total of 44 layers.

Table 18
HumanEval Dataset

Dataset	**Languages**
HumanEval	Python

					Score		
Model	**Size**	**Layers**	**Open-source**	**Training Data**	**Pass @ 1**	**Pass @ 10**	**Pass @ 100**
PolyCoder	2.7B	32	Yes	12 Programming Languages – Github	5.59%	9.84%	17.68%
CodeParrot	1.5B	48	Yes	Python – Github	3.58%	8.03%	14.96%
GPT-Neo	2.7B	32	Yes	The Pile – Mix of text and Github Code	6.41%	11.27%	21.37%
GPT-J	6B	28	Yes	The Pile – Mix of text and Github Code	11.62%	15.74%	27.74%
INCODER	6.7B	32	Yes	Public GitHub/GitLab code and StackOverflow	15.2%	27.8%	47.0%

GPT-NeoX	20B	44	Yes	The Pile - Mix of text and Github Code	15.4%	25.6%	41.2%
CodeGen-Multi	6.1B	33	Yes	BIGQUERY	18.16%	28.71%	44.85%
CodeGen-Multi	16.1B	34	Yes	BIGQUERY	18.32%	32.07%	50.80%
CodeGen-Mono	6.1B	33	Yes	BIGPYTHON	26.13%	42.29%	65.82%
CodeGen-Mono	16.1B	34	Yes	BIGPYTHON	**29.28%**	49.86%	75.00%
Codex	12B	Not indicated	No	Not mentioned	**28.81%**	46.81%	72.31%

We also show below the results of generating Python code from natural language on the APPS dataset. The APPS dataset is made up of various programming problems that were gathered from open-access coding websites, including but not limited to Codeforces and Kattis. The problems in the APPS dataset span a range of difficulty levels, from introductory to collegiate competition level, and are designed to assess both coding ability and problem-solving skills. As can be observed below, CodeRL+CodeT5 model, which is a 770 million parameter model has the highest Pass @1 score for all the different difficulty levels. In the open-source models, GPT-J, which is a 6-billion parameter model, has a Pass @1 score of 5.60 on the introductory level dataset. Codex has a Pass @100 score of 25.02 on the introductory level. In addition, a new framework, Parsel, for algorithmic reasoning, shows promise in improving the pass rates than those obtained using models like AlphaCode and Codex.

Table 19
Scores on APPS Dataset

Dataset	**Languages**
APPS	Python

			Score											
Model	**Size**	**Open-source**	**Pass @ 1**				**Pass @ 5**				**Pass @ 100**			
			Intro	**Inter**	**Comp**	**All**	**Intro**	**Inter**	**Comp**	**All**	**Intro**	**Inter**	**Comp**	**All**
Codex	12B	No	4.14	0.14	0.02	0.92	9.65	0.51	0.09	2.25	25.02	3.70	3.23	7.87
Alpha Code	1B	No									17.67	5.24	7.06	8.09
GPT-3	175B	No	0.20	0.03	0.00	0.06								
GPT-2	1.5B	Yes	1.30	0.70	0.00	0.68	3.60	1.03	0.00	1.34	25.00	9.27	8.80	12.32
GPT-Neo	2.7B	Yes	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58	27.90	9.83	11.40	13.76
GPT-J	**6B**	Yes	5.60	1.00	0.50	1.82	9.20	1.73	1.00	3.08	35.20	13.15	13.51	17.63
CodeRL+ CodeT5	**770M**	No	**7.08**	**1.86**	**0.75**	2.69	16.37	4.95	2.84	6.81	40.00	15.67	17.90	20.98

Defect Detection

In the table below, results from different models on the Devign dataset are shown for the task of defect detection in C programs. The dataset was created to study code vulnerabilities and to help researchers and developers build models that can automatically identify security issues in software code. CoTexT, which is a 220 million parameter model with 12 layers has the highest accuracy of 66.62% on this task.

Table 20
Defect Detection

Defect Detection				
Benchmark	**Dataset**	**Languages**		
CodeXGlue	Devign	C		
Model	**Size**	**Layers**	**Opensource**	**Accuracy**
RoBERTa	125M	12	Yes	61.05
CodeBERT	125M	12	Yes	62.08
PLBART	140M	6	Yes	63.18
VulBERTa-MLP	125M	12	Yes	64.75
CoTexT	220M	12	Yes	66.62

Code Summarization

The results on CodeSearchNet for the task of Code Summarization on six different programming languages are shown below. The Smoothed BLEU-4 score is the metric used here to measure the efficiency of these models. This is a widely used metric in natural language processing and machine translation, and it is often used to evaluate the quality of machine-generated translations by comparing them to one or more human-generated translations. The smoothing technique used in the calculation of the score is designed to reduce the impact of chance matches and make the score more robust and reliable. CodeT5, a 220 million parameter model, has the highest score on this task of code summarization, across all programming languages.

Table 21
Code Summarization (Code to Text)

Code Summarization (Code to Text)		
Benchmark	**Dataset**	**Languages**

CodeXGlue	CodeSearchNet	Python, Java, PHP, JavaScript, Ruby, Go

				Smoothed BLEU-4 Score						
Model	**Size**	**Layers**	**Opensource**	**Ruby**	**JavaScript**	**Go**	**Python**	**Java**	**PHP**	**Overall**
RoBERTa	125M	12	Yes	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	125M	12	Yes	12.16	14.90	18.07	19.06	17.65	25.16	17.83
PLBART	140M	6	Yes	14.11	15.56	18.91	19.3	18.45	23.58	18.32
T5	220M	12	Yes	14.18	14.57	19.17	19.26	18.35	24.59	18.35
CoTexT	220M	12	Yes	14.02	14.96	18.86	19.73	19.06	24.68	18.55
CodeT5	220M	12	Yes	**15.24**	**16.16**	**19.56**	**20.01**	**20.31**	**26.03**	**19.55**

Code Refinement

The table below shows the efficiency of different LLMs on the task of refining Java code which may have bugs. The dataset used here is Bugs2Fix. On the small test set, CoText has the highest CodeBLEU score, while CodeT5 has the highest accuracy. For the medium test set, however, CodeBERT has the highest CodeBLEU score.

Code Refinement (Code to Code)		
Benchmark	**Dataset**	**Languages**
CodeXGlue	Bugs2Fix	Java

				Score					
				Small test set			Medium test set		
Model	Size	Layers	Open-source	BLEU	Accuracy	Code BLEU	BLEU	Accuracy	Code BLEU
T5	220M	12	Yes	74.94	15.3	75.85	88.28	4.11	85.61
RoBERTa (code)	125M	12	Yes	77.30	15.90		90.07	4.10	
CodeBERT	125M	12	Yes	77.42	16.40	75.58	91.07	5.16	**87.52**

PLBART	140M	6	Yes	77.02	19.21	-	88.5	8.98	-
CodeT5	220M	12	Yes	77.43	**21.61**	-	87.64	13.96	-
CoText	220M	12	Yes	77.79	21.03	**76.15**	88.4	13.11	85.83

Codex vs Manual

We observed in the sections above, that different LLMs perform well on different programming language tasks. Also, they have all been evaluated on datasets, specific to each of those tasks. Some of these datasets also had programs used in programming competitions. However, this study intends to evaluate the applicability of these LLMs in an enterprise environment, dealing with some real and complex problem statements. We observed in the previous section, that Codex showed good results on various programming language tasks.

We now use the experimental setup described in the research design section, to measure the efficiency of Codex on a variety of tasks. This is the same as the setup used for software requirements classification. The different coding tasks will be performed by three individuals with different expertise levels in the desired programming language, manually. At the same time, the same tasks will be performed by a novice programmer but with the help of an AI code assistant like Codex / gpt3.5-turbo or GPT-4.
This setup is depicted below.

Table 22
Research Setup

First Group	Second Group	Third Group
Novice programmer (Manual code generation)	Intermediate programmer (Manual code generation)	Expert programmer (Manual code generation)
Novice programmer with AI code assistant like Codex		

We identified four programming tasks to evaluate the performance of large language models against traditional techniques:

1. Code generation - Build an Android Java App for drawing on a screen using touch
2. Code generation - Build an ASP.NET C# application for making travel reservations
3. Code translation - Translate a simple game built using C++ to Java
4. Code documentation - Document legacy code

Code Generation

For evaluating LLM for code generation, we consider two problem statements involving two different technologies:

1. Build an Android Java App for drawing on a screen using touch
2. Build an ASP.NET C# application for making travel reservations

Android Java App

The intent here was to start with a simple code generation example. We start with evaluating the capability of an AI model in helping build a simple Android Java app that enables users to draw on the screen of their mobile device using touch gestures. The app will include basic features like changing the color of the drawing and can provide a straightforward and easy-to-use tool for people who want to express their creativity or take notes using their mobile devices.

The main goal of this app is to provide a simple and convenient way for users to draw on their mobile devices without needing complex tools or software. The audience for this app can include anyone who wants a basic drawing app on their device, such as children, hobbyists, or casual users. The importance of this task lies in the fact that it provides a simple interface for people to draw and express their creativity on their mobile devices. The essential features of this app will include touch-based drawing and the ability to change the color of the drawing. It can also include basic features like a clear screen option to start a new drawing.

The code generated using Codex is attached in the appendix. The code generated was almost 90% accurate and needed some corrections to make it a fully functional app. The AI-generated code with manual corrections was used to create this mobile app and the app was deployed on an Android phone. Figure 28 below shows the screenshot of the Android app built using LLM and a simple drawing made using the same.

Individuals with different expertise levels built similar fully functional drawing apps. Everyone took different time to complete building the app.

Figure 28
Android drawing app

ASP.NET C# Travel Reservation application

The second application we want to build by augmenting an AI model is a travel reservation application. For this study, the application will not have many complexities as a real travel application. However, this application has more complexity than the previous application. This application involves having a user interface created using ASP.NET Webforms and integration with a MySQL database.

The intent of the application is for the user to be able to add passenger details like passenger name, email, address, and so on. This screen facilitates adding new passengers, deleting a passenger, and editing the passenger details. When a passenger is selected, the application will direct the user to the next page, where the travel details for that passenger will be reflected. Again, the user would have the option to add a new travel booking for the passenger or update or delete the existing details. Similarly, from the main page, the application will allow the user to view the accommodation for a selected passenger. This would direct the user to the accommodation page where the user can view all the accommodation-related details for the passenger and add new details or update or delete the existing details.

We used OpenAI Codex models initially to generate the code for the ASP.NET WebForms for all three screens. The model successfully generated the logic for the webforms as desired. Prompt engineering was required to be done to help the model understand the requirement better and accordingly generate the desired code. The model was also used to generate the SQL commands for creating the MySQL tables with the desired structure. Almost 90% of the code generated by the model was correct while the remaining 10% of the code needed some modifications/updates. Post the corrections, the application was a ready-running application.

Later, we used gpt3.5-turbo (ChatGPT) to improve the user interface of the application. Again 90% of the suggested code worked as-is without the need for major modifications. The figures below show the screenshots of the final application created by leveraging the AI models.

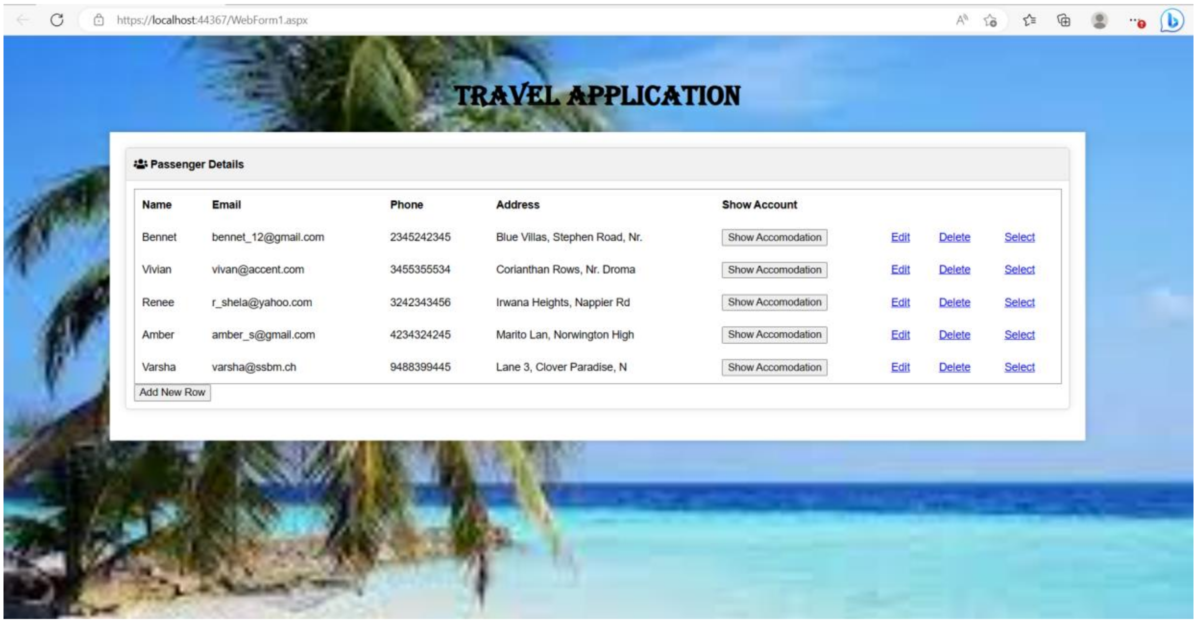


Figure 29
Screen1 - User Details

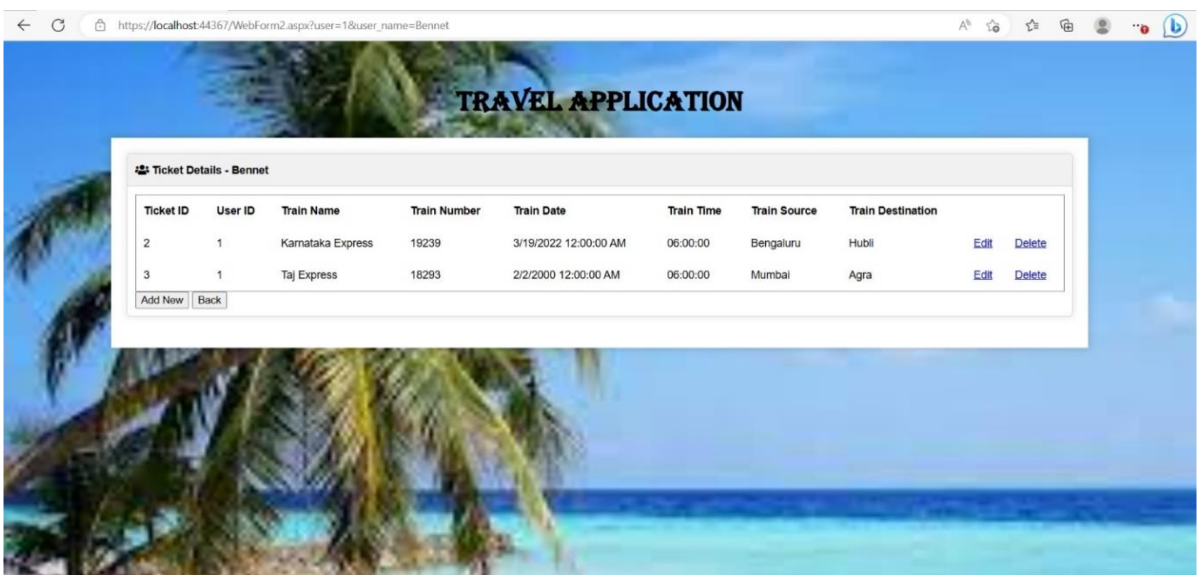


Figure 30
Screen 2 - Travel Details

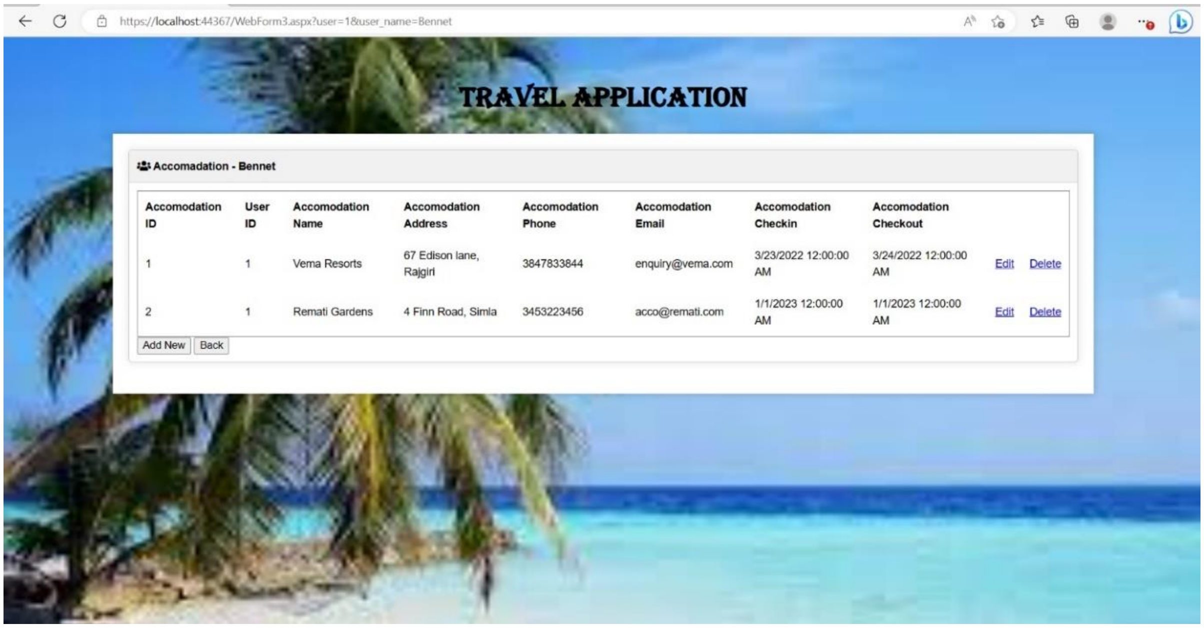


Figure 31
Screen 3 - Accommodation Details

Individuals with different expertise levels built similar fully functional drawing apps. Everyone took different time to complete building the app. The effort and cost estimates in building this app are discussed in the next section. Just as in the previous Android application, parallelly, this application was also developed manually by individuals with different expertise levels. The time taken by everyone was different but they were all able to build a fully functional prototype. The effort and cost estimates in building this application manually versus by using the AI model will be discussed in the next section.

Code Translation

For the task of code translation, an existing C++ application was selected. The target programming language that was chosen for this translation was Java. The C++ application was a game of Tic-Tac-Toe.

This application could be played in two modes:

1. Human vs Human
2. Human vs Computer

The intention is to translate this application to Java and evaluate the efficiency of the AI model in producing a working functional prototype. The source C++ code had different header and code files and some utility files. In the previous task of code generation, the challenge related to context window limitation was less, since the input was a set of instructions, which did not consume many tokens. This allowed the model to generate coherent code by utilizing the remaining number of tokens.

In the case of code translation, there were two major challenges:

1. The input to be given to the model was a piece of code that consumed many tokens. Subsequently, the available number of tokens for generating the translated code was limited.
2. The input files were sometimes large and there were dependencies between the application file and the header files. This, along with the context window limitation of the model, made the translation task using an LLM, more challenging.

The target translated Java code generated by the model for this task is attached in the appendix. The models used for the task of translation have a good understanding of modern programming languages. Hence, the model was correctly able to translate around 90% of the code. The remaining code had to be manually corrected/updated. Post the manual review, updates, and corrections, a fully functional Java application was ready. The figures below show some screenshots of the translated Java application.

As in the other cases, the same task of translating the C++ application to Java was carried out manually by individuals with different levels of expertise in both C++ and

Java. Here, the individuals were required to understand not just the source C++ language, but the target Java language as well. As in the other tasks, all the individuals were able to translate the application, but with different turnaround times. These efforts and cost estimates will be discussed in the next section.

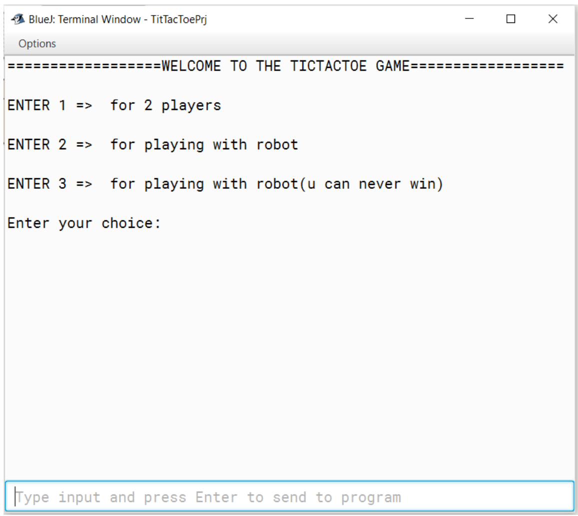


Figure 32
User options screen

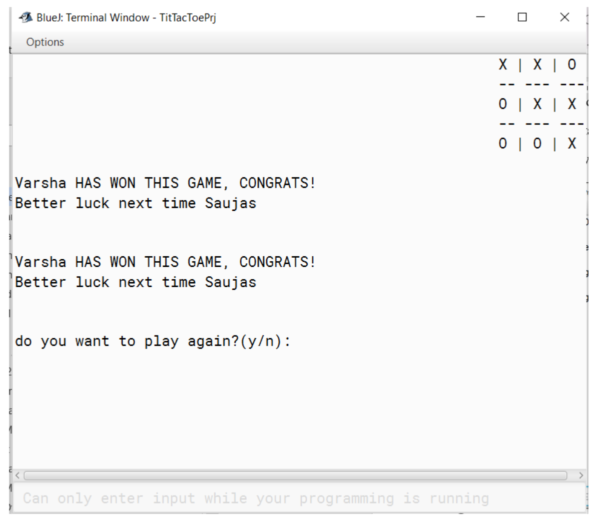


Figure 33
Human vs Human

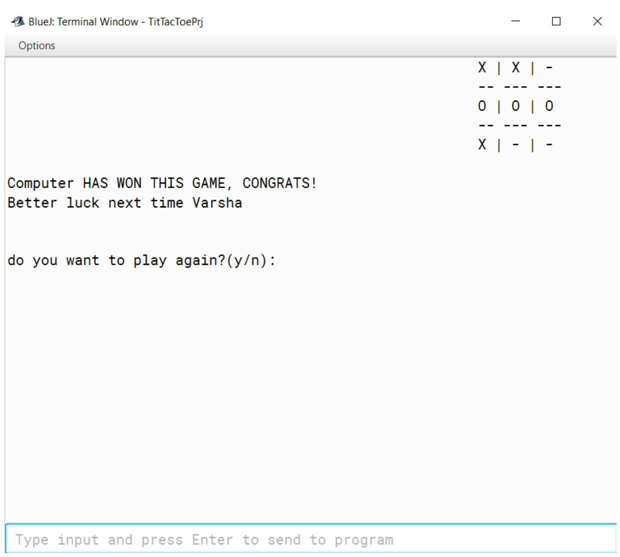


Figure 34
Human vs Computer

Technical Document Generation

For the above task of code translation, the AI model was also used to generate documentation for two source C++ files. The documentation explained the intent of the source program with other important details. This was used to validate that the generated Java code satisfies all the requirements and implements all features in the source files.

Legacy Code Documentation and Migration

Legacy systems have been around for many years and still serve a critical function within an organization. These systems, although functional, may become difficult to maintain and update as they become outdated and require specialized knowledge to maintain. One example of a legacy system is the COBOL programming language.

COBOL is a programming language that has been around since the late 1950s. It was initially designed for business applications. Despite being over 60 years old, COBOL continues to be a vital part of many critical systems in industries such as finance and government. COBOL code can be complex, and understanding it requires specialized knowledge and experience. However, maintaining the COBOL code can be a challenge. The aging workforce that worked on COBOL systems may be retiring or leaving the organization, taking their knowledge of the system with them. This knowledge gap can make it challenging to maintain and update COBOL code, especially if the documentation is lacking.

As discussed in RQ1, documentation is essential for understanding how a program works and how it should be maintained. It provides a reference for programmers and other stakeholders to understand the code's functionality and design. Documentation also helps in identifying potential areas of risk or bugs within the code. Furthermore, it provides a foundation for new developers to understand the system and make modifications effectively. However, documentation is often neglected or not updated, resulting in inadequate knowledge and comprehension of the codebase. This is particularly true for legacy systems, where documentation may have been lost over time or was never created.

Generating documentation from legacy COBOL code can be a time-consuming and challenging task. However, it is essential for maintaining and updating COBOL systems effectively. The following are some of the reasons why generating documentation from legacy COBOL code is necessary:

1. *Understanding the Codebase*

   Generating documentation can provide a comprehensive understanding of the COBOL system. It can help identify dependencies, workflows, and potential areas of risk or bugs within the code. This knowledge is crucial for maintaining and updating the system effectively.

2. *Facilitating System Upgrades*

   Legacy COBOL systems may require upgrades to meet modern requirements. However, without proper documentation, it can be challenging to identify the areas that need to be updated or modified. Documentation can provide a roadmap for making upgrades to the system.

3. *Enabling Effective Collaboration*

   Legacy COBOL systems may have multiple stakeholders, including programmers, managers, and other stakeholders. Documentation can help facilitate effective collaboration between these stakeholders, ensuring that everyone understands the system's functionality and design.

4. *Knowledge Transfer*

As mentioned earlier, the aging workforce that worked on COBOL systems may be retiring or leaving the organization. Documentation can provide a means of transferring knowledge to new developers, ensuring that the system's knowledge is not lost.

In this experiment, we aim to generate documentation for an existing COBOL application and then migrate this application to Java. Once documentation is in place, migration to modern programming languages such as Java becomes a more manageable task. The knowledge gained from the documentation can help identify potential areas of risk or bugs within the code, making it easier to address them during migration. It can also help identify the dependencies and workflows required for the system to function correctly, making it easier to make the necessary modifications required for migration.

For this experiment, we aim to migrate a “Human Resource Management System” available at “*https://www.sourcecodesworld.com/source/show.asp?ScriptID=469*” (Project - Human Resource Management System. - COBOL Projects Source Code in COBOL, no date). This is a COBOL application, which maintains data about employees, their leaves, transfer requests, confirmation status, payments, and so on. As in the previous experiments, this experiment is also carried out using a similar setup. Unlike the C++ to Java translation, which consisted of many files which had interdependencies, here the COBOL application was one single file. However, this was a large piece of COBOL code, and a block of COBOL code linked or branched to other blocks based on different conditions. Capturing the dependencies between these blocks of code, documenting them, and then migrating them to Java was a complex task.

The AI model helped accomplish this task of first generating documentation and then converting it to Java with an overall accuracy of 70%. Figure 35 below shows all the Java classes created for the single COBOL file.

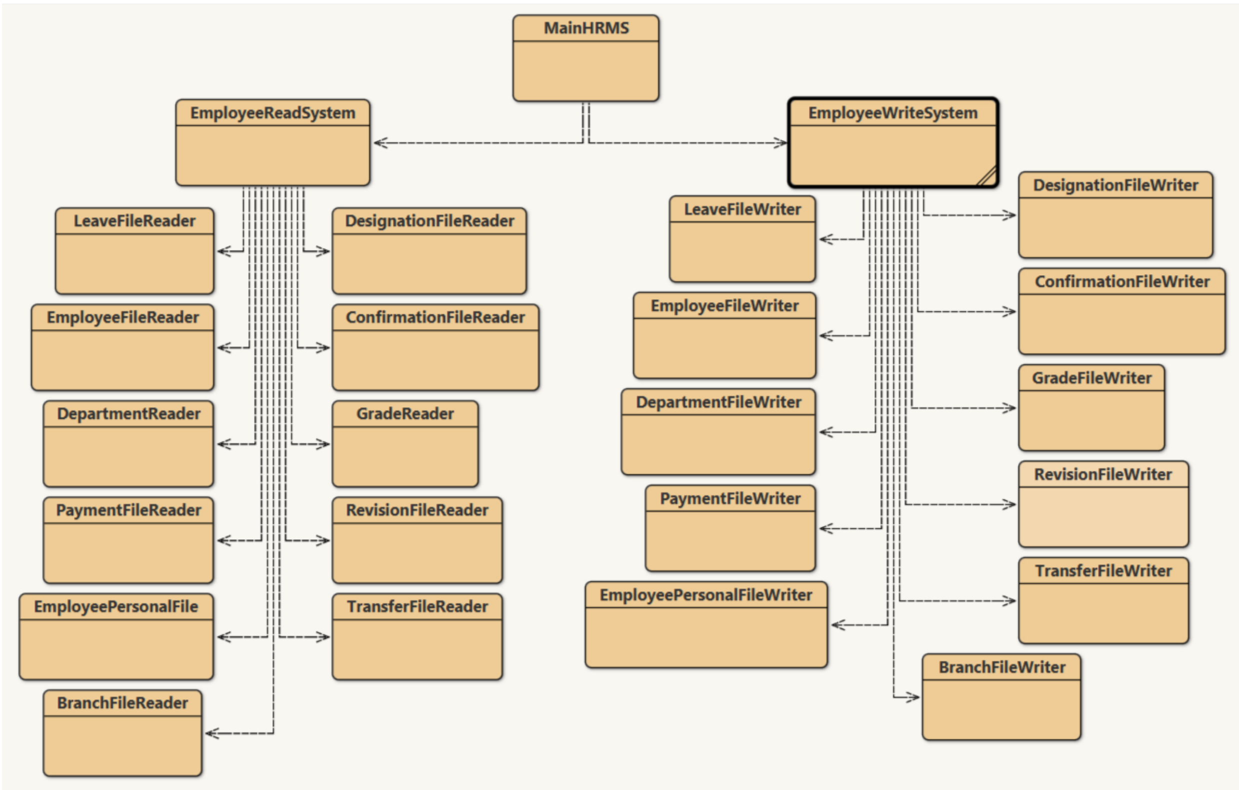


Figure 35
Java Classes for Migrated Application

Figure 36 below shows screenshots of the fully functional migrated Java application.

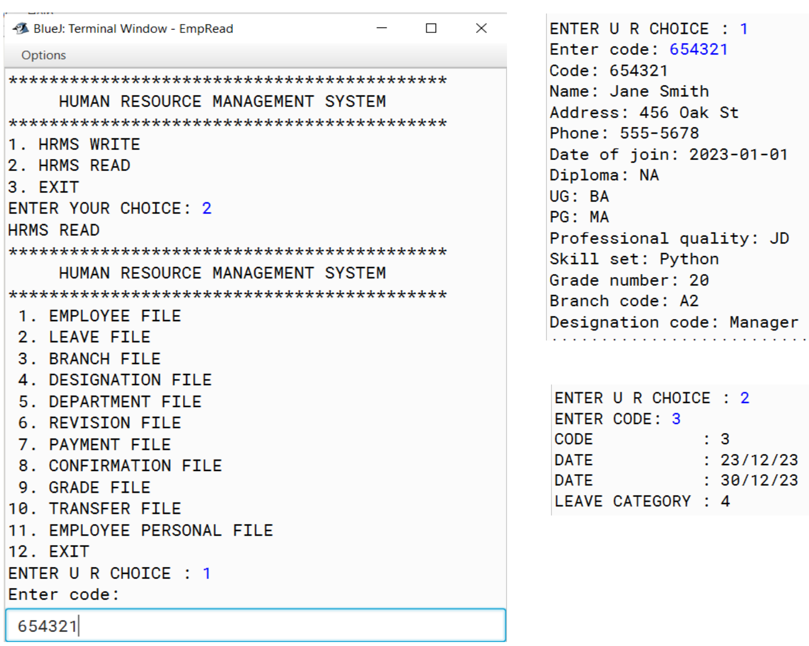


Figure 36
Migrated Java Application

It is to be noted that the COBOL application is a relatively simple application without interdependencies on other COBOL files or CICS libraries. All these dependencies will increase the complexities involved in migration and effectively reduce the accuracy.

The same task of documenting and migrating the legacy COBOL application to Java was carried out manually by three individuals with varying levels of expertise, as in the other tasks. However, there was one challenge involved in the manual setup. This experiment involves translating from a legacy language like COBOL to a modern language like Java. For the setup, where this task will be carried out manually, the resource must understand both these languages. This was a challenge, since people who have expertise in legacy languages like COBOL, generally do not have a great understanding of modern programming languages like Java and vice versa. Hence, in this setup, in the intermediate and expert group, the resources involved have an intermediate and expert understanding respectively of Java but only a basic understanding of COBOL.

All three individuals could document and migrate the application. However, all the individuals took a different amount of time to complete the tasks. The detailed cost and effort analysis for both manual and AI-supported development will be discussed in the next section.

To summarize, the code generated using the AI model, for all the problem statements was corrected and tested after corrections. The statistics on the number of edits needed and the effort needed to generate or perform the coding tasks are described in the next sections for each problem statement.

After performing the required modifications manually, all applications built using the Codex were fully functional applications meeting the desired requirements.

For all the programming tasks discussed in this section, there are a few key observations on the code written by the three individuals vs code generated using the AI model:

1. Since every individual has different expertise in the programming language and different programming styles, the code produced by all three individuals was inconsistent in terms of coding style and structure.
2. The variable names or function names used by the individuals were not always self-explanatory. The AI model, however, used naming conventions, which were self-explanatory.
3. The quality of code written by an expert was better than the code written by a novice programmer. The quality of code generated by the AI model was equivalent to that of an intermediate programmer.
4. All three individuals took significantly more amount of time than the time taken by the AI model to generate the code.

The above results in the requirements, design, and build phases support our first hypothesis, that the software requirements, design, and code capabilities of large language models are better than traditional models.

In the next section, we compare the effort-saving and cost-benefit analysis.

4.4 Research Question Three

How much effort saving and cost-benefit will organizations get by augmenting these AI-based models with a software programmer?

We hypothesized that:

Hypothesis 2

If large language models are to be used optimally, then a human-in-the-loop is necessary for validation of the result generated by the model.

Hypothesis 3

If code automation using large language models is leveraged in the industry, it can improve software quality by reducing errors and increasing consistency.

Hypothesis 4

If large language models are used for code automation, it can improve software development efficiency by reducing the time needed to develop code and provide significant cost and effort benefits to software organizations.

In our findings to RQ2, we observed that large language models can perform many code automation tasks. We also observed that the capabilities of these models are better than traditional tools or methods using certain quality metrics.

In this section, we will try to estimate the effort saving and cost-benefit, if any, that can be achieved using these large language models.

4.4.1 Planning and requirements gathering

Software requirement classification

Effort

In this section, we discuss the effort required to perform the task of software requirement classification. We analyze the effort involved when this task is performed manually as against using a large language model. As discussed in section 4.2.2.1.1, two classification tasks are being performed – binary classification and multi-class classification of NFRs.

<u>Large-language Model</u>

Approach 1 – Zero-shot learning

The following tasks were involved in performing requirement classification using zero-shot learning approach. These tasks are the same for both binary classification as well as multi-class classification.

1. Prompt-engineering

Prompt engineering is a technique used with large language models to carefully design input prompts for the model, that can guide the model toward a specific task or goal. This involves choosing the right words, phrases, and structure for the prompt, as well as selecting relevant background information and context that will help the model understand the task at hand. The goal of prompt engineering is to create a clear and concise prompt that provides the necessary context and constraints for the model to meet the specific objective or goal. Effective prompt engineering can greatly enhance the performance of these large pre-trained models and help achieve a given level of accuracy.

Prompt engineering is performed in iterations. In the first iteration, a basic prompt is designed and the model's performance is observed on a small set of data. In the case of OpenAI GPT-3, this activity is performed in the playground provided by OpenAI.

2. Data preparation/cleaning

Large language models like GPT-3 break down the input text into tokens, where each token is a smaller unit of the word or sub-words. GPT-3 uses byte-pair encoding (BPE) for tokenization, which is a technique that divides words into sub-words based on their frequency of occurrence in the training data. The resulting vocabulary of BPE tokens is used to represent the input text in a format that can be processed by the GPT-3 model. Tokenization in GPT-3 plays a crucial role in improving the performance and efficiency of the model.

Models like GPT-3 are sensitive to spaces and new-line characters. Also, any special characters like emojis, exclamation marks, and so on, which do not play a critical role in the classification process, must be removed, to improve the quality of text being fed to the model.

3. After the data is cleaned and prepared, the OpenAI GPT-3 API is invoked in a loop for each requirement to be classified. The instructions along with the requirement to be classified are sent to the end-point, which then returns the result based on the designed prompt. The results are then validated against the ground truth.
4. Based on the first set of results and the accuracy obtained, the prompt is further tuned to help the model improve the understanding of the end goal. This process is again done in the playground and a better-performing prompt is designed.
5. Step 3 is now repeated to arrive at the final scores.

Binary Classification

Table 23 below shows the effort involved in performing the binary classification task using zero-shot learning.

Table 23
Binary Classification Effort (Zero-shot)

Sr. No.	**Task**	**Time (in hours)**
1	Prompt-engineering – first iteration	2
2	Building basic solution with below steps:	
	a. Data cleaning	1
	b. Iterative prompt-engineering	5
	c. Evaluation	1
	Total Effort	**9 hours**

Time taken to perform classification on 625 requirements using the above solution:

Table 24
Processing Time for 625 requirements

Time taken to process each request	3 seconds
# of requirements	625
Total time taken	~30 minutes

The total effort involved in building the solution and performing binary classification is 9.5 hours.

Multi-class classification

Similarly, Table 25 below shows the effort involved in performing the multi-class classification task using the zero-shot learning approach.

Table 25
Multi-class classification (Zero-shot)

Sr. No.	Task	Time (in hours)
1	Prompt-engineering – first iteration	4
2	Building basic solution with below steps:	
	a. Data cleaning	1
	b. Iterative prompt-engineering	8
	c. Evaluation	1
Total Effort		**14 hours**

Time taken to perform classification on 370 requirements using the above solution:

Table 26
Processing time for 370 requirements

Time taken to process each request	3 seconds
# of requirements	370
Total time taken	~20 minutes

The total effort involved in building the solution and performing binary classification = 14.5 hours

Approach 2 – Embeddings based model

The following tasks were involved in training a custom model on top of a large language model for software requirements classification. As in the case of binary classification, these tasks are the same for both binary classification as well as multi-class classification.

1. Data cleaning and preparation

This step is the same as in approach 1. However, before performing this step, the dataset is first split into a train set (70%) and a test set (30%). All the steps are now performed on the training dataset. In this step, all the extra lines, spaces, and emojis or special characters that may be present in the text are removed. Feature extraction from the requirements is not required as the embeddings, which will be generated in the next step, will capture the semantics of the requirement, which will then be used for further training the model.

2. Embeddings generation

When the clean data is ready, the GPT-3 embeddings API is invoked to generate the embeddings for each requirement in the dataset. GPT-3 embeddings refer to the vector representations of words or tokens learned by the GPT-3 language model during training. Embeddings are a crucial component of GPT-3. These embeddings, since trained on a massive corpus of text data, enable the model to capture a wide range of linguistic patterns and semantic relationships between words. Thus, these high-dimensional vectors can be used to represent complex semantic relationships between words.

The embedding model used to generate the embeddings for the classification problem is text-similarity-davinci-001. This model generates embeddings having 12288 dimensions. OpenAI has recently released text-embedding-ada-002. These embeddings are fewer in dimension but are considered to be more powerful. This embedding model was released after all the experiments for this study were completed using the previous embedding model.

3. Model building

This step is similar to traditional model building. We build and test different classification models like Logistic Regression (LR), Random Forest Classifier (RF), and Support vector machine (SVM) among others. We apply cross-validation techniques to identify the model which will perform well on unseen data. We also perform hyperparameter tuning to further tune the model's performance to achieve better scores. The best-performing model is selected for the next step i.e. evaluating on test data.

4. Evaluating on test data

The selected model is then used to classify the requirements in the test dataset. For each requirement that must be classified, the step of cleaning the post and generating the embedding must be done, before sending it to the model for classification. The accuracy, F1-score, and other metrics are noted.

Binary Classification

Table 27 below shows the effort involved in performing binary classification using the embeddings-based approach.

Table 27
Binary classification effort (Embeddings-based approach)

Sr. No.	**Task**	**Time (in hours)**
1	Data cleaning/preparation for model training	1
2	Generating Embeddings for 437 requirements (training data)	0.25
3	Model training (training different models, cross-validation, hyper-parameter tuning)	14
4	Evaluating on test data (188 posts)	0.25
Total Effort		**15.5 hours**

Multi-class Classification

Similarly, Table 28 below shows the effort involved in multi-class classification using the embeddings-based approach.

Table 28
Multi-class classification effort (Embeddings-based approach)

Sr. No.	Task	Time (in hours)
1	Data cleaning/preparation for model training	1
2	Generating Embeddings for 259 requirements (training data)	0.25
3	Model training (training different models, cross-validation, hyper-parameter tuning)	19
4	Evaluating on test data (111 posts)	0.25
Total Effort		**20.5 hours**

Manual

The same task of binary and multi-class classification was performed manually by three individuals of different expertise.

Binary classification

1. Novice classifier – this individual was new to the classification task with little to no experience in software requirement classification.

   The individual, being a novice, needed guidance from time to time. The total time taken to perform the task was 52 hours.

   Average time per requirement is ~5 mins

   Accuracy – 439 out of 625 (~70%)

2. Intermediate classifier – this individual had some experience in software requirement classification from previous engagements.

   The total time taken to perform the task was 31 hours.

   Average time per requirement is ~3 mins

   Accuracy – 506 out of 625 (~81%)

3. Expert classifier – this individual had extensive experience in performing this task.

    The total time taken to perform the task was 22 hours.

    Average time per requirement is ~2 min

    Accuracy – 531 out of 625 (~85%)

Effort saving with Approach-1

Table 29 below shows the effort saving obtained when using LLM as compared to the same task being done manually by programmers of different levels of expertise.

Table 29
Effort saving with Approach-1

	LLM	**Novice**	**Intermediate**	**Expert**
Time taken	9.5 hours	52 hours	31 hours	22 hours
*Effort saving**		42.5 hours	21.5 hours	12.5 hours
*Percentage effort saving**		82%	69%	57%

Effort saving with Approach-2

The point to be noted here is that the total time taken here for LLM is the time taken to train the model on 70% of the dataset and test on the remaining 30% of the dataset. While the time taken manually indicates the time taken to manually label the entire dataset of 625 software requirements.

Table 30
Effort saving with Approach-2

	LLM	**Novice**	**Intermediate**	**Expert**
Total Time taken	15.5 hours	52 hours	31 hours	22 hours
*Effort saving**		36.5 hours	15.5 hours	6.5 hours
*Percentage effort saving**		70%	50%	30%

*Effort saving – Manual classification effort – LLM effort
*Percentage effort saving – Effort saving / Manual classification effort

Multi-class classification

The task of multi-class classification was more complex to be done manually as compared to binary classification since the number of classes involved here is large (11). Depicted below is the manual effort involved.

1. Novice classifier – The total time taken to perform the task was 40 hours.

   Average time per requirement is ~6.5 mins

   Accuracy – 229 out of 370 (~62%)

2. Intermediate classifier – The total time taken to perform the task was 31 hours.

   Average time per requirement is ~5 mins

   Accuracy – 263 out of 370 (~71%)

3. Expert classifier – The total time taken to perform the task was 21.5 hours.

   Average time per requirement is ~3.5 min

   Accuracy – 296 out of 370 (~80%)

Effort saving with Approach-1

Table 31 below shows the effort saving obtained when using LLM to perform multi-class classification using Approach-1, as compared to the same task being performed manually by programmers of different levels of expertise.

Table 31
Multi-class classification Effort Saving (Approach-1)

	LLM	**Novice**	**Intermediate**	**Expert**
Time taken	14.5 hours	40 hours	31 hours	21.5 hours
*Effort saving**		25.5 hours	16.5 hours	7 hours
*Percentage effort saving**		64%	53%	33%

Effort saving with Approach-2

The point to be noted here is that the total time taken here for LLM is the time taken to train the model on 70% of the dataset and test on the remaining 30% of the dataset. While the time taken manually indicates the time taken to manually label the entire dataset of 370 non-functional requirements.

Table 32
Multi-class classification Effort saving (Approach 2)

	LLM	**Novice**	**Intermediate**	**Expert**
Total Time taken	20.5 hours	40 hours	31 hours	21.5 hours
*Effort saving**		19.5 hours	10.5 hours	1 hour
*Percentage effort saving**		49%	34%	5%

*Effort saving – Manual classification effort – LLM effort
*Percentage effort saving – Effort saving / Manual classification effort

Cost Estimate

Large language Model

We used the OpenAI Davinci model for performing the binary classification using zero-shot prompt engineering. Similarly, we used OpenAI similarity embeddings for building a classifier model using the embeddings. Both these models are charged per 1000 tokens. The charges are as mentioned below:

text-davinici-002 model – $0.02 per 1000 tokens

text-similarity-davinci-001 model - $0.2 per 1000 tokens

Approach 1 - Zero-shot learning

Binary classification

Table 33 below lists various factors which contribute to calculating the cost involved in performing binary classification using the zero-shot approach.

Table 33
Factors impacting cost - binary classification (zero-shot)

# Requirements	625
Total # of tokens for all requirements	14,434
# Tokens in instruction	200

# Tokens in completion (max)	3
Total # of tokens (instruction + requirement + completion)*	14,434 + (625 * 200) + (625 * 3) = 1,41,309
Cost of completion API	$0.02 per 1000 tokens
Total cost of inferencing	$3

* Total # of tokens = Total # of tokens for all requirements + (# requirements * # of tokens in instruction) + (# requirements * # of tokens in completion)
* Total cost = Total # of tokens * Cost of API / 1000

Labor cost

Assuming an hourly rate of $40 for AI resource in performing classification using LLM, the cost of performing the classification using Approach-1 is:

Total effort = 9.5 hours

Rate = $40/hour

Resource cost = $40* 9.5 = $380

Total cost = Effort cost + Cost of inferencing = $383

Multi-class classification

Similarly, Table 34 below shows different factors involved in calculating the cost for the multi-class classification of NFRs.

Table 34
Factors impacting cost multi-class classification (zero-shot)

# Non-functional requirements	370
Total # of tokens for all requirements	8,932
# Tokens in instruction	59
# Tokens in completion (max)	10
Total # of tokens (instruction + requirement + completion)*	8,932 + (370 * 59) + (370 * 10) = 34,462
Cost of completion API	$0.02 per 1000 tokens
Total cost of inferencing	$0.69

* Total # of tokens = Total # of tokens for all posts + (# posts * # of tokens in instruction) + (# posts * # of tokens in completion)
* Total cost = Total # of tokens * Cost of API / 1000

Labor cost

Assuming an hourly rate of $40 for AI resource in performing classification using LLM, the cost of performing the classification using Approach-1 is:

Total effort = 14.5 hours

Rate = $40/hour

Resource cost = $40 * 14.5 = $580

Total cost = Effort cost + Cost of inferencing = $581

Approach 2 – Embeddings based model

Binary classification

The table below lists the factors considered during calculating the cost for performing binary classification using the embeddings-based approach

Table 35
Factors for cost calculation (Binary classification – embeddings based)

# Requirements	625
Total # of tokens for all requirements	14,434
Cost of embedding API	$0.2 per 1000 tokens
Total cost of generating embeddings	$3

Labor cost

Assuming an hourly rate of $40 for an AI resource involved in building the classification model using LLM, the cost of performing the classification using Approach-2 is:

Total effort spent in building and evaluating the model = 15.5 hours

Rate = $40/hour

Resource cost = $40 * 15.5 = $580

Total cost = Resource cost + cost of embeddings = $583

Multi-class classification

Similarly, Table 36 below lists the factors considered during calculating the cost for multi-class classification using the embeddings-based approach

Table 36
Factors impacting cost multi-class classification (embeddings-based)

# Requirements	370
Total # of tokens for all requirements	8,932
Cost of embedding API	$0.2 per 1000 tokens
Total cost of generating embeddings	$2

Labor cost

Assuming an hourly rate of $40 for a data scientist involved in building the classification model using LLM, the cost of performing the classification using Approach-2 is:

Total effort spent in building and evaluating the model = 20.5 hours

Rate = $40/hour

Resource cost = $40 * 20.5 = $820

Total cost = Resource cost + cost of embeddings = $822

<u>Manual</u>

Binary classification

The following costs will be applicable when performing the task of binary classification of software requirements manually - labor cost of the classifiers, the overhead costs associated with the task, and any other expenses incurred during the classification process. Based on these cost factors, the following is the manual cost involved for each manual classifier.

Labor cost

Assuming an hourly rate of $20 for novice classifiers, $30 for intermediate classifiers, and $50 for expert classifiers, the cost of performing the classification task is as follows:

Table 37
Cost of manual binary classification

Novice classifiers: 52 hours x $20/hour = $1040
Intermediate classifiers: 31 hours x $30/hour = $930
Expert classifiers: 22 hours x $50/hour = $1100

Overhead costs

Overhead costs are indirect costs associated with performing the task, such as training, supervision, and administrative expenses. Assuming an overhead cost of 25% of the labor cost, the total cost of performing the task is as follows:

Table 38
Overhead costs – binary classification

Novice classifiers: $1040 x 1.25 = $1,300
Intermediate classifiers: $930 x 1.25 = $1,163
Expert classifiers: $1100 x 1.25 = $1,375

Other expenses

Other expenses associated with performing the task include any expenses associated with obtaining clarification or feedback from stakeholders and any additional expenses incurred during the classification process. Assuming other expenses of $500, the total cost of performing the task is as follows:

Table 39
Other expenses – binary classification

Novice classifiers: $1,300 + $500 = $1,800
Intermediate classifiers: $1,163 + $500 = $1,663

Expert classifiers: $1,375 + $500 = $1,875

Cost saving – Approach-1

The following table shows the cost savings obtained when using an LLM and using the first approach, as against the same task being done manually by individuals with varying levels of expertise.

Table 40
Cost saving (Binary Classification) – Approach 1

	LLM (Zero-shot)	**Novice**	**Intermediate**	**Expert**
Cost	$383	$1,800	$1,663	$1,875
*Cost saving**		$1,417	$1,280	$1,492
*Percentage cost saving**		79%	77%	80%

Cost saving – Approach-2

Similarly, the following table shows the cost savings obtained using approach 2 with LLM as against the manual approach.

Table 41
Cost saving (Binary Classification) – Approach 2

	LLM (Embeddings)	**Novice**	**Intermediate**	**Expert**
Cost	$623	$1,800	$1,663	$1875
*Cost saving**		$1,177	$1,040	$1,252
*Percentage cost saving**		65%	62%	67%

Multi-class classification

As in the case of manual binary classification, the following costs will be applicable when performing multi-class classification manually – labor cost of the classifiers, the overhead costs associated with the task, and any other expenses incurred during the classification process. Based on these cost factors, the following is the manual cost involved for each manual classifier.

Labor cost

Assuming an hourly rate of $20 for novice classifiers, $30 for intermediate classifiers, and $50 for expert classifiers, the cost of performing the classification task is as follows:

Table 42
Cost of manual classification (multi-class)

Novice classifiers: 40 hours x $20/hour = $800
Intermediate classifiers: 31 hours x $30/hour = $930
Expert classifiers: 21.5 hours x $50/hour = $1075

Overhead costs

As in the case of manual binary classification, assuming an overhead cost of 25% of the labor cost, the total cost of performing the task is as follows:

Table 43
Overhead costs (multi-class)

Novice classifiers: $800 x 1.25 = $1000
Intermediate classifiers: $930 x 1.25 = $1163
Expert classifiers: $1075 x 1.25 = $1344

Other expenses

Assuming other expenses of $500, the total cost of performing the task is as follows:

Table 44
Other expenses (multi-class)

Novice classifiers: $1000 + $500 = $1,500
Intermediate classifiers: $1163 + $500 = $1,663
Expert classifiers: $1156 + $500 = $1,844

Cost saving – Approach-1

The following table shows the cost savings obtained in multi-class classification using Approach-1.

Table 45
Cost saving (Multi-class Classification) – Approach 1

	LLM (Zero-shot)	**Novice**	**Intermediate**	**Expert**
Cost	$581	$1,500	$1,663	$1,844
*Cost saving**		$919	$1,082	$1,263
*Percentage cost saving**		61%	65%	68%

Cost saving – Approach-2

The following table shows the cost savings obtained in multi-class classification using Approach-2.

Table 46
Cost saving (Multi-class Classification) – Approach 2

	LLM (Embeddings)	**Novice**	**Intermediate**	**Expert**
Cost	$822	$1,500	$1,663	$1,844
*Cost saving**		$678	$841	$1,022
*Percentage cost saving**		45%	51%	55%

Overall

Binary classification

The following table shows the overall metrics for binary classification, using both approaches.

Table 47
Overall (Binary Classification)

	LLM (Zero-shot)	LLM (Embeddings)	Novice	Intermediate	Expert
Accuracy	81%	92%	70%	81%	85%
% Effort saving with zero-shot LLM			82%	69%	57%
% Effort saving with embedding LLM			70%	50%	30%
% Cost saving with zero-shot LLM			79%	77%	80%
% Cost saving with embedding LLM			65%	63%	67%

We observe, that when using an embeddings-based model-building approach, the effort saving is 30% when compared to the same work done by an expert manually. In terms of cost, we see a saving of 67% as against the classification done manually by an expert classifier. Also, the accuracy of the model is greater (92%) than manual expert classification (85%).

We also observe that the accuracy of the zero-shot approach is the same as the manual intermediate classifier (81%) but lower than the manual expert classifier (85%). However, the cost and effort savings in both cases are considerably higher. The effort saving against a manual intermediate classifier is 69% and against a manual expert classifier is 57%. Similarly, the cost saving against both the manual intermediate and manual expert classifier is 77% and 80% respectively.

Multi-class classification

Table 48 provides the overall metrics for multi-class classification, using both approaches.

Table 48
Overall (Multi-class classification)

	LLM (Zero-shot)	LLM (Embeddings)	Novice	Intermediate	Expert
Accuracy	71%	81%	62%	71%	80%
% Effort saving with zero-shot LLM			64%	53%	33%
% Effort saving with embedding LLM			49%	34%	5%
% Cost saving with zero-shot LLM			61%	65%	68%
% Cost saving with embedding LLM			45%	51%	55%

We observe, that when using the embeddings-based model building approach, the effort saving is 5% and cost saving 55% as against the classification done manually by an expert classifier. Also, the accuracy of the model is almost similar (81%) as compared to manual expert classification (80%).

We also observe that the accuracy of the zero-shot approach is the same as the manual intermediate classifier (71%) but lower than the manual expert classifier (80%). Also, the effort saving against a manual intermediate classifier is 53%, and against a manual expert classifier is 33%. Similarly, the cost saving against both the manual intermediate and manual expert classifier is 65% and 68% respectively.

4.4.2 Coding and Testing

4.4.2.1 Code Generation

Task 1 – Android Java App

Intent

The intent of this Android app is very simple – to draw on the screen using touch. Also, the app will have buttons to erase the drawing and to change the colors.

Effort

Large Language Model (Codex)

Tasks

1. Prompt-engineering

Initial prompt

The model was instructed to generate code for such an app with the desired features. The model provided a template code. The template code was accurate with a few missing points:

a. The action to be done on a click of button-1 (erase the drawing was missing)

b. A second button to change the color of the drawing was missing

Prompt-tuning-1

Based on the code generated, further instructions were provided to the model to add the action of erasing the drawing.

Result – the model successfully regenerated the code with this feature in place

Prompt-tuning-2

The model was given further instructions to generate code to add a button which when clicked will change the color to be used for the drawing.

Result – again the model was successfully able to add this feature and provide the final code.

2. Code correction

The next step was to create a project in Android Studio and copy the code. There were a few errors that were corrected.

For example:

- the generated code imported the class `android.app.Activity`. This had to be changed to `androidx.appcompat.app.AppCompatActivity`.

- the generated code set the contentView to the DrawingView object i.e. `setContentView(dv);`. This was changed to set the contentView to the main activity and add the drawingView object to it i.e.

```
setContentView(R.layout.activity_main);
LinearLayout ll = (LinearLayout)findViewById(R.id.llayout1);
ll.addView(dv);
```

Table 49 below shows the statistics on the lines of code generated by the model and the number of edits required.

3. Code testing

In this step, the code is tested to check if the generated code has all the features and functions correctly.

Code Statistics

Table 49
Code Statistics

# LOC generated	# Insertions	# Deletions	# Edits
139	9	1	1

Effort Analysis

Table 50
Effort Analysis

Sr. No.	Task	Time (in hours)
1	Prompt engineering (code generation in multiple iterations)	2.5 hours
2	Code correction	0.5 hours
3	Testing the code and deploying in Android phone	1 hour
Total Effort		**4 hours**

<u>Manual</u>

The same task was given to each of the individual programmers in the novice, intermediate, and expert programmer group. At a high level, the following are the steps that will be required to be done.

1. Setting up the project: This involves creating a new Android project, configuring the project settings, and setting up the layout and UI elements for the app.
2. Implementing touch input: The app needs to handle touch input to enable users to draw on the screen. This may involve setting up touch listeners, detecting touch events, and drawing on the canvas.
3. Adding the erase button: The app needs to include a button that allows users to erase their drawings. This may involve creating a new button widget, setting up a click listener for the button, and implementing the erase functionality.
4. Adding the color button: The app needs to include a button that allows users to change the color of their drawing. This may involve creating a new button widget, setting up a click listener for the button, and implementing the change color functionality.
5. Testing and debugging: Once the app is implemented, it needs to be thoroughly tested to ensure that it works as intended and is free of bugs and errors.

Given that the complexity of the program is simple, the following is the time taken by the individual groups to complete the task:

- Novice programmer – 20 hours
- Intermediate programmer – 14 hours
- Expert programmer – 6 hours

Effort saving

The following table shows the effort saving obtained when using LLM for code generation as against the code written manually by individuals with different levels of expertise.

Table 51
Effort Saving (Code Generation - Android App)

	LLM	**Novice**	**Intermediate**	**Expert**
Time taken	4 hours	20 hours	14 hours	6 hours
*Effort saving**		16 hours	10 hours	2 hours
*Percentage effort saving**		80%	71%	33%

Cost Estimate

Large language model

We used the OpenAI Codex model (code-davinci-002) for generating code, given instructions in natural language. The pricing for this model is the same as text-davinci-002 model used for requirements classification i.e., $0.02 per 1000 tokens

Table 52
Cost of inferencing (Android App)

# Average LOC generated (including newlines) in one pass	169
# of iterations of prompt engineering/fine-tuning	15
# Total tokens generated	30,000
Cost of completion API	$0.02 per 1000 tokens
Total cost of inferencing*	$0.6

* Total cost = Total # of tokens * Cost of API / 1000

Labor cost

Assuming an hourly rate of $40 for AI resource in inferencing LLM and performing prompt engineering, the cost of generating code for this task is:

Total effort spent in code generation and testing – 4 hours

Hourly rate - $40

Resource cost - $40 * 4 = $160

Total cost = Resource cost + cost of inferencing = $161

Manual

Novice programmer: A novice programmer takes longer to complete each step of the development process, and requires more guidance and supervision.

Intermediate programmer: An intermediate programmer has more experience with Android development and can complete each step of the process more quickly and with less guidance.

Expert programmer: An expert programmer has extensive experience with Android development and can complete each step of the process quickly and efficiently.

Assuming an hourly rate of $20 per hour for a novice programmer, $30 for an intermediate programmer, and $50 for an expert programmer, the cost of building this app manually is as follows:

Table 53
Cost of manually building the app – Android App

Novice programmer: 20 hours x $20/hour = $400
Intermediate programmer: 14 hours x $30/hour = $420
Expert classifiers: 6 hours x $50/hour = $300

Cost saving

The following table shows the cost savings obtained when using LLM for building the app.

Table 54
Cost saving - Android App

	LLM (Zero-shot)	**Novice**	**Intermediate**	**Expert**
Cost	$161	$400	$420	$300

*Cost saving**		$239	$259	$139
*Percentage cost saving**		60%	62%	46%

Overall

The following table shows the overall effort and cost savings obtained in building this app using LLM.

Table 55
Overall Savings - Android App

	LLM (Zero-shot)	**Novice**	**Intermediate**	**Expert**
% Effort saving		80%	71%	33%
% Cost saving		60%	62%	46%

Cost-Benefit Analysis (with respect to Expert programmer)

We observe, that the effort saving when using a large language model like Codex, is around 33% and cost saving 46% as against the code written manually by an expert programmer. Similarly, the cost and effort savings in both the other cases (novice and intermediate) are also considerable. The effort saving against a manual programmer is 80% and against an intermediate programmer is 71%. Similarly, the cost saving against both the manual intermediate and manual expert programmer is around 60%.

Task 2 – ASP.NET Travel reservation application

Intent

This task intends to build a small travel reservation app using ASP.NET. The app will allow adding a new user with details like age, name, gender, and so on. The user WebForm will be connected to a backend SQL database. There will option to add/edit/update/delete a user. There will be two more WebForms - one for handling ticketing details for each user, again with the option to edit the details, and another for handling the accommodation for the user. Both these WebForms will also be connected to a backend SQL table.

Effort

Large Language Model (Codex)

Tasks

1. Prompt-engineering

WebForm

The model was first instructed to generate code for the 'user' WebForm. Prompt engineering was done to provide the right instructions to the model to produce the desired output. At times the model only produced the backend logic without the WebForm. In such cases, the model had to be directed to produce the code for the WebForm as well.

Here the requirement was that the WebForm should be connected to the back-end SQL database. The model generated the code which linked the GridView in the WebForm to the SQL table. In addition, when the model was given instructions to use TemplateField instead of BoundField, the model was successfully able to do so as well. Two WebForms were generated in this manner using Codex – User and Ticket

SQL

The model was also used to generate SQL code for creating the User and Ticket tables. The model was provided with the structure of the table and the key and relationship details.

2. Code correction

To test the generated code, a project was created in Visual Studio and the generated code was copied and pasted to the right place. There were some edits, deletions, and insertions needed. For example, in the WebForm code that was generated, there were some placeholder text boxes created as shown below:

```
<asp:TextBox ID="txtName" runat="server" placeholder="Name"></asp:TextBox>
<asp:TextBox ID="txtEmail" runat="server" placeholder="Email"></asp:TextBox>
<asp:TextBox ID="txtPhone" runat="server" placeholder="Phone"></asp:TextBox>
<asp:TextBox ID="txtAddress" runat="server" placeholder="Address"></asp:TextBox>
<asp:Button ID="btnAdd" runat="server" Text="Add" />
```

This was not required and hence deleted. In the application code, there was logic creating the connection string to the database. An edit to the connection string name was required. There were some insertions needed like storing the state of the datatable in a ViewState as the current table as shown below:

```
ViewState["CurrentTable"] = dt;
```

This was required so that the state of the DataTable object could be persisted across postbacks.

Table 56 below shows the statistics on the code edits, deletions, and insertions required.

Table 56
Statistics - ASP.NET

# LOC generated	# Insertions	# Deletions	# Edits
1326	40	30	50
% modification needed in code	~9%		

3. Code testing

The generated code, after the required modifications, is tested to ensure the application has the desired functionality.

Effort Analysis

The following table shows the effort involved in building the ASP.NET application with the assistance of LLM.

Table 57
Effort Analysis - ASP.NET using LLM

Sr. No.	Task	Time (in hours)
1	Prompt engineering (code generation in multiple iterations for: - Three forms with the backend logic - SQL code - Improving UI	10 hours
2	Code correction	3 hours
3	Testing and debugging the code	3 hours
Total Effort		**16 hours**

Manual

The same task of building this travel reservation application was given to each of the individual programmers in the novice, intermediate, and expert programmer group. At a high level, the following are the steps that were required to be done.

1. Setting up the project: This involves creating a new ASP.NET project and configuring the project settings.
2. Creating the three webforms with all the desired options
3. Adding the required back-end logic for each of the three forms
4. Linking the forms as required
5. Testing and debugging: Once the application is implemented, it needs to be thoroughly tested to ensure that it works as intended and is free of bugs and errors.

Building such a travel reservation application is a medium-complexity task. Following is the time taken by the individual groups to complete the task:

- Novice programmer – 160 hours
- Intermediate programmer – 80 hours

- Expert programmer – 40 hours

Effort saving

The following table shows the effort saving obtained in building this app using LLM.

Table 58
Effort saving - ASP.NET

	LLM	Novice	Intermediate	Expert
Time taken	16 hours	160 hours	80 hours	40 hours
*Effort saving**		144 hours	64 hours	24 hours
*Percentage effort saving**		90%	80%	60%

Cost Estimate

Large language model

As in the previous code generation task, we used the OpenAI Codex model (code-davinci-002) for generating code for building this application.

Table 59
Cost of inferencing with Codex (ASP.NET)

# Average LOC generated (including newlines) in one pass	360
# of iterations of prompt engineering/fine-tuning	20
# Total tokens generated	90,000
Cost of completion API	$0.02 per 1000 tokens
Total cost of inferencing*	$1.8

* Total cost = Total # of tokens * Cost of API / 1000

Labor cost

Assuming an hourly rate of $40 for AI resource in inferencing LLM and performing prompt engineering, the cost of generating code for this task is:

Total effort spent in code generation and testing – 16 hours

Hourly rate - $40

Resource cost - $40 * 16 = $640

Total cost = Resource cost + cost of inferencing = $642

Manual

As in the previous case, assuming an hourly rate of $20 per hour for a novice programmer, $30 for an intermediate programmer, and $50 for an expert programmer, the cost of building this app manually is as follows:

Table 60
Cost of manually building the app – ASP.NET

Novice programmer: 160 hours x $20/hour = $3200
Intermediate programmer: 80 hours x $30/hour = $2400
Expert classifiers: 40 hours x $50/hour = $2000

Cost saving

The following table shows the cost savings obtained in building the app using an AI model as against an entirely manual approach.

Table 61
Cost saving - ASP.NET

	LLM (Zero-shot)	**Novice**	**Intermediate**	**Expert**
Cost	$642	$3200	$2400	$2000
*Cost saving**		$2558	$1758	$1358
*Percentage cost saving**		80%	73%	68%

Overall

The following table shows the overall cost and effort savings obtained by using LLM for building the application.

Table 62
Overall - ASP.NET

	LLM (Zero-shot)	Novice	Intermediate	Expert
% Effort saving		90%	80%	60%
% Cost saving		80%	73%	68%

As can be observed, the effort saving when using a large language model like Codex is around 60%, and cost-saving is 68% as against the code written manually by an expert programmer. Similarly, the cost and effort savings in both the other cases (novice and intermediate) are also considerable. The effort saving against a manual programmer is 90% and against an intermediate programmer is 80%. Similarly, the cost saving against the manual intermediate programmer is 73% and the manual novice programmer is 80%.

4.4.2.2 Code Translation

Intent

The intent of this task is to translate an existing game of Tic-Tac-Toe from C++ to Java.

Source

For this experiment, the code is picked from a public GitHub repository, https://github.com/ash-dodek/TicTacToe (Harsh, 2022).

As the author mentions, the game is for two players. The game offers two modes of play – Person against a person and Person against a computer. The intent is to translate this code to Java and do the cost-benefit analysis for the same.

The original C++ repository consists of a set of C++ files along with some header files. Provided below is a brief description of some of the files:

1. Two header files – one each to check if 'X' player is a winner or 'O' player is a winner. The function in the first header returns true if 'X' player is a winner and the second function in another header returns true if 'O' player is a winner.

2. Other than this, there are two header files – one to make each player move on the board and return true if the move was successful.
3. There are two header files for human-to-robot interaction. One file marks the move for the computer while the other checks if the computer wins the game and returns true if it does.
4. One starter file with the main function to start the game and provide options for the user to choose from. The options provided are to play against a human vs a computer.

Overall, seven C++ files in this application need to be migrated to Java.

Effort

Large Language Model (Codex)

Translating this C++ application to Java using a large language model was a challenging task, especially because the model has limitations on context length. Listed below are the steps involved in performing this translation:

1. Analyze the C++ application: Before starting the translation process, it's essential to understand the structure of the C++ application. This step involved analyzing the code and identifying the functions, classes, data types, and libraries used.
   The initial analysis showed that there were six header files, for performing the various moves in the game and for checking the winner. Also, there was one main file that served as an entry point for the program.
2. Preprocess the C++ files: In this step, the C++ files are pre-processed to remove any comments, unnecessary code, and other elements that may not be relevant to the translation process. This helps reduce the input size and makes it easier for the language model to process.
   As shown in Figure 37, the code contained multiple lines of code which were commented. All such lines were removed.

```
                // break;
                // break;
                // defCase();
```

Figure 37
Code with comments

Similarly, all extra new lines present in the code, as shown in Figure 38 were removed as well.

```
                break;


            case 2:
```

Figure 38
Code with line breaks

3. Divide the C++ files into smaller chunks: Next, the C++ files are divided into smaller chunks to avoid exceeding the context length limitations of the language model. This was done by breaking down the code into individual functions or smaller code blocks. This task was challenging as when performing the chunking it was important to not lose the context. For example, when splitting the code, it was observed that including some variable declarations or some function definitions helped the model to understand the code better and hence translate better. For the following piece of code, during the else condition, the control is transferred to the location of the label ifFail. For converting the program to Java, it is critical to understand what the program must do during the else block. Providing the required context becomes critical for the correct conversion of the program.

```
else if(move==8)
{
    if (gameb[2][1]=='-'){
            gameb[2][1]={'O'};
    clr();
    cout
      <<gameb[0][0]<<" | "<<gameb[0][1]<<" | "<<gameb[0][2]<<"\n"
      <<"-- --- --- \n"
      <<gameb[1][0]<<" | "<<gameb[1][1]<<" | "<<gameb[1][2]<<"\n"
      <<"-- --- --- \n"
      <<gameb[2][0]<<" | "<<gameb[2][1]<<" | "<<gameb[2][2]<<"\n";
  }
  else goto ifFail;
}
```

When dealing with large pieces of code, this step will be the most crucial step and will determine the efficiency of translation.

4. Translate the code using the language model: In this step, the C++ code is translated into Java. The smaller code blocks are fed into the model one at a time, prompt engineering is done and the model's output is used to create the equivalent Java code. This process is repeated until all the C++ code has been translated.
5. Review and refine the translated code: Once the code has been translated, it is reviewed to ensure that it is accurate and complete. The code is also refined by adjusting any syntax errors or inconsistencies and tested to ensure it works as intended.

Provided below are the statistics on the lines of code generated, edited, and deleted.

Table 63
Code statistics – Code Translation

# LOC generated	# Insertions	# Deletions	# Edits
800	35	15	20
% modification needed in code		~9%	

6. Optimize the translated code: The translated code is optimized by removing any redundant or inefficient code, and optimized for performance.
7. Compile and test the Java application: Finally, the Java application is compiled and tested thoroughly to ensure that it works as expected. Debugging of any issues that arise is done, and further optimizations are done as needed.

   For each of the above tasks, given below is the effort taken.

Effort Analysis

The following table shows the different tasks involved in the translation process using LLM and the total effort involved.

Table 64
Effort Analysis - Code Translation

Sr. No.	Task	Time (in hours)
1	Code analysis	0.5
2	Pre-processing	1.5
3	Code chunking	2
4	Prompt engineering and code translation	5
5	Code stitching and code correction	1
6	Code optimization	2
7	Code testing	2
Total Effort		**14**

Thus, the total effort to translate the application from C++ to Java is 14 hours.

<u>Manual</u>

The same task was given to each of the individual programmers in the novice, intermediate, and expert programmer group. Since this is the task of translation, the requirement here is that the developer must have an understanding of both languages C++ and Java. In the novice group, the programmer had a basic understanding of both programming languages while in the expert group, the programmer had a good understanding of both programming languages. At a high level, the following are the steps that were required to be done.

1. Analyze the C++ application

   Based on the expertise of the programmer, the effort needed in this step differs. For example, for a novice developer, this step involves considerable effort, as the novice developer needs to spend significant time learning the syntax and structure of the C++ language and understanding the application's code. However, the expert developer, being familiar with both C++ and Java can analyze the code more efficiently.

2. Convert the C++ code to Java:

   As in step 1, for a novice developer, this step requires significant effort as the novice developer needs to learn the syntax and structure of Java, and translating complex C++ code into Java could be challenging. However, the expert developer being familiar with both C++ and Java can translate the code more quickly and accurately.

3. Review and refine the translated code

   As in the other steps, identifying and correcting errors in the code can be challenging for a novice developer as compared to an intermediate or expert programmer who can identify and correct errors more efficiently.

4. Optimize the Java code:

This task requires an understanding of Java optimization techniques. For a novice developer, this will require considerable effort as compared to an expert programmer who would be familiar with various Java optimization techniques.

5. Compile and test the Java application:

   This step requires compiling, debugging, and testing the Java code. Again based on the expertise, the effort needed in this step will vary for each programmer.

Provided below is the analysis of the effort needed by each group to perform this task.

Effort Analysis

The following table shows the manual effort involved by individuals of different levels of expertise in performing the translation.

Table 65
Effort Analysis - Code Translation

	Effort (in hours)		
	Novice	**Intermediate**	**Expert**
Analyze Code	16	8	4
Translate Code	56	32	14
Refine code	12	8	4
Optimize code	10	5	3
Debug and Test	12	8	5
Total Effort	**106**	**61**	**30**

Effort saving

Table 66 below shows the effort savings obtained in performing the translation by leveraging AI as against the same task done manually.

Table 66
Effort Saving – Code Translation

	LLM	Novice	Intermediate	Expert
Time taken	14 hours	106 hours	61 hours	30 hours
*Effort saving if model is used**		92 hours	47 hours	16 hours
*Percentage effort saving**		87%	77%	53%

*Effort saving = Manual effort – LLM effort
*Percentage effort saving – Effort saving / Manual effort

Cost Estimate

Large language Model (Codex)

In the discussion to research question 1, it was observed that OpenAI Codex was capable of translation from one modern programming language to another. This model understands both C++ and Java. Hence, this model was used for translation.

Table 67
Cost of inferencing using Codex

# chunks passed to the model	12
Average #LOC translated in one pass	100
# of iterations of prompt engineering/fine-tuning	15
# Total tokens (translated + generated) including different passes	9,450,000 (2,250,000 tokens during experimentation + 7,200,000 during translation)
Cost of completion API	$0.02 per 1000 tokens
Total cost of inferencing*	$189

* Total cost = Total # of tokens * Cost of API / 1000

Labor cost

Assuming an hourly rate of $40 for AI resource in inferencing LLM and performing prompt engineering, the cost of generating code for this task is:

# of hours to complete the translation – 14

Hourly rate - $40

Cost of AI resource – 14 * 40 = $560

Cost of inferencing - $189

Total cost - $749

Manual

Assuming an hourly rate of $20 per hour for a novice programmer, $30 for an intermediate programmer, and $50 for an expert programmer, the cost of translating this application manually is as follows:

Table 68
Cost of manual translation

Novice programmer: 106 hours x $20/hour = $2120
Intermediate programmer: 61 hours x $30/hour = $1830
Expert classifiers: 30 hours x $50/hour = $1500

Cost saving

The following table shows the cost savings obtained by leveraging LLM for the task of code translation.

Table 69
Cost saving using LLM - Code Translation

	LLM (Zero-shot)	**Novice**	**Intermediate**	**Expert**
Cost	$749	$2120	$1830	$1500
*Cost saving**		$1371	$1081	$751
*Percentage cost saving**		65%	59%	50%

Overall

The following table shows the overall savings obtained using LLM as compared to performing the translation manually.

Table 70
Overall Savings - Code Translation

	LLM (Zero-shot)	Novice	Intermediate	Expert
% Effort saving		87%	77%	53%
% Cost saving		65%	59%	50%

As expected, the percentage of effort saving is highest when compared to a novice programmer. The percentage of effort saving decreases as the level of expertise increases. For example, the AI model can save 87% of the effort for novices, 77% for intermediate level, and 53% for experts. Similarly, for the novice programmer, the overall cost saving is 65%, which means that the AI model can perform the task with 65% less cost compared to the manual effort. Like effort saving, the percentage of cost saving decreases as the level of expertise increases. For example, the AI model can save 65% of the cost for novices, 59% for intermediate level, and 50% for experts.

4.4.2.3 Technical document generation

In Research Question 1, there was a brief mention of ChatGPT and the code generation and code understanding capabilities of this new model from OpenAI. In continuation to the code translation for the Tic-Tac-Toe game, we also tried to generate the technical documentation out of the original C++ code. Shown below is a snippet of the document generated using ChatGPT.

An important point to note is the capability of this model to identify that the two functions have a lot in common and can be combined. These insights from existing code can be very helpful, especially in tasks like code translation/migration. Identifying such technical debt in code and providing suggestions or resolutions for the same is one of the big advantages that models like Codex and ChatGPT provide.

Function Name: playerOne(board, p1, p2)

Description:

This function is responsible for the game play for player 1. It takes in the current state of the board, the names of the two players and it prompts player 1 for their move. Once the move is made, the board state is updated and the function returns `true`.

Input:

- `board`: a 3x3 character array representing the current state of the game board
- `p1`: a string representing the name of player 1
- `p2`: a string representing the name of player 2

Output:

- `true`: If the player 1's move was valid and the board was updated successfully

Steps:

1. Clear the console
2. Print the current state of the board to the console
3. Prompt player 1 for their move
4. If the move is valid:
   - Update the state of the board
   - Clear the console
   - Print the current state of the board to the console
   - Return `true`
5. If the move is invalid:
   - Print an error message and go back to step 3

Figure 39
playerOne Documentation

Function Name: playerTwo(board, p1, p2)

Description:

This function is responsible for the game play for player 2. It takes in the current state of the board, the names of the two players and it prompts player 2 for their move. Once the move is made, the board state is updated and the function returns `true`.

Input:

- `board`: a 3x3 character array representing the current state of the game board
- `p1`: a string representing the name of player 1
- `p2`: a string representing the name of player 2

Output:

- `true`: If the player 2's move was valid and the board was updated successfully

Steps:

1. Clear the console
2. Print the current state of the board to the console
3. Prompt player 2 for their move
4. If the move is valid:
    - Update the state of the board
    - Clear the console
    - Print the current state of the board to the console
    - Return `true`
5. If the move is invalid:
    - Print an error message and go back to step 3

Note:

The two functions have a lot of similarities and could potentially be combined into one function with an additional parameter specifying which player is making the move.

Figure 40
playerTwo Documentation

4.4.2.4 Legacy Code Documentation and Migration

Intent

The requirement is to generate documentation for an existing COBOL application and migrate it to Java. The application is a Human Resource Management System that maintains employee records and maintains all employee information like personal details, leave requests, transfer requests, payment information, and so on.

Two types of documentation were to be generated using the large language model:

1. Generic documentation which describes the intent of the piece of code, various variables or data structures used, and the functionality performed.
2. Design document – generated with the intent to re-architect the entire application

The next step was to generate the Java code for this application with the help of the generated documentation.

Effort

<u>Large Langue Model (gpt-3.5 turbo alias ChatGPT)</u>

The following tasks are involved when a large language model is leveraged for generating documentation from legacy code.

1. Code analysis

   The original COBOL application had 1300 lines of COBOL code. The total number of tokens in this COBOL application were around 28,000. The gpt-3.5 turbo model has a limitation of 4096 tokens (including input and generated output). Hence, analyzing the code and strategizing the chunking process to achieve the best results becomes a crucial step to perform this task.
2. Code cleaning

In this stage, the code is cleaned of all extra lines, spaces, and comments. This avoids sending unnecessary information to the model, reducing the total number of tokens being consumed and improving the chances of getting better results from the model.

3. Code chunking

The next step is to create logical chunks for the entire piece of code. The model is fed with these chunks of code to extract the documentation from it. This is a crucial step and has a great impact on the efficiency of translation.

For example, randomly selecting 12 lines of COBOL code as shown below:

```
RECORD KEY IS EPEMPID
  FILE STATUS IS FSEP.
DATA DIVISION.
FILE SECTION.
FD EMPFILE
  LABEL RECORDS ARE STANDARD
  VALUE OF FILE-ID IS "EMP.DAT".
01 EMPREC.
  02 EEMPID    PIC X(6).
  02 EEMPNAME  PIC X(25).
  02 EEMPADDR  PIC X(30).
```

and feeding them to the model would generate documentation as shown:

```
This piece of code defines the record key and file status for the EMPFILE. The record key is defined as
EPEMPID and the file status is defined as FSEP.

The DATA DIVISION and FILE SECTION are defined to specify the file structure.

The FD EMPFILE section describes the file and specifies the file identifier as "EMP.DAT". The LABEL
RECORDS ARE STANDARD clause indicates that each record in the file has a standard label.

The EMPREC section defines the record structure of the EMPFILE. It contains three fields: EEMPID,
EEMPNAME, and EEMPADDR. EEMPID is defined as a 6-character alphanumeric field, EEMPNAME is
```

defined as a 25-character alphanumeric field, and EEMPADDR is defined as a 30-character alphanumeric field.

This code is used to define the file structure and record structure for a file named EMP.DAT, which contains employee information. It is written in COBOL programming language, commonly used for business applications.

On the other hand, providing a logical block of code as shown below:

```
DATA DIVISION.
FILE SECTION.
FD EMPFILE
   LABEL RECORDS ARE STANDARD
   VALUE OF FILE-ID IS "EMP.DAT".
01 EMPREC.
   02 EEMPID    PIC X(6).
   02 EEMPNAME  PIC X(25).
   02 EEMPADDR  PIC X(30).
   02 EPHONE    PIC X(10).
   02 EDOJ      PIC X(10).
   02 EDIP      PIC X(10).
   02 EUG       PIC X(4).
   02 EPG       PIC X(4).
   02 EPROFQ    PIC X(4).
   02 ESKILL    PIC X(10).
   02 EGRDNO    PIC 99.
   02 EBRNID    PIC X(6).
   02 EDESID    PIC X(6).
```

produces coherent documentation like:

This piece of code is a COBOL program that defines the data structure and file descriptor for an employee data file. The program is designed to read and write data to a file named "EMP.DAT".

The DATA DIVISION specifies the file section, which contains the file descriptor (FD) for the EMPFILE. The LABEL RECORDS ARE STANDARD clause indicates that the record format is fixed-length, with each record having the same size. The VALUE OF FILE-ID IS "EMP.DAT" clause specifies the name of the file.

The FILE SECTION is followed by the data definition section, which defines the structure of the EMPREC record. The EMPREC record contains 14 fields that correspond to various employee details such as name, address, phone number, date of joining, department ID, project ID, professional qualification, skill, grade number, branch ID, and designation ID.

Each field is defined using a Picture Clause (PIC) that specifies the data type and length of the field. For example, the EEMPID field is defined as a string of six characters using the PIC X(6) clause, while the EGRDNO field is defined as an integer with two digits using the PIC 99 clause.

Overall, this program provides a data structure and file descriptor for an employee data file, which can be used for data input/output operations in a COBOL program.

4. Prompt-engineering

   After chunking the code into logical blocks, each block is fed to the model and appropriate instructions are given to the model to generate the documentation. Prompt engineering is done to come up with the best prompt which works well on all chunks of code. Zero-shot and one-shot techniques are experimented with and a final prompt and technique is decided to perform this task.

   Steps 3 and 4 are performed repeatedly till the best technique for chunking the code is devised along with the prompt for it.

   As discussed above, the large language model was used to generate two kinds of documentation. The first is generic documentation to understand the intent of the code and get key insights from it. Figure 41 below shows the generic documentation that is generated for COBOL code which is added in Appendix B (Figure 115 and Figure 116).

Generated documentation

This piece of code is written in COBOL programming language and it contains several paragraphs for different functionalities related to an HRMS (Human Resource Management System) application. Each paragraph is responsible for a specific task, such as reading and displaying information from different input files.

Let's go through each of the paragraphs and understand their functionality:

1. CONFIRMATION-PARA: This paragraph is responsible for reading the confirmation file and displaying the confirmation code, employee code, and confirmation date on the screen. It accepts the confirmation code as input from the user and uses it to read the corresponding record from the confirmation file. If the record is not found, it jumps to the ERROR-CONFIRMATION-PARA paragraph.

2. GRADE-PARA: This paragraph is responsible for reading the grade file and displaying the grade number and designation for the given grade number. It accepts the grade number as input from the user and uses it to read the corresponding record from the grade file. It uses the GR-READ-PARA paragraph to read all the records until the end of the file is reached.

3. TRANSFER-PARA: This paragraph is responsible for reading the transfer file and displaying the transfer code, employee code, old branch code, and transfer date on the screen. It accepts the transfer code as input from the user and uses it to read the corresponding record from the transfer file. If the record is not found, it jumps to the ERROR-TRANSFER-PARA paragraph.

4. EMPPERSONAL-PARA: This paragraph is responsible for reading the employee personal file and displaying various personal details of the employee such as the temporary address, phone number, date of birth, blood group, etc. It accepts the employee code as input from the user and uses it to read the corresponding record from the employee personal file. If the record is not found, it jumps to the ERROR-EMPPERSONAL-PARA paragraph.

5. ERROR-EMP-PARA: This paragraph is responsible for handling errors related to the employee file. It closes the employee file and displays an error message on the screen.

6. ERROR-LEAVE-PARA: This paragraph is responsible for handling errors related to the leave file. It closes the leave file and displays an error message on the screen.

7. ERROR-BRANCH-PARA: This paragraph is responsible for handling errors related to the branch file. It closes the branch file and displays an error message on the screen.

8. ERROR-DEPARTMENT-PARA: This paragraph is responsible for handling errors related to the department file. It closes the department file and displays an error message on the screen.

Overall, this piece of code reads and displays information from different input files based on the user's input. It also handles errors related to file reading and displays appropriate error messages on the screen.

Figure 41
Generated Generic documentation

For the same piece of code, Figure 42 below shows the design documentation that is generated

Design Document for HRMS Read Menu

Introduction
The HRMS (Human Resource Management System) Read Menu is a COBOL-based application that enables users to read and display information from different input files. This document outlines the design of the application, including the functional requirements, system architecture, and data flow.

Functional Requirements
The HRMS Read Menu application has the following functional requirements:
- Read and display information from the confirmation file based on the confirmation code provided by the user.
- Read and display grade number and designation from the grade file based on the grade number provided by the user.
- Read and display transfer code, employee code, old branch code, and transfer date from the transfer file based on the transfer code provided by the user.
- Read and display various personal details of the employee from the employee personal file based on the employee code provided by the user.
- Handle errors related to file reading and display appropriate error messages on the screen.

System Architecture
The HRMS Read Menu application is designed using a modular approach to improve maintainability and scalability. It consists of several paragraphs, each responsible for a specific task. The main paragraphs are as follows:
- CONFIRMATION-PARA: Reads and displays information from the confirmation file.
- GRADE-PARA: Reads and displays information from the grade file.
- TRANSFER-PARA: Reads and displays information from the transfer file.
- EMPPERSONAL-PARA: Reads and displays personal details of the employee from the employee personal file.
- ERROR-EMP-PARA: Handles errors related to the employee file.
- ERROR-LEAVE-PARA: Handles errors related to the leave file.
- ERROR-BRANCH-PARA: Handles errors related to the branch file.
- ERROR-DEPARTMENT-PARA: Handles errors related to the department file.

Data Flow
The HRMS Read Menu application reads and displays information from different input files. Each paragraph is responsible for opening the respective input file, reading the required information based on the user's input, and displaying it on the screen. If the file reading fails due to invalid input or any other error, the application jumps to the appropriate error handling paragraph.

The data flow of the application can be summarized as follows:
- The user selects the desired function from the HRMS Read Menu.
- The application prompts the user to provide the required input, such as confirmation code, grade number, transfer code, or employee code.
- The application opens the respective input file and reads the required information based on the user's input.
- The application displays the information on the screen.
- If any error occurs during the file reading, the application jumps to the appropriate error handling paragraph and displays an error message on the screen.

Conclusion
The HRMS Read Menu application is a simple, yet effective COBOL-based application that enables users to read and display information from different input files. The modular design and data flow ensure easy maintainability and scalability of the application.

Figure 42
Generated Design documentation

Further, it also generates the proposed solution for migration to Java as shown in Figure 43 below.

Proposed Solution:

The given code appears to be a COBOL program and needs to be migrated to Java. The proposed solution would be to create a Java program that would perform the same functionality as the COBOL program. The Java program would use classes to represent the different entities such as Employee, Grade, Transfer, and EmployeePersonal, and methods to perform the different operations such as reading from files and displaying data on the screen.

Class Design:

1. Employee:
- Fields: empId, empName, empDept, confirmationCode, confirmationDate
- Methods: readEmployee(), displayEmployee()

2. Grade:
- Fields: gradeNo, designation
- Methods: readGrade(), displayGrade()

3. Transfer:
- Fields: transferCode, empCode, oldBranchCode, transferDate
- Methods: readTransfer(), displayTransfer()

4. EmployeePersonal:
- Fields: empCode, tempAddress, phone, dob, pob, langKnown, bloodGroup, weight, height, vision, fatherName, fatherDob, motherName, motherDob, spouseName, childName, childDob
- Methods: readEmployeePersonal(), displayEmployeePersonal()

Method Design:

1. readEmployee():
- Opens the input file for employee data
- Prompts the user to enter the employee code
- Reads data from the file for the given employee code, including the confirmation code and confirmation date
- Closes the input file

2. displayEmployee():
- Displays the employee details on the screen, including the confirmation code and confirmation date

3. readGrade():
- Opens the input file for grade data
- Prompts the user to enter the grade number
- Reads data from the file for the given grade number
- Closes the input file

4. displayGrade():
- Displays the grade details on the screen

5. readTransfer():
- Opens the input file for transfer data
- Prompts the user to enter the transfer code
- Reads data from the file for the given transfer code
- Closes the input file

6. displayTransfer():
- Displays the transfer details on the screen

7. readEmployeePersonal():
- Opens the input file for employee personal data
- Prompts the user to enter the employee code
- Reads data from the file for the given employee code
- Closes the input file

8. displayEmployeePersonal():
- Displays the employee personal details on the screen

9. errorHandler():
- Closes the file that caused an error
- Displays an error message on the screen

Overall, the updated Java program would be designed to read data from files, display it on the screen, and handle errors in case of invalid input. The program would also include the confirmation code and confirmation date for each employee.

Figure 43
GPT-3 proposed solution

5. Document Review

   The document is reviewed to ensure that it reflects the functionality of the code and uses a similar style of documentation for all chunks of code.

6. Document Organization

   The next step is to organize and if required, consolidate the documentation generated for different chunks of code

7. Code Migration to Java

   This step involves multiple sub-steps:

   a. Analyzing documentation: Understand the documentation and identify the chunking logic for generating Java code. This is crucial as for generating equivalent Java code for a piece of COBOL code, it is important to include the entire context (different data structures, file systems, and other dependencies) for that piece of COBOL code.

   b. Chunking the code: Based on the understanding from the documentation, the COBOL code is chunked into logical blocks of code by including the necessary context and dependencies.

As in the case of documentation, this is the most crucial step and plays a critical role in the kind of output that will be generated. For example, randomly selecting a few lines of code as shown below:

```
 DISPLAY(7 1) " UG                :" EUG.
 DISPLAY(8 1) " PG                :" EPG.
 DISPLAY(9 1) " PROFESSIONAL QUALITY :" EPROFQ.
 DISPLAY(10 1)" SKILL SET          :" ESKILL.
 DISPLAY(11 1)" GRADE NUMBER       :" EGRDNO.
 DISPLAY(12 1)" BRANCH CODE        :" EBRNID.
 DISPLAY(13 1)" DESIGNATION CODE   :" EDESID.
 CLOSE EMPFILE.
 DISPLAY(20 10)
  "PRESS ENTER TO RETURN TO HRMS READ MENU".
 STOP ' '.
 GO TO MAIN-PARA.
LEAVE-PARA.
 DISPLAY(1 1) ERASE.
 OPEN INPUT LEAVEFILE.
 DISPLAY "ENTER CODE :".
 ACCEPT LEMPID.
 DISPLAY(1 1) ERASE.
 READ LEAVEFILE INVALID KEY GO TO ERROR-LEAVE-PARA.
 DISPLAY(1 1) " CODE          :" LEMPID.
```

generates documentation and code as:

> This code is written in COBOL language and it displays employee details such as UG (undergraduate) qualification, PG (postgraduate) qualification, professional quality, skill set, grade number, branch code, and designation code. It then closes the employee file and displays a message to return to the HRMS read menu.
>
> The second part of the code is for handling employee leaves. It opens an input file for leave records, accepts employee code as input, reads the leave file, and displays the employee code. If the employee code is not found in the file, it goes to an error handling section.

Java equivalent code:

```
// Display employee details
System.out.println("UG                :" + EUG);
System.out.println("PG                :" + EPG);
System.out.println("PROFESSIONAL QUALITY :" + EPROFQ);
System.out.println("SKILL SET         :" + ESKILL);
System.out.println("GRADE NUMBER      :" + EGRDNO);
System.out.println("BRANCH CODE       :" + EBRNID);
System.out.println("DESIGNATION CODE  :" + EDESID);
EMPFILE.close();
System.out.println("PRESS ENTER TO RETURN TO HRMS READ MENU");
Scanner scanner = new Scanner(System.in);
scanner.nextLine(); // Wait for user to press enter
scanner.close();
MAIN-PARA();
// Handle employee leaves
System.out.println("ENTER CODE :");
Scanner scanner = new Scanner(System.in);
String LEMPID = scanner.nextLine();
scanner.close();
LEAVEFILE.openInput();
System.out.println("CODE        :" + LEMPID);
LEAVEFILE.read();
LEAVEFILE.close();
if (LEAVEFILE.isInvalid()) {
   ERROR-LEAVE-PARA();
}
```

However, by utilizing the understanding from the generated documentation, it can be inferred that LEAVE-PARA contains information about employee leave.

By including all the necessary context and dependencies around LEAVE-PARA in the block of code during chunking as shown below:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. EMPREAD.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
    SELECT LEAVEFILE ASSIGN TO DISK
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
```

```
    RECORD KEY IS LEMPID
    FILE STATUS IS FSL.

DATA DIVISION.
FILE SECTION.
FD LEAVEFILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "LEAVE.DAT".
01 LEAVEREC.
    02 LEMPID    PIC X(6).
    02 LFMDATE   PIC X(10).
    02 LTODATE   PIC X(10).
    02 LLEVCAT   PIC X(3).

LEAVE-PARA.
    DISPLAY(1 1) ERASE.
    OPEN INPUT LEAVEFILE.
    DISPLAY "ENTER CODE :".
    ACCEPT LEMPID.
    DISPLAY(1 1) ERASE.
    READ LEAVEFILE INVALID KEY GO TO ERROR-LEAVE-PARA.
    DISPLAY(1 1) " CODE          :" LEMPID.
    DISPLAY(2 1) " DATE          :" LFMDATE.
    DISPLAY(3 1) " DATE          :" LTODATE.
    DISPLAY(4 1) " LEAVE CATEGORY :" LLEVCAT.
    CLOSE LEAVEFILE.
    DISPLAY(20 10)
      "PRESS ENTER TO RETURN TO HRMS READ MENU".
    STOP ' '.
    GO TO MAIN-PARA.
```

improves the quality of output generated as shown below:

> This code is a COBOL program that reads a file called "LEAVE.DAT" using an indexed access mode and a dynamic organization. The file contains records with four fields: LEMPID, LFMDATE, LTODATE, and LLEVCAT. The program prompts the user to enter a code (LEMPID) and displays the corresponding record's fields on the screen. If the record is not found, the program goes to a label called ERROR-LEAVE-PARA. After displaying the record, the program prompts the user to press Enter to return to the HRMS READ MENU.

GPT-3 Generated Java code:

```
import java.io.*;
import java.util.Scanner;

public class LeaveFileReader {
   private final String FILENAME = "LEAVE.DAT";
   private RandomAccessFile file;
   private Scanner input;

   public LeaveFileReader() throws FileNotFoundException {
      file = new RandomAccessFile(FILENAME, "r");
      input = new Scanner(System.in);
   }

   public void readRecord() throws IOException {
      String LEMPID, LFMDATE, LTODATE, LLEVCAT, record;
      boolean found = false;
      System.out.print("ENTER CODE: ");
      LEMPID = input.nextLine();
      while ((record = file.readLine()) != null) {
         String[] fields = record.split("\\s+");
         if (fields[0].equals(LEMPID)) {
            LFMDATE = fields[1];
            LTODATE = fields[2];
            LLEVCAT = fields[3];
            System.out.println("CODE          : " + LEMPID);
            System.out.println("DATE          : " + LFMDATE);
            System.out.println("DATE          : " + LTODATE);
            System.out.println("LEAVE CATEGORY : " + LLEVCAT);
            found = true;
            break;
         }
      }
      if (!found) {
         System.out.println("Record not found.");
      }
   }

   public void close() throws IOException {
      file.close();
   }
}
```

The above code compiles without any errors, clearly indicating the improvement in the quality of generated code.

c. Prompt-engineering: Next step is to perform prompt-engineering to translate the chunks of COBOL code into equivalent Java.

d. Code correction: The generated code is copied into a Java IDE and corrections are done, wherever required.

For the 1400 lines of COBOL code, 25 Java classes were generated. The following table gives the statistics on the lines of Java code generated, edited, and deleted.

Table 71
Code statistics - COBOL to Java

# LOC generated	**# Insertions**	**# Deletions**	**# Edits**
1570	80	50	175
% Modification needed in code		~20%	

e. Code compilation and testing: The final migrated code is compiled and tested to ensure that the migrated Java application achieves the desired functionality. 23 out of 25 generated Java classes compiled without errors. The corrections were needed mostly to edit the class names to match the calling program. Few insertions were needed to include the user input in the class rather than receiving it as input parameters.

Effort Analysis

The following two tables depict the effort involved in performing the two tasks i.e. Legacy Code Documentation (Generic documentation and Design documentation generation) and Code migration, performed as part of this experiment.

Legacy Code Documentation (Generic and Design Document)

Table 72
Effort Analysis - Legacy Code Documentation

Sr. No.	Task	Time (in hours)
1	Code analysis and understanding	2
2	Code cleaning	2
3	Code chunking*	8
4	Prompt engineering and generation of code documentation*	16
5	Document Review	4
6	Document Organization	2
Total Effort		**34**

*These tasks are performed repeatedly till an optimal solution is identified

COBOL to Java migration

Table 73
Effort Analysis - COBOL to Java migration

Sr. No.	Task	Time (in hours)
1	Analyzing generated documentation*	4
2	Code chunking*	12
3	Prompt engineering and code generation	24
4	Code review and correction	12
5	Code compilation and testing	8
Total Effort		**60**

*These tasks are performed repeatedly till an optimal solution is identified

Thus, the total effort to generate documentation for the legacy COBOL application is 34 hours, and migrating the application to Java is 60 hours.

Manual

The same task was given to each of the individual programmers in the novice, intermediate, and expert programmer group. For the task of generating documentation, the group consisted of programmers with basic, intermediate, and expert understanding of COBOL. However, for the task of migrating to Java, knowledge of both COBOL and Java was required. This was challenging, primarily due to the difference in the programming paradigms between COBOL and Java. Also, these languages differ in their syntax, keywords, data types, and other programming concepts.

COBOL is a procedural language while Java is an object-oriented language. For a programmer skilled in one modern programming language like C++, it is comparatively easy to also be an expert in another modern programming language like Java. However, for a programmer who is an expert in a procedural language like COBOL, mastering the skills of an object-oriented language like Java may be challenging. Hence, for the task of migration to Java, the intermediate and expert groups had a developer which had the desired expertise in Java but only basic knowledge of COBOL. Thus, in the novice group, the programmer had a basic understanding of both source and target programming languages while in the expert group, the programmer had a good understanding of Java but only a basic understanding of COBOL.

At a high level, the following are the steps that were required to be done.

Code Documentation (Generic documentation and Design document)

1. Understanding the COBOL application

   Before starting documentation, the developer needs to understand the COBOL application thoroughly. This includes understanding the program's logic, file structures, input/output formats, and other critical components. The developer needs to review the COBOL code line by line and understand the purpose of each statement. The developer also needs to identify the program's data structures and understand how they are used throughout the program.

2. Identifying documentation requirements

Once the developer has a good understanding of the COBOL application, they need to identify the documentation requirements. This includes determining what needs to be documented, such as the program's flow, data structures, input/output formats, and processing logic. The developer needs to decide which parts of the COBOL program require documentation and how detailed the documentation needs to be.

3. Creating a generic documentation

   The developer needs to create generic documentation that outlines the COBOL application's overall functionality, data structures, input/output formats, and processing logic. This documentation should provide an overview of the program and its purpose. It should also describe the program's data structures and how they are used.

4. Creating a design document

   Based on the generic documentation, the developer needs to create a design document that outlines the detailed technical specifications of the COBOL application. The design document should describe the processing logic and the steps involved in executing the program. It should also provide details about the input/output formats and the program's data structures. The document should also include a proposed solution for migrating the code to Java.

Code Migration (COBOL to Java)

1. Writing equivalent Java Code

   With the understanding of the COBOL code gathered in Step 1 of producing documentation of the COBOL code, and the produced documentation and design, the developer can start the process of writing the equivalent Java code. The developer will need to convert the COBOL program's processing logic, data structures, and input/output formats to Java. The developers in all the groups, including the expert and intermediate programmers, may only have a basic understanding of the source language while having the desired expertise in the target language. Hence, they may need to refer to additional resources to gain an understanding of the COBOL syntax and structure.

2. Testing the Java code

   Once the Java code is written, it needs to be tested to ensure that it produces the same output as the original COBOL program.

3. Debugging the Java code

   During testing, the developer may encounter errors or bugs in the Java code. Debugging involves identifying and fixing these errors to ensure that the Java code produces the correct output.

Table 74 below details the effort needed by each group to perform the task of documenting the legacy code.

Table 74
Effort Analysis - Manual code documentation

	Effort (in hours)		
	Novice	**Intermediate**	**Expert**
Code Understanding	40	30	20
Documentation Requirement Identification	20	15	10
Generic documentation	30	20	10
Design document generation	50	35	25
Total Effort	**140**	**100**	**65**

Similarly, Table 75 below details the effort needed by each group to migrate the code to Java.

Table 75
Effort Analysis - Manual code migration

	Effort (in hours)		
	Novice	**Intermediate**	**Expert**
Writing equivalent Java code	110	70	50
Testing Java code	40	30	20
Debugging Java code	30	20	10
Total Effort	**180**	**120**	**80**

Effort saving

The following table shows the effort saving obtained when performing the task of code documentation by leveraging an AI model, as against the same task done manually.

Table 76
Effort Saving - Code Documentation

	LLM	**Novice**	**Intermediate**	**Expert**
Time taken	34 hours	140 hours	100 hours	65 hours
*Effort saving if model is used**		106 hours	66 hours	31 hours
*Percentage effort saving**		76%	66%	48%

*Effort saving = Manual effort – LLM effort
*Percentage effort saving – Effort saving / Manual effort

Similarly, the table below shows the effort saving in the code generation task.

Table 77
Effort Saving - Code Generation

	LLM	**Novice**	**Intermediate**	**Expert**
Time taken	60 hours	180 hours	120 hours	80 hours
*Effort saving if model is used**		120 hours	60 hours	20 hours
*Percentage effort saving**		67%	50%	25%

*Effort saving = Manual effort – LLM effort
*Percentage effort saving – Effort saving / Manual effort

The table below shows the total effort saving obtained in the end-to-end process.

Table 78
Total Effort saving in end-to-end process

	LLM	Novice	Intermediate	Expert
Time taken	94 hours	320 hours	220 hours	145 hours
*Effort saving if model is used**		226 hours	126 hours	51 hours
*Percentage effort saving**		71%	57%	35%

Cost Estimate

Large Langue Model (gpt-3.5 turbo alias ChatGPT)

The model that was used to perform this task of COBOL documentation and COBOL to Java migration was gpt-3.5 turbo (ChatGPT). The pricing for this model is $0.002 / 1K tokens.

Code documentation (Generic documentation + Design Document)

Table 79
Cost of inferencing - documentation

# LOC per sample used for prompt-engineering	150
# Tokens in input + generated documentation	4096
# of iterations of prompt engineering for generic documentation	15
# of iterations of prompt engineering for design document generation	25
Total # of tokens consumed during prompt engineering for both steps	143360 (generic) + 225280 (design) = 368640
# chunks passed to the model	33
# times response regenerated	3
# Total tokens (input + generated documentation) including different experiments	774,144 (368,640 tokens during experimentation + 405,504 tokens for documentation)
Cost of completion API	$0.002 per 1000 tokens
Total cost of inferencing*	$1.54

* Total cost = Total # of tokens * Cost of API / 1000

Labor cost

Assuming an hourly rate of $40 for AI resource in inferencing LLM and performing prompt engineering, the cost of generating both documentation for this task is:

# of hours to generate both generic and design documentation – 34

Hourly rate - $40

Cost of AI resource - $1360

Cost of inferencing - $1.54

Total cost - $1362

Code migration (COBOL code and documentation to Java)

Table 80
COBOL code and documentation to Java

# LOC used per sample for prompt-engineering	150
# Tokens in input + generated code	4096
# of iterations of prompt engineering for generating Java code	45
Total # of tokens consumed during prompt-engineering	184,320
# chunks passed to the model	33
# times response regenerated	3
# Total tokens (input + generated code) including different experiments	589,824 (184,320 tokens during experimentation + 405,504 during code generation)
Cost of completion API	$0.002 per 1000 tokens
Total cost of inferencing*	$1.18

* Total cost = Total # of tokens * Cost of API / 1000

Labor cost

With the same assumptions as above, the cost of generating Java code for this task is:

# of hours to generate Java code – 60

Hourly rate - $40

Cost of AI resource - $2400

Cost of inferencing - $1.18

Total cost - $2401

Manual

Assuming an hourly rate of $20 per hour for a novice programmer, $30 for an intermediate programmer, and $50 for an expert programmer, the cost of generating documentation for this application manually is as follows:

Table 81
Cost of manual documentation

Novice programmer: 140 hours x $20/hour = $2800
Intermediate programmer: 100 hours x $30/hour = $3000
Expert classifiers: 65 hours x $50/hour = $3250

Cost saving (for documentation)

The following table shows the cost saving obtained in the documentation stage, by leveraging LLM.

Table 82
Cost saving documentation

	LLM (Zero-shot)	**Novice**	**Intermediate**	**Expert**
Cost	$1362	$2800	$3000	$3250
*Cost saving**		$1438	$1638	$1888
*Percentage cost saving**		51%	55%	58%

Similarly, the cost of generating code manually is as follows:

Table 83
Cost of manual code generation

Novice programmer: 180 hours x $20/hour = $3600
Intermediate programmer: 120 hours x $30/hour = $3600

Expert classifiers: 80 hours x $50/hour = $4000

Cost saving (for code generation)

The following table shows the cost saving obtained in the code generation stage, by leveraging LLM.

Table 84
Cost saving for code generation

	LLM (Zero-shot)	Novice	Intermediate	Expert
Cost	$2401	$3600	$3600	$4000
*Cost saving**		$1199	$1199	$1599
*Percentage cost saving**		33%	33%	40%

Total cost saving for the end-end process

The following table shows the total cost saving obtained in the end-to-end process.

Table 85
End-to-end Cost saving

	LLM (Zero-shot)	Novice	Intermediate	Expert
Cost	$3763	$6400	$6600	$7250
*Cost saving**		$2637	$2837	$3487
*Percentage cost saving**		41%	43%	48%

Overall

The following table shows the overall savings obtained in the end-to-end process when leveraging AI for the task of legacy code migration, as compared to the same task being performed manually by programmers of different levels of expertise.

Table 86
Overall savings (end-to-end process)

	LLM (Zero-shot)	Novice	Intermediate	Expert
% Effort saving		71%	57%	35%

% Cost saving		41%	43%	48%

As observed in the previous cases, ChatGPT model proves to be effective in minimizing effort and cost in both code documentation and code migration. As the skill level increases from Novice to Expert, there is a noticeable decline in the percentage of effort savings and an increase in cost savings. The effort saving when using an ML model versus the same task being performed by a novice programmer is 71% and the cost saving is 41%. However, as the skill level changes from novice to expert, the effort saving is only 35% and the cost saving is 48%. This indicates that using an ML model does not provide a significant advantage against an expert programmer in terms of effort saving. This is because an expert programmer would take only slightly more time than the time taken to perform the task using an ML model. However, in terms of cost saving, the benefit of using an ML model as against an expert programmer is almost 50%.

The complexity of the program that was considered for this experiment was relatively simple. It is expected that as the complexity increases, the number of activities and effort involved in using an ML model which is not particularly trained in a programming language like COBOL, to perform the task of migration, will be considerably higher. Based on the expertise of the programmers available to perform this task manually, a thorough cost-benefit analysis would need to be performed before employing an ML model for the same.

4.4.2.5 Complex tasks

In this section, the objective is to highlight certain code automation tasks that are still challenging and may require significant effort when performed using a large language model.

1. Generating code for microservices architecture

In a microservices architecture, different services have complex dependencies and communication patterns. Large language models still struggle to generate code for these scenarios, as providing them with the complete context to understand the relationships between services can be challenging due to the context length limitations of these models. Similarly, handling the intricate details of service-to-service communication stays a challenge for these models.

2. Framework-specific code generation

   When generating code for specific frameworks, such as React or Angular, large language models might struggle to produce code that adheres to best practices, patterns, and conventions specific to that framework. This is because the model's training data may not include enough examples or the latest practices of these frameworks.

3. Complex build systems

   For large software projects that rely on intricate build systems, such as Gradle or Maven, large language models may have difficulty generating accurate build configurations or managing dependencies across multiple sub-projects. This is due to the complexity and context-specific nature of these systems.

4. Code generation for embedded systems

   Embedded systems often have strict requirements regarding memory usage, performance, and hardware constraints. Large language models are not yet mature enough to generate efficient code tailored to these constraints, as they lack the context to understand and optimize for specific hardware and software requirements.

When exploring the possibility of using large language models for such tasks, it is important to evaluate the amount of effort that would be required in data preparation and contextualizing the models, so that they can provide some effort savings. Based on the size of the codebase and the criticality of the requirement, a detailed analysis of the various tasks required to get the maximum benefit from the model should be done.

4.5 Research Question Four

What types of jobs will be completely replaced, if any, due to the adoption of AI in the software industry?

We hypothesized that:

Hypothesis 5

If code automation using large language models is adopted in the software industry, it can improve software development flexibility by allowing developers to focus on higher-level tasks and by providing more options for code generation.

Based on the exhaustive analysis and set of experiments that were performed, we analyzed the impact the adoption of these large language models may have on various jobs in the software industry. In our research, we observed the following jobs that can be augmented with the help of large language models:

1. Planning and Requirements

   We discussed in section 4.2.1.to 4.2.3, how large language models can augment different activities in the planning and requirements stage. They can help in software requirement classification and clustering; they can automate tracing the requirement to the design; detecting similarity between a given set of requirements; extracting requirements from unstructured sources and detecting ambiguity from SRS documents.

   We did a detailed study of the efficiency, cost, and effort saving we get by leveraging large language models in tasks like requirement classification. We observe that though the models can perform different tasks in this phase, manual validation will always be required and human cannot be eliminated or replaced for this task.

They do however provide a significant effort saving and can free up the bandwidth of the person to work on higher-order decision-making tasks.

For example, in the requirements gathering phase, the person who would earlier manually classify the requirements could now leverage the requirement classification experience to perform the following tasks during the Requirements Analysis phase:

- Analyzing requirements to identify potential gaps or areas of ambiguity.
- Working with stakeholders to clarify and refine requirements.
- Prioritizing requirements based on business value and technical feasibility.
- Defining the scope of the project based on the requirements.
- Developing use cases, scenarios, and other artifacts to further refine the requirements.
- Collaborating with the implementation team to make sure that the requirements are clear and actionable.

By focusing on these higher-order decision-making tasks related to requirements, the person can ensure that the software development project stays on track and delivers a product that meets the needs of the stakeholders.

2. Design

We also observe how large language models can be leveraged in the design phase for tasks like action extraction from requirements; design pattern recognition and classification; class extraction and designing websites.

As in the previous cases, we observe that LLM can only augment and not replace a human in these tasks. The creation of design documents often requires human expertise and judgment to ensure that they accurately represent the design of the software system. Additionally, individuals involved in the design phase also play a key role in ensuring that the design meets the functional and non-functional requirements of the software system and that it is scalable, maintainable, and secure.

3. Coding

   We observed during the various code automation tasks that we attempted during the previous phase, that these large language models can surely help reduce effort by providing code suggestions or snippets for specific tasks. However, we also observed that they do not always guarantee correctness. Human developers remained essential for reviewing the AI-generated code, verifying its correctness, and integrating it into the existing codebase.

   Similarly, by automating certain code-related tasks, the freed-up bandwidth of the programmer could be used for:

- Architecture and Design: The person could focus on designing the overall architecture of the application, including selecting the appropriate design patterns, frameworks, and technologies. They could also develop high-level design specifications that guide the development team in implementing the application.
- Testing and Quality Assurance: The person could focus on ensuring that the application meets the required quality standards and is free from defects. They could develop test plans and test cases, perform testing activities, and work closely with the development team to identify and resolve any issues that arise.
- Performance Optimization: The person could focus on optimizing the performance of the application by identifying potential bottlenecks and developing strategies to improve performance. This could involve analyzing code, configuring servers and databases, and developing caching and optimization strategies.
- Security and Compliance: The person could focus on ensuring that the application is secure and complies with relevant regulations and standards. They could perform security assessments, implement security controls, and develop compliance policies and procedures.

4. Software testing and quality assurance

We observed in Research Question One that large language models (LLMs) can be employed to automate test case generation, complementing traditional AI-based testing tools. By leveraging their natural language understanding capabilities and vast knowledge of programming languages, LLMs can generate test cases based on given requirements and conditions.

We have seen that:

- LLMs can understand human-readable requirements and convert them into test cases, streamlining the process of writing test cases from scratch. Developers and testers can provide high-level descriptions of desired test scenarios, and the LLM can generate the corresponding test cases in the appropriate programming language.
- LLMs can generate test cases that cover a wide range of scenarios, including functional, integration, and system testing. By understanding the codebase and the intended functionality, LLMs can create test cases that address different aspects of the software, helping to ensure comprehensive testing.
- As LLMs are trained on vast amounts of code, they can adapt to changes in the codebase, generating new test cases when updates are made. This enables a more agile testing process, as the LLM can quickly produce relevant test cases when modifications occur.
- LLMs can generate test cases that address edge cases and negative testing scenarios. By identifying potential failure points, LLMs can create tests that target these vulnerabilities, ensuring a more robust and reliable software product.

However, there are still limitations and challenges associated with using LLMs for automated test case generation:

- Interpretation of AI-generated test cases

Human expertise is necessary for reviewing and interpreting the test cases generated by LLMs, as the models may produce test cases that are not relevant, incorrect, or incomplete. Testers must understand the implications of the generated test cases and make informed decisions on which ones to use and how to address any issues that arise.

- Ensuring comprehensive test coverage

  While LLMs can generate a variety of test cases, they may not always anticipate every possible scenario. Human testers play a critical role in designing and implementing test strategies that cover a wide range of scenarios, including those that the LLM might not have generated.

- Quality assurance and maintenance

  Testers must ensure that the AI-generated test cases align with best practices and meet the project's quality standards. They also need to maintain and update the test cases as the software evolves, addressing any new requirements or issues that emerge.

Thus, LLMs can significantly improve the process of automated test case generation in software testing. However, human expertise remains vital in interpreting AI-generated test cases, ensuring comprehensive test coverage, and maintaining quality standards. When LLMs and human testers work together, they can create a more effective and efficient testing process, ultimately resulting in more robust software products.

5. Supporting code refactoring and optimization

   We observed in the previous sections that large language models can analyze codebases to identify inefficiencies, suggest improvements, and even generate optimized code. However, their suggestions may not always align with best practices or project requirements. Human developers with domain knowledge are needed to review AI-generated recommendations, taking into account factors like maintainability, scalability, and performance.

6. Enhancing technical documentation

   We also observed that large language models can help generate technical documentation, which can help technical writers by providing a starting point for creating user guides, API documentation, and other materials. AI models can extract information from code and produce coherent, well-structured content. However, human writers are still needed to ensure that the documentation is accurate, clear, and tailored to the target audience. They can also add valuable context and insights that AI models may not be able to infer from the code alone.

7. Teaching programming language

   As we conducted various experiments, we realized the impact of these AI models in helping developers learn new programming languages, frameworks, and libraries by providing examples and explanations. These models can thus help in expanding the skill sets of the programmers and thus improve overall productivity.

   This is especially true when generating documentation from legacy languages like COBOL. The large language model, especially ChatGPT, can explain in detail what the piece of code does which helps expand the knowledge base of the developer.

In summary, while AI adoption in the software industry offers significant benefits in terms of productivity and efficiency, it does not render human expertise obsolete. Instead, AI serves as a powerful tool that augments human capabilities, enabling software professionals to focus on higher-level tasks and achieve greater flexibility in software development. The true potential of AI in the software industry will be realized when humans and AI work together, leveraging the strengths of both to create better software solutions.

4.6 Research Question Five

What is the new set of skills that will emerge as AI is increasingly adopted across the organization for software development?

There are two hypotheses that we consider:

Hypothesis 6

If code automation using large language models is adopted in the industry, then reskilling software professionals is necessary.

Hypothesis 7

If code automation continues to increase in popularity and effectiveness, it will have a significant impact on the software industry and software professionals.

We have seen in all our experiments so far, that large language models (LLMs) have the potential to transform the software development industry by providing ways to automate tasks such as generating code, translating code, requirements elicitation, and more. As LLMs become increasingly adopted across organizations for software development, it is essential to consider the new set of skills that may emerge.

We will now evaluate the two hypotheses related to the impact of LLMs on the software industry and the necessary reskilling of software professionals.

Hypothesis 6: If code automation using large language models is adopted in the industry, then reskilling software professionals is necessary.

The first hypothesis suggests that the adoption of LLMs for code automation in the industry will require software professionals to be reskilled to adapt to the changing landscape.

We observed in our experiments that using LLMs for tasks like requirement classification, code generation, and code translation indeed saves considerable effort. While this can lead to increased efficiency and speed in software development, it also means that software professionals will need to be trained in new areas. Based on the experiments we identify the following skills that may become increasingly important when working with LLMs in software development. These skills include:

Prompt engineering

Prompt engineering involves creating high-quality prompts that can generate accurate and relevant results using LLMs. Prompts are the input texts or instructions that are provided to the LLMs to generate the desired output. Developing effective prompts requires an understanding of the way large language models process and understand text, a clear understanding of the task that needs to be done, domain knowledge, and natural language. By creating effective prompts, software professionals can ensure that the code generated or translated by LLMs meets the required quality and is relevant to the problem at hand.

We consider this as one of the crucial skills required to leverage code automation using large language models. We observed in the experiments conducted, that changing the way the instruction is given to the model, greatly helped improve the output achieved. For example, for a task of generating documentation for a piece of code, rather than giving simple instructions like "Generate documentation for the following piece of code", building a more comprehensive prompt with the required context helped get much better results.

Code analysis

As LLMs generate code automatically, software professionals should possess good analytical skills to analyze the generated code to identify any errors or inconsistencies. This skill will be essential to evaluate the quality of the code generated by LLMs.

In our experiments, we observed that LLMs tend to hallucinate. This means that a perfect-appearing code may contain incorrect functionality or bugs. This is especially true when the model does not have the required knowledge to perform the task or the knowledge of the programming language involved. In such cases, we have to provide some examples to the model to help it understand the task at hand. We observed that there were cases, where, given a few examples, the model started to produce code but the code was mostly a copy of the examples provided in the task. The code though appeared correct was not performing the intended task. To identify such cases, the analytical skills of programmers will play a critical role to analyze the code generated by the LLM. Based on the analysis done, further prompt engineering may be needed to nudge the model to produce code that is more aligned with the requirements.

Debugging skills

Debugging skills will be essential for software professionals to identify and fix any errors or issues that may arise when working with LLMs. As LLMs generate code automatically, it can be challenging to identify the root cause of errors or inconsistencies. Therefore, software professionals must have strong debugging skills to identify the problem and apply the necessary fixes. These skills will become even more critical as LLMs become more complex and generate more complex code.

Collaboration and communication skills

Collaboration and communication skills will be vital as software development teams become more cross-functional and include data scientists and other team members. With the increasing complexity of software development processes, it will be essential for team members to work together effectively to achieve the desired outcomes. Collaboration and communication skills are essential to ensuring that team members work together seamlessly, share their expertise, and achieve common goals.

Creativity and innovation

While LLMs can automate many software development tasks, creativity and innovation will still be essential for software professionals to solve complex problems that LLMs cannot solve and develop new products and features. Additionally, software professionals must find innovative solutions to incorporate LLMs effectively into the software development workflow.

Model fine-tuning and customization

To optimize the performance of LLMs in specific domains, software professionals will need to learn how to fine-tune and customize these models. This includes understanding the underlying architecture, training data, and methodologies, as well as leveraging transfer learning and other techniques to adapt the models to specific tasks or industries.

In some of the code-related tasks, we observed that using the pre-trained model and performing prompt engineering may not suffice. This is particularly applicable in the case of programming languages that the model does not have sufficient knowledge on. For such tasks, provided there is sufficient data available, fine-tuning the model is a better option. For example, for legacy languages, fine-tuning the model with the legacy language has a better chance of performing as the model would now be able to understand the legacy language. Also, since during finetuning, the knowledge the model already has is retained, the final fine-tuned model has a better chance of performing the task than the pre-trained model.

AI ethics and responsible AI development

As LLMs become more widely adopted, software professionals will need to address the ethical considerations when utilizing them. This includes understanding the potential biases in AI-generated code, addressing privacy concerns, and making sure that AI systems are transparent, fair, and accountable.

Data-driven decision-making

As LLMs can process and analyze vast amounts of data, software professionals will need to develop skills in data-driven decision-making. This includes understanding how to use data to inform software design, feature prioritization, and system optimization, as well as leveraging insights from AI-generated analysis to drive improvements.

Overall, we understand that the adoption of large language models for code automation will necessitate the reskilling of software professionals. New skills, such as prompt engineering, analytical and debugging skills, model fine-tuning, and responsible AI development, will become increasingly important.

Hypothesis 7: If code automation continues to increase in popularity and effectiveness, it will have a significant impact on the software industry and software professionals.

The second hypothesis suggests that the increasing popularity and effectiveness of code automation using LLMs will have a significant impact on the software industry and software professionals.

We have observed as part of our previous experiments that LLMs can automate a significant portion of software development tasks and can be employed to automate various software development processes. One potential impact of this trend is a shift in the job market for software professionals.

The impact of code automation using large language models like GPT-3, GPT3.5, Codex, and Bloom can be profound across various industries, including service-based companies and product-based companies.

Service-based companies

Workforce: Code automation can lead to a decrease in the demand for entry-level developers, as many routine tasks can be automated. However, there will be an increased need for skilled professionals who can manage, maintain, and integrate these AI-powered tools into existing systems.

Business: The revenue impact could be positive, as service-based companies can reduce costs by automating tasks and increasing efficiency, allowing them to take on more projects or clients. The reduced costs and increased efficiency from code automation could result in a better profit margin and allow for a more competitive pricing strategy. However, businesses may need to invest in training employees to work with new technologies and ensure a seamless transition to the AI-driven workflow.

Way of working: Service-based companies might need to adapt to a more hybrid way of working, where AI-driven automation tools complement human expertise. This can involve reevaluating business models and redefining roles within the company to make the best use of AI capabilities.

Thus, in the service industry, where software companies offer development services to clients, the adoption of LLMs for code automation can lead to faster delivery times and improved quality of work. With LLMs, software development tasks can be automated, allowing for faster delivery times, and software developers can focus on more complex tasks that require human creativity and expertise. This could potentially lead to an increase in demand for software development services, as software companies that can leverage LLMs effectively could deliver projects more efficiently than their competitors.

Product-based companies

Workforce: Code automation could decrease the demand for certain roles, such as entry-level developers, as routine tasks are automated. However, there will be an increased need for skilled professionals who can manage, maintain, and integrate AI-driven tools into existing workflows.

Business: Product-based companies can benefit from increased efficiency and reduced time-to-market for their products due to automation. This can lead to increased revenue opportunities, as companies can focus on innovation and expanding their product portfolios.

Way of working: Product-based companies might need to adopt a more collaborative approach between AI tools and human expertise. This involves investing in employee training, redefining roles, and responsibilities, and creating a culture that embraces innovation and change. Also, the role of product developers could evolve to focus more on designing and training models, rather than coding the product.

Thus, in the product industry, where software companies develop and sell software products, the adoption of LLMs for code automation could lead to increased competition. With the increased efficiency and automation offered by LLMs, new entrants into the market could develop software products and features faster and at a lower cost. This could lead to increased competition for existing software companies, particularly those that rely on manual processes to develop and maintain their products.

Overall, code automation using large language models has the potential to reshape various industries. While it can lead to workforce shifts and the need for new skills, the overall impact on business and the way of working can be positive, with increased efficiency, reduced costs, and a focus on innovation.

4.7 Summary of Findings

Our findings indicate that different software development tasks can be automated using large pre-trained models. In section 4.2., we discussed how these models can be leveraged across all the phases of the software development lifecycle from software requirements planning to design, development, and testing.

We also studied how traditional techniques compare against LLM-based approaches. Our findings indicate that for tasks like software requirements classification, LLMs provide comparable or even better accuracy than traditional techniques. The task of classifying requirements into functional vs non-functional gave an accuracy of 85% manual (expert). Traditional ML vs LLM-based classification also are comparable with an average f-score of 0.92. Similarly, NFR classification, when done manually by an expert gave an accuracy of 80%. Tradition ML techniques gave precision and recall between 72% to 90% and LLM-based NFR classification gave an average accuracy of 81%. Similarly, we observed that in code-related tasks, leveraging LLM was more efficient as compared to performing the same task manually. The average accuracy on different tasks like code generation, translation, and documentation, related to a modern programming language and simple to medium complexity applications was around 80%.

Our findings also indicate that significant cost and effort savings are obtained by leveraging artificial intelligence for different tasks as compared to the same tasks done traditionally. It was observed that for tasks like classifying software requirements, there was an effort saving of almost 57% and a cost saving of 80% as compared to the task done manually by an expert. Similarly, for multi-class classification, these figures were 33% and 68% respectively. We see similar savings of more than 60% on code-generation tasks, more than 50% on code translation tasks, and between 35% to 45% on end-to-end code migration tasks.

Our research findings also indicate that machine-learning models are probabilistic and cannot replace human jobs. Manual review and validation will always be required. These models can be used to augment the developer and serve as a peer programmer.

This would also allow the developers to concentrate on higher-order decision-making tasks like business understanding, stakeholder management, designing, and building for scalability and performance.

Our research findings also provide insights into the skillset which would be required when leveraging large pre-trained models for code automation. Analytical and debugging skills, prompt-tuning, and model-tuning will be the new set of skills that will be required, as indicated by the findings. The findings also indicate how code automation can impact both service-based and product-based industries. Workforce, ways of working, and business will all be redefined with the adoption of LLM in the industry.

CHAPTER V:

DISCUSSION

5.1 Discussion of Results

In the previous section, we did various experiments to answer the research questions raised at the beginning of this study. We also tested various hypotheses listed in the previous section and got some results. The results obtained and the experiments performed are all listed in section Chapter IV. We now discuss the results obtained to arrive at some conclusions.

5.2 Research Question One

"What are the software development tasks that can be automated using artificial intelligence-based language models?"

In section 4.2.1, we investigated the potential of large language models (LLMs) for automating various software development tasks. The primary goal was to assess the extent to which these models could improve efficiency, reduce human error, and contribute to the overall software development cycle. The discussion below highlights our main findings and delves into the implications of incorporating large language models into software development processes.

5.2.1 Planning and requirements gathering

In our study, we explored the potential of LLMs to automate and enhance various aspects of the planning and requirements-gathering phase of software development. Our findings indicate that these models can be effectively utilized in requirement classification, clustering, elicitation, traceability, similarity detection, and ambiguity detection.

The discussion below highlights the principal findings and their significance in the realm of planning and requirements gathering.

5.2.1.1 Requirement Classification and Clustering

Our results demonstrate that LLMs can help classify and cluster requirements based on their type and functionality. The models used for validating this task were from the GPT3.5 series. Figure 3 and Figure 4 show that by leveraging natural language understanding and pattern recognition capabilities, these models can automate the functional and non-functional requirement classification and clustering process, leading to time savings, reduced human errors, and better organization of requirements for subsequent analysis and development.

5.2.1.2 Requirement Elicitation

LLMs show proficiency in eliciting requirements from stakeholders. In section 4.2.1.5, we tested the capability of the GPT-3.5 series of models to perform this task of eliciting software requirements from customer reviews. Figure 7 shows the results obtained. We observe that these large language models can generate contextually relevant questions, analyze responses to refine requirements and ensure that the gathered requirements align with stakeholder expectations. This automation contributes to the overall quality of the requirements and streamlines the elicitation process.

5.2.1.3 Traceability

According to our findings, LLMs can be instrumental in maintaining traceability across the software development lifecycle. We observe in section 4.2.1.3, how for a requirement of multi-lingual support, the traceability matrix could help identify that both the code artifact and the test artifact were missing support for the French language, which was part of the original requirement.

In addition to using the inbuilt capability of the model in pattern recognition, when we have a large set of code, design, and test artifacts, pre-trained models like GPT-3 can be fine-tuned with this dataset. This helps overcome the context limitation of 4096 tokens that these models have. As a result of finetuning, given any new set of new artifacts, the model can generate the traceability matrix for the same without needing to be fed some examples for it to understand the task to be done.

Thus, we have seen that by automatically linking related requirements and their corresponding design, code, and test artifacts, these models can enhance project management capabilities and facilitate the handling of requirement changes.

5.2.1.4 Similarity Detection

In our study in section 4.2.1.4, we found that LLMs have the potential to identify similar or duplicate requirements within the gathered data. By analyzing the semantic meaning of the requirements, these models can detect overlaps, redundancies, and potential inconsistencies. This can contribute to improving the overall coherence and quality of the requirements set. This will ultimately result in a more efficient and accurate software development process.

5.2.1.5 Ambiguity Detection

In section 4.2.1.6, we observed that language models demonstrate the ability to detect ambiguous or unclear requirements. We saw various examples, where, by recognizing vague, ambiguous, or incomplete requirement descriptions, LLMs can flag them for further clarification and refinement. This finally ensures a higher quality and more consistent set of requirements.

Limitations and Future Work

Despite the promising results, it is important to recognize the drawbacks of LLMs when it comes to planning and requirements gathering. Their accuracy depends on the quality of training data, and they may struggle with understanding complex or domain-specific requirement descriptions. Moreover, biases lying in the underlying dataset can inadvertently be propagated by the models.

Future research should focus on enhancing the ability of LLMs to handle diverse and complex requirements, addressing biases in training data, and developing methodologies for seamlessly integrating these models into existing requirement engineering processes.

In conclusion, LLMs show great potential in automating and enhancing various aspects of the planning and requirements-gathering phase of software development. By improving requirement classification, clustering, elicitation, traceability, similarity detection, and ambiguity detection, these models can contribute to the overall success of software development projects. As we continue to refine these models and explore their integration into requirement engineering processes, further breakthrough and continued improvements are expected in this area.

5.2.2 Design

In section 4.2.1.2, we investigated the potential of LLMs to automate and enhance various aspects of the design phase of software development. Our findings indicate that these models can be effectively utilized in action extraction, design pattern recognition and classification, class extraction, and website design generation. The discussion below highlights the key findings and their substantial impact with regard to the design phase.

5.2.2.1 Action Extraction from Requirements Document

Our findings demonstrate that LLMs can effectively extract actions, actors, action details, as well as conditional actions from software requirements. LLMs extract actions by identifying and understanding the verbs and their corresponding objects. This enables the automatic generation of tasks or user stories, which can streamline the design process and facilitate better communication among team members. We also observed that by providing some samples to the model, the model learns better on the task of action extraction that it is required to perform.

5.2.2.2 Design Pattern Recognition and Classification

LLMs have shown promise in recognizing and classifying design patterns within existing code or design documents. In section 4.2.2.2, we have seen examples, where given a problem description, a large language model can correctly identify the best design pattern applicable to the problem description. Similarly, we have also seen how the LLM can identify the design pattern for a given piece of code. Thus, by understanding the structure and intent of design patterns, these models can assist developers in making informed decisions about the appropriate design patterns to implement, ultimately improving the overall software design quality.

5.2.2.3 Class Extraction for Given Problem Statement

Our research indicates that LLMs can be used to extract classes and their properties from problem statements or requirements documents. In section 4.2.2.3, we have seen how GPT-3 model can identify classes, their attributes, the methods, and the relationships between the classes, given a problem statement. By analyzing the text and identifying relevant entities and relationships, these models can ease the process of extracting the information required for creating class diagrams or object-oriented design structures, which serve as a basis for the implementation phase.

5.2.2.4 Website Design Generation

LLMs have demonstrated the ability to generate website designs based on high-level descriptions or design requirements. We have seen in section 4.2.2.4, two different website designs produced by two different models – text-davinici-003 and gpt3.5-turbo. Both designs meet the design requirements. By further prompt engineering, it would be possible to continue to develop and refine the design and convert the template code into a properly functioning website. Thus, we see that by interpreting the design intent expressed in natural language, these models can generate wireframes, layout templates, or even fully functional prototypes, effectively reducing the time and effort spent on the design process and allowing designers to focus on more complex tasks.

In conclusion, LLMs hold significant promise for automating various aspects of the design phase in software development. By leveraging their natural language understanding and pattern recognition capabilities, these models can improve the overall design quality and efficiency throughout the software engineering process. However, it is imperative to consider the limitations of LLMs, such as potential biases and inaccuracies, and to validate their outputs before integrating them into the design process.

5.2.3 Coding and Testing

In our study, we investigated the potential of LLMs to automate various aspects of the coding process in software development. Our findings indicate that these models can be effectively utilized in code completion and suggestion, code generation, code summarization and documentation, code translation, code review, and test case generation. The discussion below highlights the salient findings and their consequential significance in relation to code automation.

5.2.3.1 Code Completion and Suggestion

Our findings demonstrate that LLMs can effectively provide context-aware code suggestions and completions, reducing the amount of manual coding effort and improving developer productivity. We discussed in section 4.2.3.1, various AI-based tools and LLMs that are available today which provide automatic code suggestions. By understanding the semantic and syntactic patterns in the code, LLMs can predict and suggest the most appropriate code snippets, function names, or variable names based on the current context. Thus, these tools like TabNine, and CoPilot as well as large language models like Codex, ChatGPT, and GPT-4 can greatly help improve the productivity of developers.

5.2.3.2 Code Generation

We saw in section 4.2.3.2, how LLMs have shown promise in generating code based on high-level descriptions or requirements. By interpreting natural language input or pseudo-code, these models can generate syntactically and semantically accurate code snippets or full functions, effectively reducing the time and effort spent on coding and allowing developers to focus on more complex tasks.

5.2.3.3 Code Summarization and Documentation

Our research indicates that LLMs can be used to automatically generate code summaries, comments, and documentation. By analyzing code structures and semantics, these models can provide concise explanations of the code's functionality, aiding developers in understanding and maintaining the codebase.

5.2.3.4 Code Translation

In our study, LLMs have demonstrated the ability to translate code between different programming languages, allowing developers to convert existing codebases or reuse code snippets across multiple projects. By understanding the syntax and semantics of both source and target languages, LLMs can generate accurate translations while maintaining the original code's functionality.

5.2.3.5 Code Review

Our findings suggest that LLMs can assist in the code review process by detecting potential issues, such as syntax errors, logical flaws, or violations of coding standards. By analyzing the code and providing feedback, these models can help improve code quality and ensure adherence to best practices.

5.2.3.6 Test Case Generation

LLMs have shown potential in generating test cases based on code analysis or requirements. By understanding the code's functionality and identifying edge cases, these models can generate appropriate test inputs and expected outputs, allowing developers to automate testing and ensure that the code meets the desired specifications.

In conclusion, LLMs hold significant promise for automating various aspects of the coding process in software development. By leveraging their natural language understanding and code analysis capabilities, these models can improve developer productivity, code quality, and overall efficiency throughout the software development lifecycle. However, it is important to consider the limitations of LLMs, such as potential biases and inaccuracies, and to validate their outputs before integrating them into production systems.

5.3 Research Question Two

"How do large language models' code automation capabilities compare against traditional code generation?. We extend this question to different activities in the software development life cycle.

5.3.1 Planning and requirements gathering

Software requirement classification

In section 4.3.1, we discussed different approaches for performing the task of software requirements classification. Two tasks were evaluated:

a. Binary classification - categorizing software requirements as functional or non-functional
b. Multi-class classification of non-functional software requirements into multiple categories

Manual classification

As we saw in the earlier section, manual classification is a time-consuming process that requires human expertise and knowledge of the software domain. While it can be accurate and effective in certain cases, it can be prone to errors and inconsistencies due to human bias and subjectivity. To perform an objective analysis of both the above tasks when done manually, three candidates with different expertise were invited to perform the task. The person who was a novice in the field of performing classification manually and with limited domain knowledge could classify the requirements for the task of binary classification with an accuracy of 70%. Similarly, candidates with intermediate and expert knowledge in this task could accurately classify the requirements 81% and 85% of the time respectively.

These same candidates could classify the non-functional requirements into multiple categories with an accuracy of 62%, 71%, and 80% respectively.

Traditional ML

Similarly, we saw that traditional machine learning techniques have been explored in the past to automate these tasks of binary and multi-class classification of software requirements. The study by Dias Canedo and Cordeiro Mendes, 2020 showed that using a TF-IDF approach followed by logistic regression gave them an F-measure score of nearly 0.9 for binary classification and around 0.7 for NF classification. Similarly, the study by Almanza, which used CNN based deep learning approach for multi-class classification gave an F-measure of 0.77. Another study by Kurtanovic which did a manual selection of features followed by different pre-processing techniques helped them achieve a precision and recall between 72% to 90% for classifying NFRs. However, we also note the need for three critical components when building these classification models:

1. A significant amount of training data

2. Labeled data with clean and correct labels
3. Computational resources for training the ML models

All the above factors make traditional machine learning difficult to adopt, especially, in the case of data dearth. Also, traditional techniques like word2vec or TF-IDF can face limitations when dealing with complex language structures and contexts.

Large Language Models

As we have seen in the earlier section, large language models offer a solution for the limitations of the traditional machine learning approach discussed above. Since these models are pre-trained, they have a lot of in-built knowledge which can be leveraged for tasks like the classification of software requirements. We have seen that by giving them accurate instructions, these models like GPT-3 can perform the desired task with good accuracy. This reduces the need of training the model and thus reduces the need for significant data and compute. Also, the architecture of these models makes use of an attention mechanism, that enables the model to stay focused on the most important parts of the input data when making predictions. This helps the model to capture the relevant information and ignore the noise, making it more effective at text classification tasks.

We discussed two different approaches for performing classification using large-language models. The first requires no training and involves providing instructions and desired context to the model to perform the classification. This technique is called zero-shot learning. This technique gave an accuracy of 81% for binary classification. For multi-class, the accuracy score was 71%.

In the second technique, the embeddings from the LLM are utilized to build a classifier on top following the traditional approach. This approach though involving training gives better results since the model learns better from the small data set and is thus able to perform the classification better. The results show that the embeddings-based approach resulted in an accuracy of 92% and 81% for the tasks of binary and multi-class classification respectively.

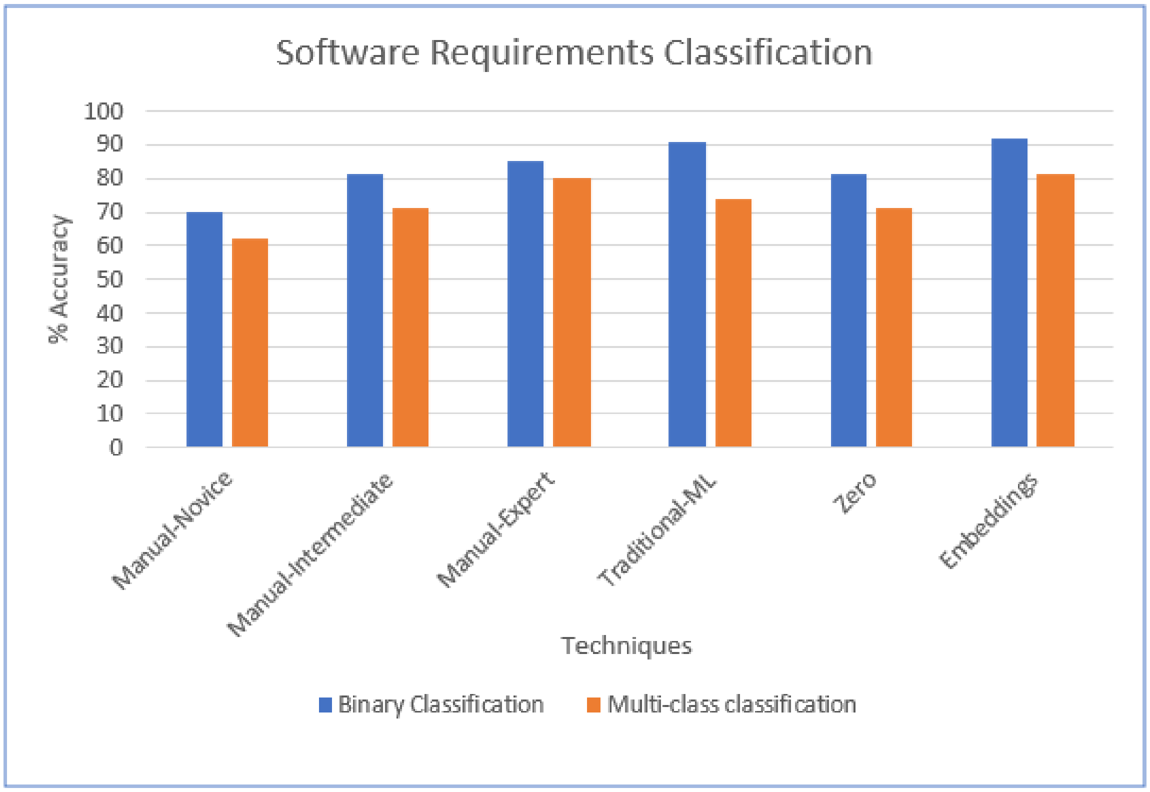


Figure 44
Software Requirements Classification - Comparison

Figure 44 above depicts the accuracies obtained using various techniques for both binary and multi-class classification. As is evident from the graph, both traditional machine learning and embedding-based approach give equivalent results for the binary classification of software requirements. For multi-class classification of NFRs, the results from the embeddings-based approach are superior to traditional ML and close to manual labeling done by an expert. The zero-shot approach from the LLM, though inferior to the embeddings-based LLM approach and traditional ML approach, produces results that are comparable to manual labeling done by a person with intermediate domain and classification skills. Also, zero-shot learning approaches can be improved iteratively by providing better context and instructions to the model. They can also be improved by techniques like few-shot learning, where the model can learn better by giving it some examples.

These results indicate:

1. Large language models can provide results comparable to or better than traditional approaches.
2. Large language models do not need as much data as would be needed for traditional machine learning.
3. Approaches like zero-shot learning can be applied even when there is no labeled data. Also, since this approach relies on instructions given to the model, the results will be consistent as compared to other approaches.
4. Large language models, being probabilistic, their zero-shot approach and embedding-based approach, would still require manual review.

5.3.2. Coding and Testing

5.3.2.1 Code Generation

As discussed in section 4.3.2, two problem statements were selected for the task of code generation. The first was an Android mobile application for drawing using touch. The second application was an ASP.NET C# application for building a travel reservation application.

Traditional techniques

We have seen that traditionally code is written manually. There are many editors available that aid in writing the code and provide suggestions or pointers to the correct libraries. For developing Android applications, Android Studio is one of the widely used tools, which helps in quickly building and designing Android applications. Similarly, Visual Studio is the application used for building ASP.NET applications.

Both the applications mentioned above were built manually by three individuals (novice, intermediate, and expert programmers) according to the setup discussed earlier. Android Studio and Visual Studio 2019 were used by programmers for building their applications. All were able to build the application but took a different amount of time to build the application. Also, the code written by all three individuals differed in terms of style, structure, and naming conventions.

Large Language Model

The large language model considered here was Codex. ChatGPT model was also used to assist in the UI building for the ASP.NET application. We observed that in both cases the code generated using the AI model was almost 90% correct. The remaining code needed corrections to convert the code into a fully functional application. The same editors i.e. Android Studio and Visual Studio 2019 were used to edit and make corrections to the code. The code produced by the AI model was more structured, followed the right naming conventions, and was less error-prone as compared to the code written manually.

5.3.2.2 Code Translation

The second task we evaluated was code translation. Here, the source application was a C++ game of Tic-Tac-Toe. We attempted to translate it to Java. The experiment setup was the same as in the previous case.

Traditional Techniques

To perform the task of translating the code from C++ to Java manually, the programmers involved, based on their category, were required to have basic to expert knowledge of both the source and target programming language. The editor used by the programmers for building the Java application was BlueJ. All three individuals were able to translate the source application to the target language. However, as in the case of code generation, they all had different turn-around times. The other observations also were the same as in the task of code generation i.e., the quality of code, code consistency, structure, and programming style varied from individual to individual.

Large Language Model

As in the case of code generation, the OpenAI Codex model was leveraged for performing the task of code translation. The source C++ application consisted of multiple application and header files that were linked together. For meaningful translation, it was important to provide the required context i.e., snippets of relevant code from dependent files, to the piece of code being translated. This improved the accuracy of the translation. By applying various techniques and performing the prompt-engineering, the model was able to generate code with almost 90% accuracy and required a few modifications/corrections to turn the code into a working application. In the previous section, we have seen screenshots of the entire translated Java application, which was aided by the AI model. The factors, related to code quality, structure, and style, when considered for the code generated by an AI model were better than those of a novice programmer and comparable to an intermediate programmer or expert programmer.

5.3.2.3 Legacy Code Documentation and Migration

Here, the task was to document legacy COBOL code and migrate it to Java. For documentation, two types of documentation were required:

- Generic documentation which describes the intent of the program and gives a high-level overview of the program
- Design document

Finally, it was required to migrate the application to Java.

Traditional Techniques

When performing the task of code translation manually, there were multiple challenges:

1. The individuals involved in this task should have an understanding of both a procedural language like COBOL and an object-oriented language like Java
2. For the task of writing documentation for the code and creating the design document, a good understanding of COBOL was required.
3. The task of writing documentation and creating design documents manually is effort intensive, irrespective of the expertise in the programming language involved.

The individuals who performed the task of this legacy code migration were those with a basic understanding of COBOL and either basic, intermediate, or expert programming skills in Java. They all accomplished the task but required a considerable amount of time to complete it. The factors like code quality and consistency were the same as in the other tasks. These got extended to the documentation process as well i.e., the documentation produced by everyone was structured differently and was written in different styles.

Large Language Model

The model used here for performing the task of documenting the COBOL code and generating the Java code was ChatGPT (gpt3.5-turbo). The challenge here for leveraging the AI model was different than the translation task. In this task, the COBOL code was extremely large. All this code could not be fed to the model, at the same time, to either generate the documentation or generate the Java code. Different engineering techniques were applied to chunk the code into coherent blocks to get the desired functionality. In the previous sections, we have seen screenshots of the fully functional and migrated Java application. The generated documentation was well-structured and coherent. Similarly, the generated Java code was well structured and comparable in quality to that written by an expert.

By considering various factors like the ability of the AI models to generate, document, or translate code, the accuracy of the generated code/documentation, quality of code, and so on., and the various experiments conducted in the software requirement classification and coding and testing phase, we validate our hypothesis that "The code generation/translation capabilities of large language models are better than traditional code generation and rule-based tools."

It is important to note, however, that the code generation tasks considered here, were comparatively simple, as compared to a complex application with multiple features, security considerations, and performance requirements that may require specialized expertise and manual coding to ensure optimal results. In such cases, relying solely on a language model for code generation may not be sufficient and may need to be supplemented with manual coding and rigorous testing to ensure the functionality, security, and performance of the application. In such cases, it may still be useful to utilize a language model for generating certain portions of the code or for providing suggestions and guidance to the developers.

5.4 Research Question Three

How much effort saving and cost-benefit will organizations get by augmenting these AI-based models with a software programmer?

5.4.1 Planning and requirements gathering

Software requirement classification

As discussed in the previous sections, we experimented with employing a large language model for the task of software requirements classification. The model was used to perform:

- Binary classification i.e., functional vs non-functional requirements classification
- Multi-class classification i.e., NFR classification into multiple categories

As discussed in section 4.4.1, for each classification process, two approaches were evaluated to perform the desired task. The first was zero-shot learning, which involved prompt engineering. We discussed the various steps involved in building a classification solution, for both binary classification as well as multi-class categorization using the zero-shot approach. The second approach was building an embeddings-based model. Again, we discussed various steps involved in building this model for both binary and multi-class prediction.

To calculate the cost and effort savings, the same task of binary and multi-class classification was also performed manually by people with different levels of expertise. Following graphs depict the cost and effort involved in performing binary and multi-class classification of requirements using AI-based and manual techniques.

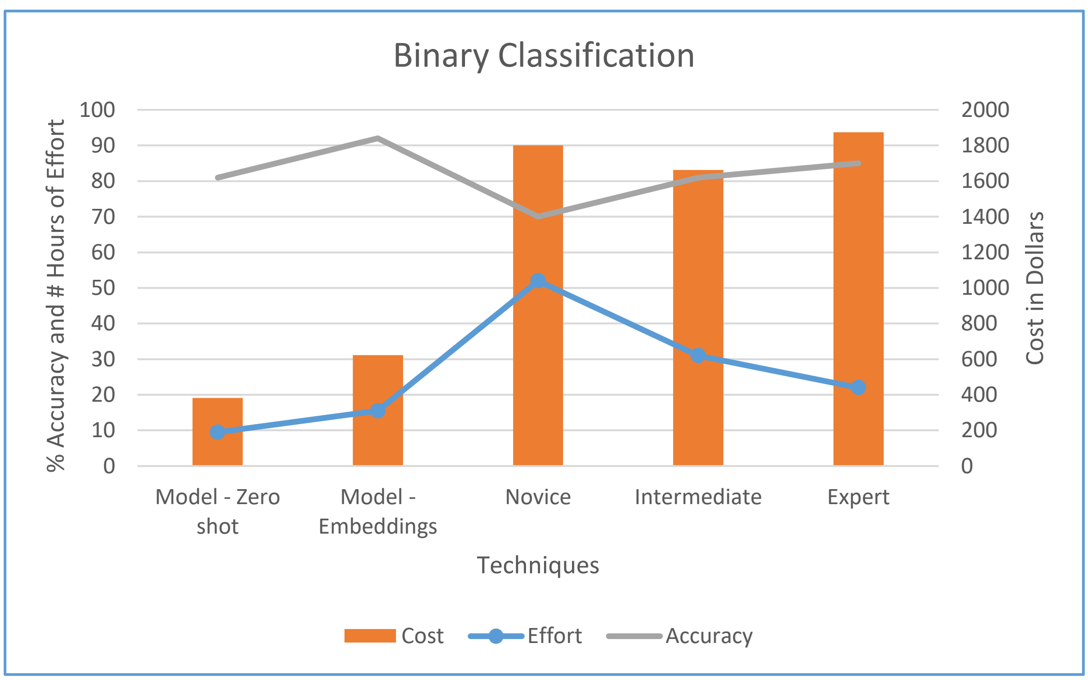


Figure 45
Binary Classification - Cost and Effort

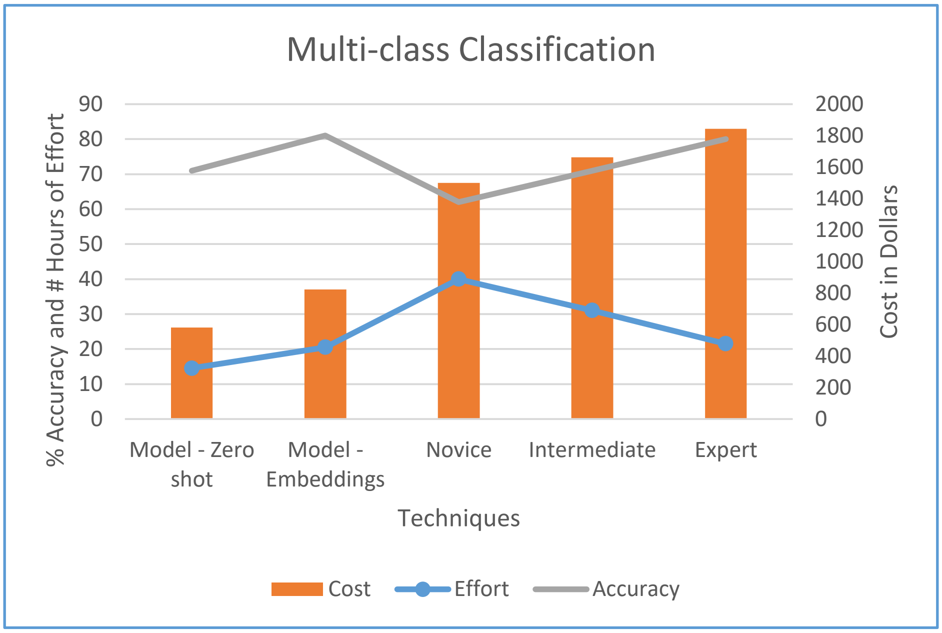


Figure 46
Multi-class Classification - Cost and Effort

Similarly, the graphs below depict the percentage cost and percentage effort saving obtained when using zero-shot learning and embeddings-based approach for binary classification as compared to the same being performed manually by people of different levels of expertise.

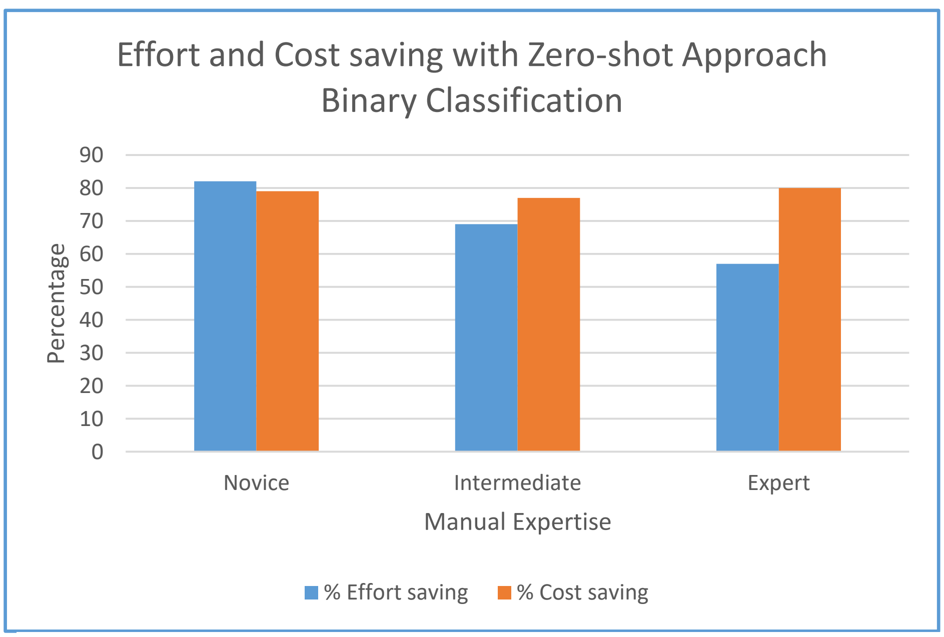


Figure 47
Effort and Cost saving - Zero-shot (Binary)

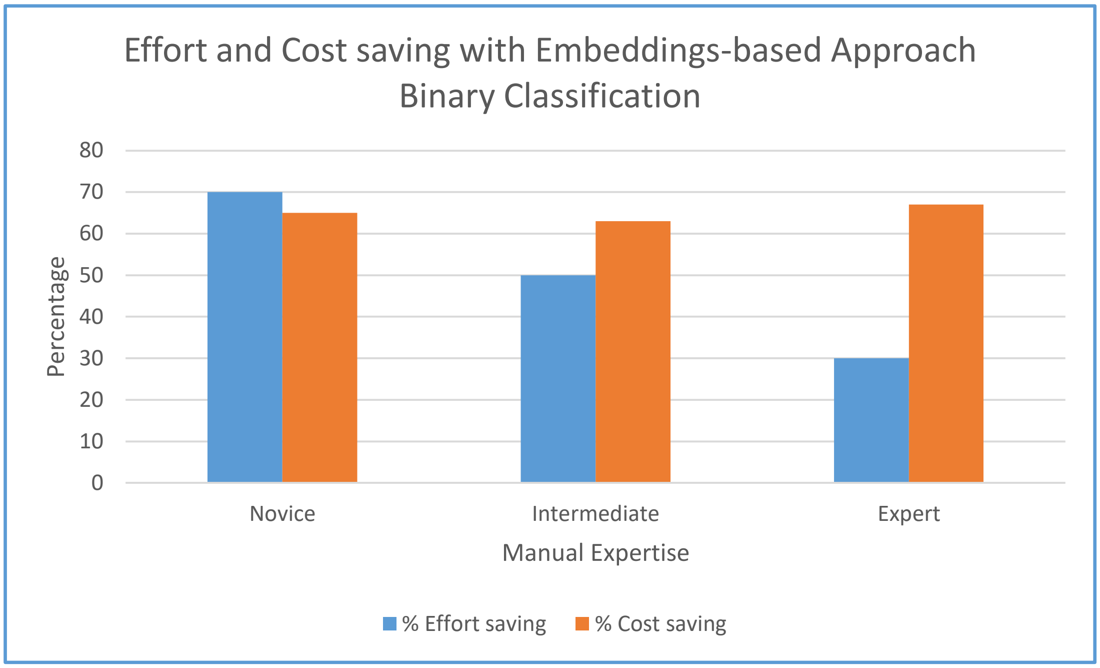


Figure 48
Effort and Cost Saving – Embeddings based (Binary)

For the current task, which involves simple sets of software requirements, we observe the following:

Binary Classification

- AI techniques vs Manual (Novice)

  If the binary classification is done using the model instead of being done manually by a person who is a novice, the model will provide significant savings in both cost and effort. The effort will be reduced by almost 82% using the zero-shot approach and 70% using the embeddings-based approach. Similarly, the cost will be reduced by 79% and 65% in both these cases. We have also seen earlier that in this case, the accuracy obtained using AI techniques is much higher as compared to the manual approach.

- AI techniques vs Manual (Intermediate)

Similarly, when comparing using an AI model versus manual performance by an individual with intermediate skills, the cost savings is greater than the effort savings for performing binary classification. This indicates that implementing the AI model in this scenario would result in a reduction of effort by 69% (zero-shot) and 50% (embeddings-based) and cost by 77% (zero-shot) and 63% (embeddings-based). Additionally, previous observations have shown that the accuracy of the AI model is 81% (zero-shot) and 92 % (embeddings-based) which is either the same or higher than that performed manually (81%) in this case.

- AI techniques vs Manual (Expert)

  When comparing the performance of an AI model to that of an individual with expert skills, we find that the AI model offers much higher benefits in cost than in effort. Notably, the AI model provides significant cost savings compared to the manual approach, with a reduction of up to 80%. Additionally, while the effort savings may be less pronounced at 57%, the AI model can perform the task much more quickly and consistently than a human expert, improving efficiency and reducing the likelihood of errors. Regarding accuracy, the accuracy of the AI model (92%) exceeds that of a human expert (85%) in the case of the embeddings-based approach, while is only slightly lower (81%) than the human expert in the case of zero-shot approach, ensuring high-quality results.

Similarly, the graphs below depict the percentage cost and percentage effort saving obtained when using zero-shot learning and embeddings-based approach for multi-class classification as compared to the same being performed manually by people of different levels of expertise.

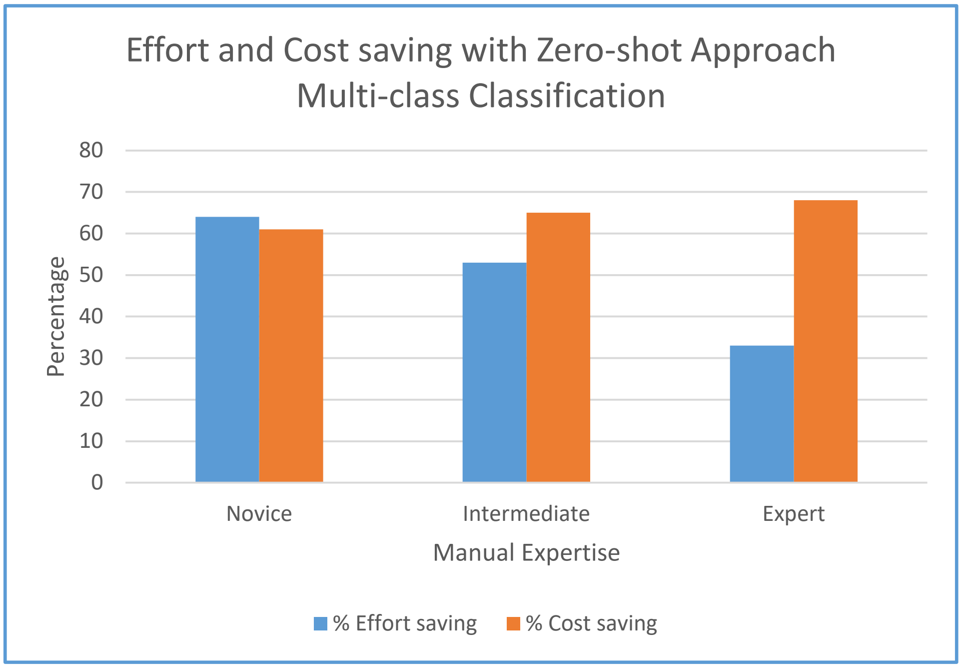


Figure 49
Effort and Cost saving - Zero shot (Multi-class)

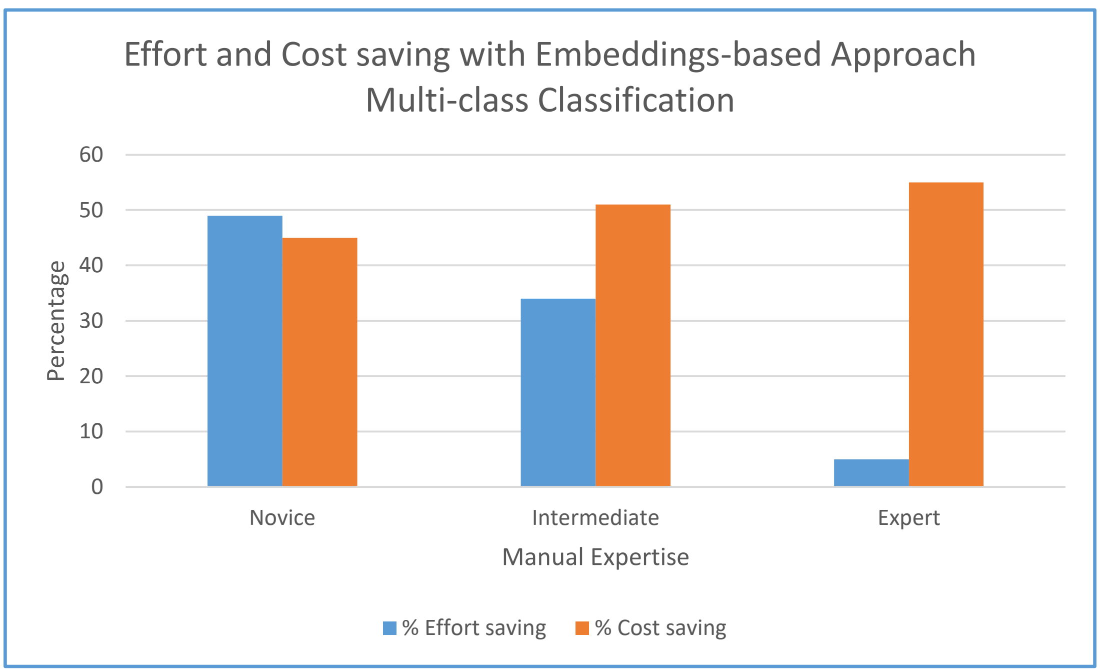


Figure 50
Effort and Cost saving - Embeddings based (Multi-class)

Multi-class Classification

- AI techniques vs Manual (Novice)

Just like in the case of binary classification, AI techniques, both zero-shot and embeddings-based approaches, provide significant cost and effort saving for the task of multi-class classification of non-functional requirements, as compared to the task being done manually by a person who is a novice. The effort will be reduced by almost 64% using the zero-shot approach and 49% using the embeddings-based approach. Similarly, the cost will be reduced by 61% and 45% in both these cases. The accuracy percentage obtained manually, in this case, is 62% which is considerably lower as compared to 71% (zero-shot) and 81% (multi-class).

We can see that there is a significant drop in effort savings, and hence cost savings, when using an embedding-based approach. This is because building a multi-class model and applying different techniques to optimize the model's performance consumes considerable time and effort, thus reducing the overall effort savings.

- AI techniques vs Manual (Intermediate)

Similarly, as in the case of binary classification, when comparing the AI model versus the manual classifier with intermediate skills, the cost savings is greater than the effort savings for performing multi-class classification. We see a reduction of effort by 53% (zero-shot) and 34% (embeddings-based) and cost by 65% (zero-shot) and 51% (embeddings-based). Additionally, we have seen that the accuracy of the AI model is 71% (zero-shot) and 81 % (embeddings-based) which is either the same or higher than that performed manually in this case.

We can see that as the expertise of the person performing the task increases, the effort savings obtained from the model decrease.

- AI techniques vs Manual (Expert)

In this case, we have seen that the drop in effort saving is much higher than in the last two cases. Here, the effort saved using the zero-shot approach is 33% while using the embeddings-based approach is only 5%. However, the cost savings are better at 68% (zero-shot) and 55% (embeddings-based). The accuracy however, using the zero-shot approach (71%) is lower compared to that obtained by the manual expert (80%). Conversely, the embeddings-based model performs at par with the manual expert, in terms of accuracy. Overall, we can see that for multi-class classification, the advantages obtained in using an AI model are more towards saving cost than effort. Especially, in the embeddings-based approach, the effort saving is negligible but both cost saving and accuracy are better than what can be achieved by a manual expert.

It is critical to highlight, that the current study has been done with a set of requirements that are simple in complexity. These requirements are short and of few sentences only. Also, when categorizing the non-functional requirements into specific categories, the categories are of known types like “Look and Feel”, “Security” and so on.

The time taken to build a solution using OpenAI models, in this case, was comparatively less due to two reasons:

1. The model is a large language model which is already pre-trained and has a very good understanding of both functional and non-functional requirements. Hence, making it learn the task of performing the classification did not take much effort.
2. The requirements were simple and short. Hence, it did not involve significant effort in cleaning up the requirements.

Conversely, in the case of complex and sizable requirements, where the task entails classifying them into some custom categories, the effort involved in building such a solution will increase considerably. Following would be the considerations that would come into play when building such a solution:

1. Every large model has a context window limitation i.e., there is a limit on the total number of words the model can process including the output to be generated, at any given time. If the requirements are large and exceed this limit, then it would involve building a few more steps into the solution. This would include breaking down the requirement into smaller chunks and then generating the summary for the requirement. The summary would then be used for classification purposes.
2. If the categories are custom, then, it would involve significant prompt-engineering techniques. Zero-shot may not be efficient in such a case. Few-shot or chain-of-thought approach would have to be leveraged. It could also involve creating complex instructions with a step-by-step process that the model must follow to perform the classification.
3. Another approach that may have to be tried is to build a pipeline where the output from the previous step feeds into the next step, where each step is used to extract different information from the requirements. In the final step, all the extracted information can be leveraged to perform the classification.
4. For any approach that is built, a manual review of the classification being done by the model with inputs to improve the performance will be required.

Building such solutions will involve much more effort and hence also increase the cost. However, these solutions will still be useful as compared to the manual approach, due to the following two reasons:

1. Building the solution using prompt engineering is possible even when the available dataset is extremely small and not labeled.
2. Given that the large language model understands the semantic meaning of words, it can extract meaningful information from large requirements and is much less prone to errors as compared to the manual approach.

3. Once the solution is built, the time used to process new requirements and perform the classification will be much less as compared to the manual approach. Also, the accuracy of these models is comparable to at least a person with intermediate expertise in the domain.

Considering all the above points, we validate all three hypotheses that we had proposed to be true:

- *If large language models are to be used optimally, then a human-in-the-loop is necessary for validation of the result generated by the model.*
- *If code automation using large language models is leveraged in the industry, it can improve software quality by reducing errors and increasing consistency.*
- *If large language models are used for code automation, it can improve software development efficiency by reducing the time needed to develop code and provide significant cost and effort benefits to software organizations.*

5.4.2 Coding and Testing

5.4.2.1 Code Generation

As we discussed in section 4.4.2.1, the first app that was developed using both manual and AI-assisted techniques was an Android app for drawing on the screen using touch and the other was an ASP.NET application. The graphs below depict the cost and effort involved in building these applications. Another set of graphs also depicts the percentage effort saving and percentage cost saving we get by leveraging these AI models as compared to these applications being built manually from scratch by developers of different levels of expertise.

We observe that leveraging an AI model gives significant effort savings, especially when compared to a novice programmer. As the expertise of the programmer increases, the effort savings obtained reduce. However, the cost savings are still significant. We have seen that even when compared to an expert programmer, the model can provide a cost saving of at least 68% when considering the task of code generation.

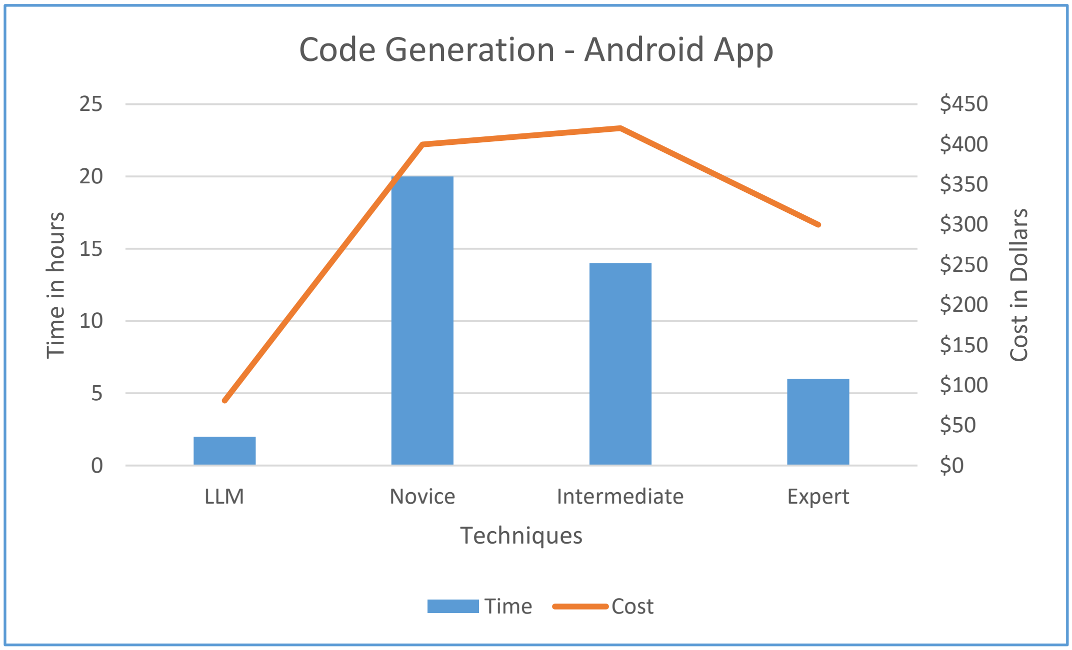


Figure 51
Cost and Effort - Code Generation (Android App)

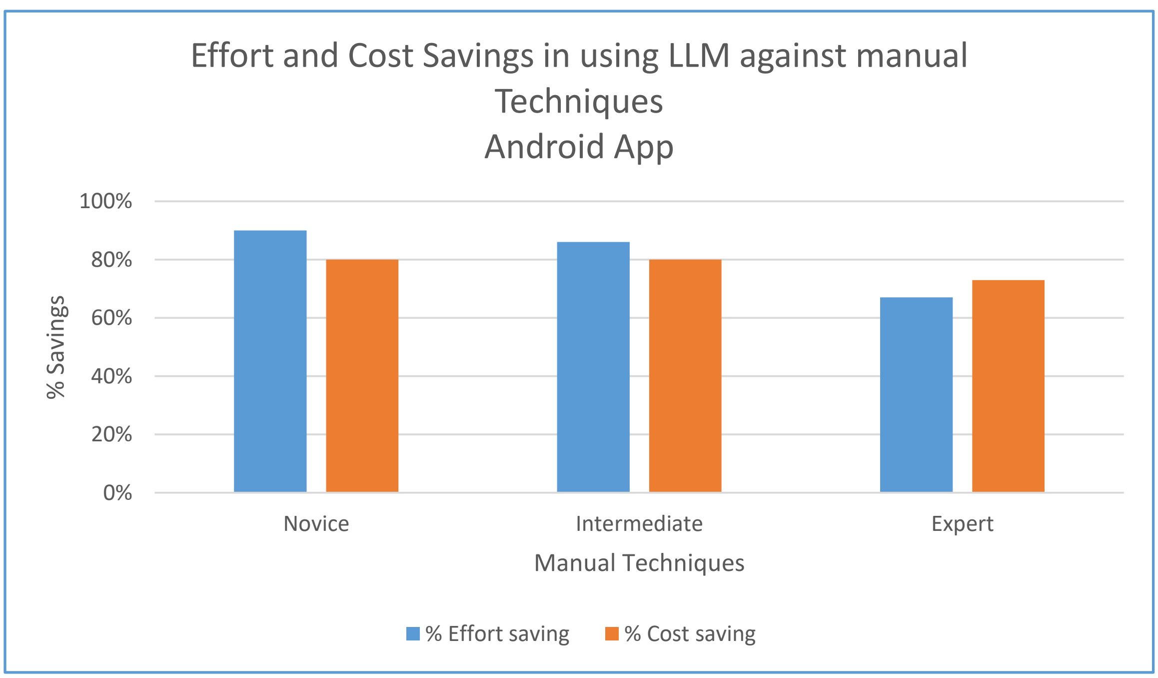


Figure 52
Effort and Cost Saving- Android App

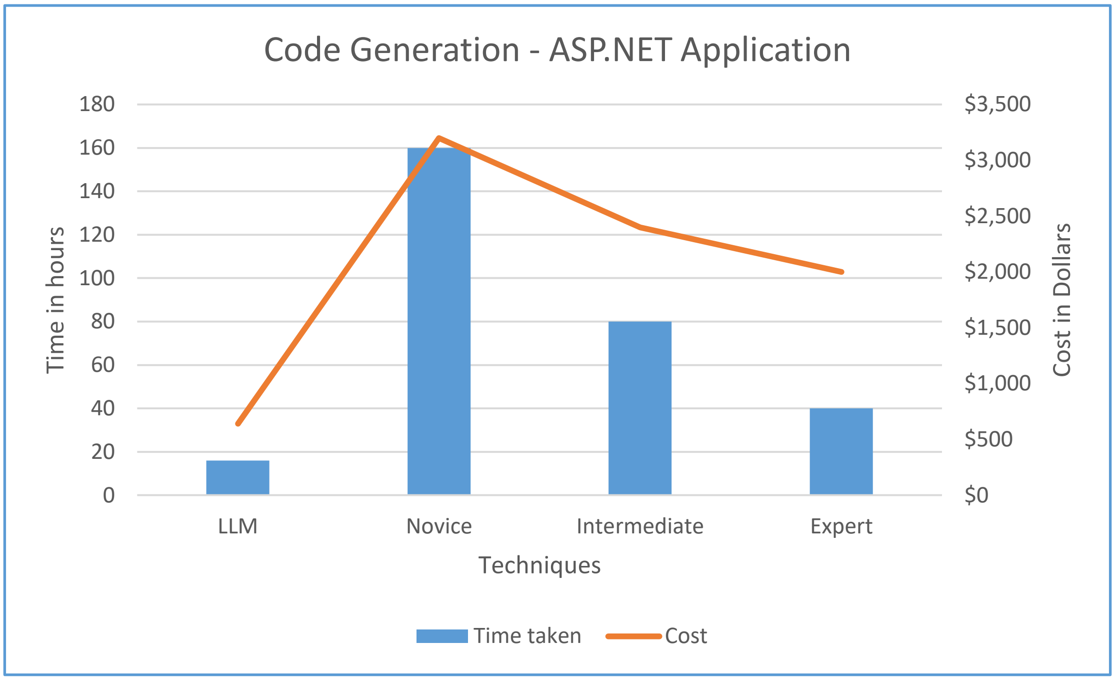


Figure 53
Cost and Effort - Code Generation (ASP.NET)

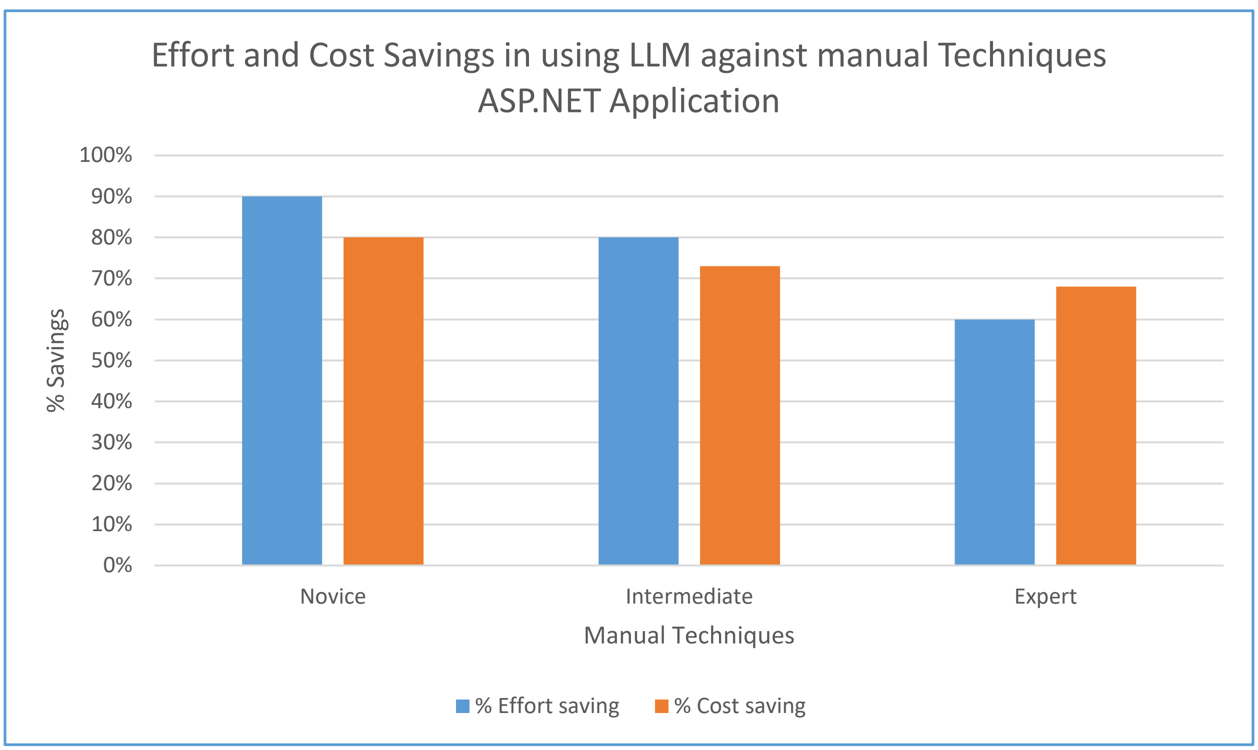


Figure 54
Effort and Cost Savings (ASP.NET)

We have also seen that the accuracy of the suggested code in both cases was almost 90%. This means around 10% of the code had to be corrected to build a complete working application.

Cost-Benefit Analysis (with respect to Expert programmer) – Android App

Table 87
Cost-Benefit Analysis - Android App

	Manual Code Writing	**Code Generation using AI model**	**Saving**
LOC	150	150	
Approximate % code which is generated correctly		80%	80%
Developer's effort (hours)	6	2.5	
% Effort saving using AI		60%	60%
Developer's rate ($ per hour)	50	40	
Developer's cost to client ($)	300	100	
Manual effort involved (correction/validation) in hours		1.5	
Effective % Effort Saving (Overall)		33%	33%
Cost of manual effort (correction/validation) needed with AI model		60	
Cost of API usage		1	
Total cost to client	300	161	46%

Cost-Benefit Analysis (with respect to Expert programmer) – ASP.NET Application

Table 88
Cost-Benefit Analysis - ASP.NET

	Manual Code Writing	**Code Generation using AI model**	**Saving**
LOC	1326	1326	
Approximate % code which is generated correctly		90%	90%
Developer's effort (hours)	40	10	
% Effort saving using AI		75%	75%
Developer's rate ($ per hour)	50	40	
Developer's cost to client ($)	2000	400	
Manual effort involved (correction/validation) in hours		6	
Effective % Effort Saving (Overall)		60%	60%
Cost of manual effort (correction/validation) needed with AI model		240	
Cost of API usage		2	
Total cost to client	2000	642	68%

We must note, that the applications built were of simple to medium complexity and were small applications. There are a few points to consider when leveraging AI models for the task of code generation and building large applications:

1. Language models are probabilistic and it is important to have a human in the loop when we build applications using these models. Manual intervention is required to review and correct the code.
2. Large language models have context window limitations, meaning they can only consider a limited amount of context when generating code. This can cause them to

miss important information needed to generate effective and efficient code for large applications.

3. Pre-trained language models lack the ability to understand the complex relationships between different pieces of code within an application, which can lead to generating code that is not well-optimized and may not work effectively within the larger application.
4. In addition to point 3, a large application may have interdependencies between many different components or frameworks, which further complicates the challenge of generating effective code. These large models may not be able to capture these complex relationships, leading to generating code that may again not be well-optimized or effective within the larger application.

To overcome these limitations, it is important that large pre-trained transformer models should be used in conjunction with other techniques, such as:

- Analyzing existing code to understand dependencies and interdependencies between different pieces of code.
- Incorporating human input and oversight into the code generation process.

Due to the challenges involved, the effort involved in building a large application with many different components and interdependencies will be considerably higher.

For example, consider building an enterprise-level application that involves building a frontend with technologies like Angular or ReactJS, backend implementation, database integration, features like payment processing with gateway integration, authentication libraries, and so on. Let us assume this application consists of approximately 50,000 LOC and it takes 20 weeks with four developers to build the application. Let us also assume that in these four developers, there is one expert programmer and two programmers with intermediate expertise, and one novice developer. Given, these conditions, the following table shows the cost-benefit analysis for building this application manually versus when assisted with an AI model.

Table 89
Cost-Benefit Analysis - Code Generation (Complex application)

		Manual Code Writing	**Code Generation using AI model**	**Saving**
LOC		50000	50000	
Approximate % code which is generated correctly			45%	45%
Developer's effort (hours)		3200	800	
% Effort saving using AI			75%	75%
Developer's rate ($ per hour)			40	
	Expert	50		
	Intermediate	30		
	Novice	20		
Developer's cost to client ($)		104000	32000	
Manual effort involved (correction/validation) in hours			1700	
Effective % Effort Saving (Overall)			22%	22%
Cost of manual effort (correction/validation) needed with AI model			51000	
Cost of API usage			600	
Total cost to client ($)		104000	83600	20%

Thus, we can see that in a large application with around 50000 LOC and many interdependencies, if the model generates approximately 45% of the code correctly, the actual effort saving is only 22% in contrast to the 60% effort saving observed in the study. This is due to the fact, that there is considerable manual effort involved in correcting, validating, and stitching the remaining 55% of the code. Similarly, we can see that the cost saving is approximately 20% of the cost taken to manually build the application compared to the 68% effort saving observed during the study.

Also, the current model being used is an API-based model. Hence, there is only API usage cost involved. If the model used is an open-source model which also involves deploying the model on-premise and fine-tuning the model, this will add to the effort involved in fine-tuning the model and the cost of hardware (compute) required to deploy the model. This will depend on the size of the model used and the amount of compute needed for deployment, fine-tuning, or inferencing.

5.4.2.2 Code Translation

We discussed in section 4.4.2.2, how an AI-assisted approach can be followed for the task of code translation. The source was a C++ file for a Tic-Tac-Toe application and the target was a Java program for the same. The graph below depicts the cost and effort involved in performing the translation using AI models.

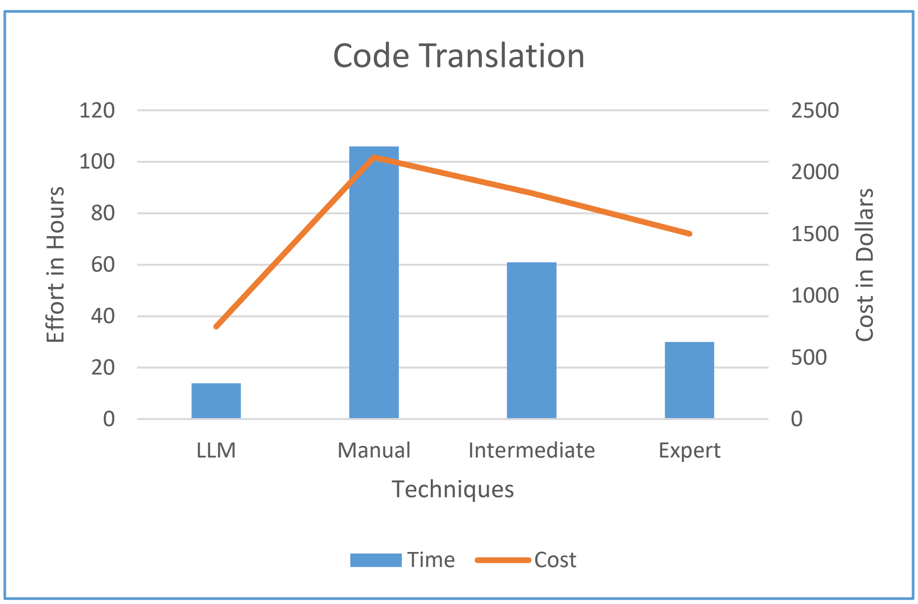


Figure 55
Cost and Effort - Code Translation

Similarly, the graph below depicts the percentage effort saving and percentage cost saving we get by leveraging these AI models as compared to these applications being translated manually from scratch by developers of different levels of expertise.

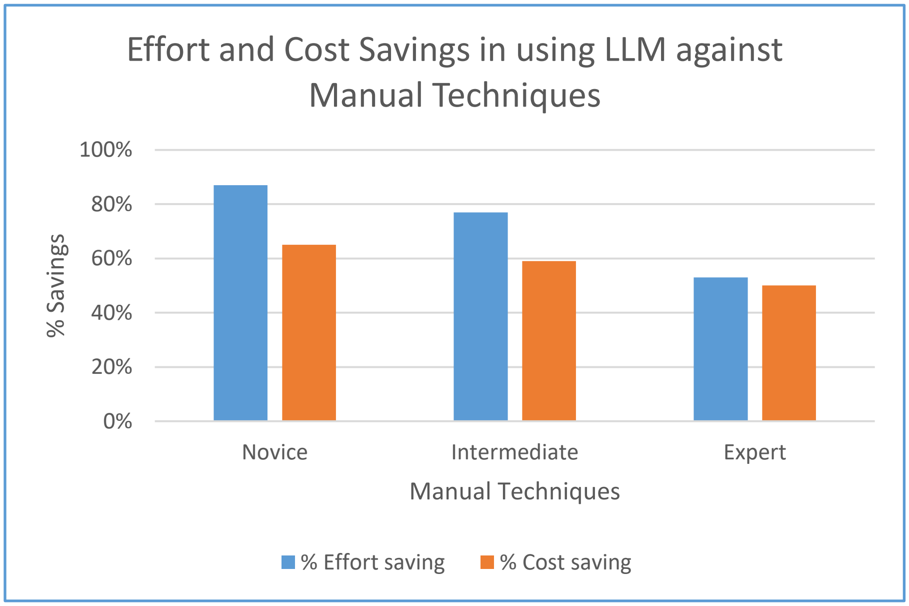


Figure 56
Effort and cost saving - Code Translation

Like code generation, we observe that leveraging an AI model gives significant effort savings, especially when compared to a novice programmer. Also, we observe that the effort and cost savings obtained when compared to an expert are less than those obtained against a novice programmer. However, the savings are still significant. We observe that both effort and cost savings obtained by leveraging AI against as compared to an expert programmer are approximately around 50%.

The accuracy of the translated code was almost 90%. This means around 10% of the code had to be corrected to build a complete working application.

Cost-Benefit Analysis (with respect to Expert programmer)

Table 90
Cost-Benefit Analysis - Code Translation

	Manual Code Writing	**Code Generation using AI model**	**Saving**
LOC	800	800	
Approximate % code which is generated correctly		90%	90%
Developer's effort (hours)	30	9	
% Effort saving using AI		53%	70%
Developer's rate ($ per hour)	50	40	
Developer's cost to client ($)	1500	360	
Manual effort involved (correction/validation) in hours		5	
Effective % Effort Saving (Overall)		53%	53%
Cost of manual effort (correction/validation) needed with AI model		200	
Cost of API usage		189	
Total cost to client	1500	749	50%

Like code generation, the application translated here was of simple to medium complexity and was a small application. All the points mentioned during the code generation section are applicable for code translation too when leveraging AI models for translating large applications. In addition, during code translation, it is important to be aware that the translation is generally an as-is translation. This means that any code optimization or re-architecting that may be needed as part of the translated application, will bring in additional effort. Hence, combining different techniques including rule-based and tool-based where applicable, is recommended.

For example, let us consider the case of a company that has a C-based system that is used to manage inventory and sales data, but the system has become outdated and difficult to maintain. The company wants to modernize the system by transitioning to a more modern programming language like Java.

The C-based system has around 100,000 lines of code, including functions, structures, and libraries. The translation process would involve analyzing the existing code, identifying any language-specific features or dependencies, and then rewriting or refactoring the code to work with Java.

This process could be challenging, as C is a low-level language that requires manual memory management, while Java is a high-level language that uses automatic garbage collection. The translation effort would involve mapping C constructs to Java equivalents, such as translating C functions to Java methods, C structures to Java classes, and C libraries to Java packages.

There may also be performance considerations related to moving the system to Java, as C is typically faster than Java. The performance impact of the translation would have to be studied and the code would have to be optimized as needed.

Given the complexities involved, let us say, it takes approximately 400 developer days to do this translation manually. Just as in code generation, the assumption is there is 1 expert programmer and 2 programmers with intermediate expertise and 1 is a novice developer

Given, these conditions, the following table shows the cost-benefit analysis for building this application manually versus when assisted with an AI model.

Table 91
Cost-Benefit Analysis - Code Translation (Complex Application)

		Manual Code Writing	**Code Generation using AI model**	**Saving**
LOC		100000	100000	
Approximate % code which is translated correctly			45%	45%
# Avg. LOC per person		250	1000	
# Developers		4	2	
# Avg. LOC per day		1000	2000	
# Days for translation		100	50	
Developer's effort (hours)		3200	800	
% Effort saving using AI			75%	75%
Developer's rate ($ per hour)			40	
	Expert	50		
	Intermediate	30		
	Novice	20		
Developer's cost to client ($)		136000	32000	
Manual effort involved (correction/validation) in hours			1760	
Effective Hours Overall			2560	
Effective % Effort Saving (Overall)			20%	20%
Cost of manual effort (correction/validation) needed with AI model			80000	
Cost of API usage			600	
Total cost to client ($)		136000	112600	26%

Thus, we can see that when trying to translate a large application with around 100000 LOC, if the model generates approximately 45% of the code correctly, the actual effort saving is only 20%. This is due to the fact, that there is considerable manual effort involved in correcting, validating, and stitching the remaining 55% of the code. Similarly, we can see that the cost saving is approximately 20%.

As in code generation, the current model being used is an API-based model. Hence, there is only API usage cost involved. Using an open-source model and fine-tuning it will change the accuracy figures and accordingly the effort and cost savings incurred.

5.4.2.3 Legacy Code Documentation and Migration

In section 4.4.2.4, we measured the cost and effort saved in generating code documentation (generic and design) from legacy COBOL code and migrating the code to Java. The graphs below depict the cost and effort involved in performing the individual steps of documentation followed by migration and in performing the end-to-end process. The percentage effort saving and percentage cost saving obtained by leveraging these AI models as compared to performing migration manually by developers of different levels of expertise is also shown below.

As in the previous tasks, we observe that leveraging an AI model gives significant effort savings, especially when compared to a novice programmer. We also observe that significant effort and cost savings are obtained by leveraging AI even against an expert programmer.

As observed in the figure below, while a novice programmer takes 140 hours to document the COBOL code into generic and design documentation, the AI model can perform the same task in only 34 hours. Due to this, we see a significant difference in the cost incurred in this process when using the AI model.

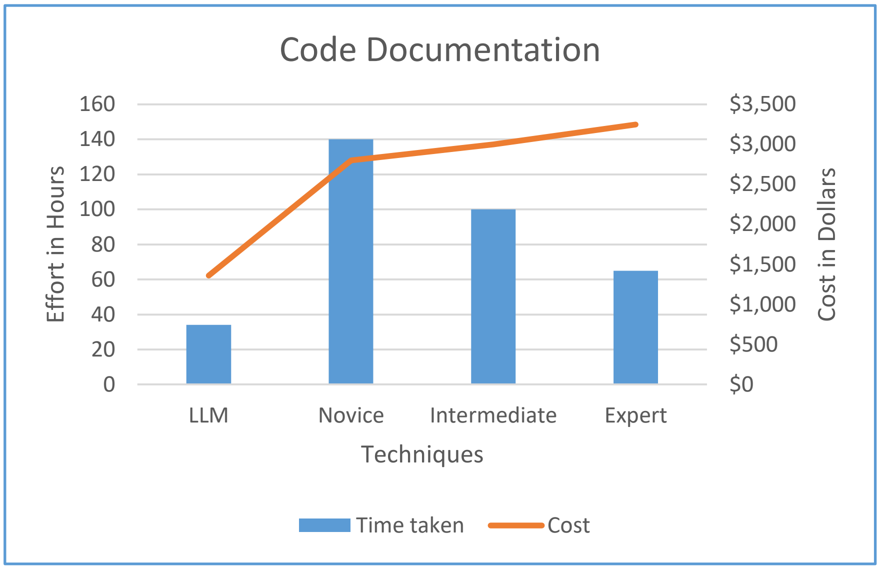


Figure 57
Cost and Effort - Code Documentation

Similarly, we see that a novice programmer takes approximately 180 hours to generate Java code by referring to the documentation and code. While the AI model can generate the code in 60 hours. This again causes a significant difference in the Java code generation process.

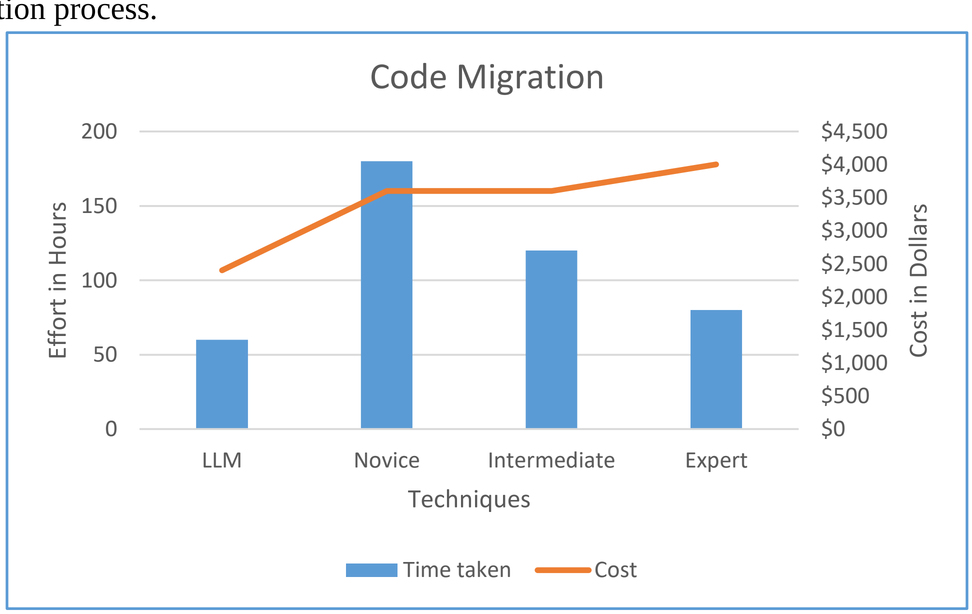


Figure 58
Cost and Effort - Code Migration

We also observe, that while the novice programmer takes 180 hours, an expert programmer takes significantly less time, approximately 80 hours to do the same task. Hence, the effort saving obtained in this case is less as compared to the previous case.

Figure 59 below shows the effort and cost involved in the end-to-end process of the code migration.

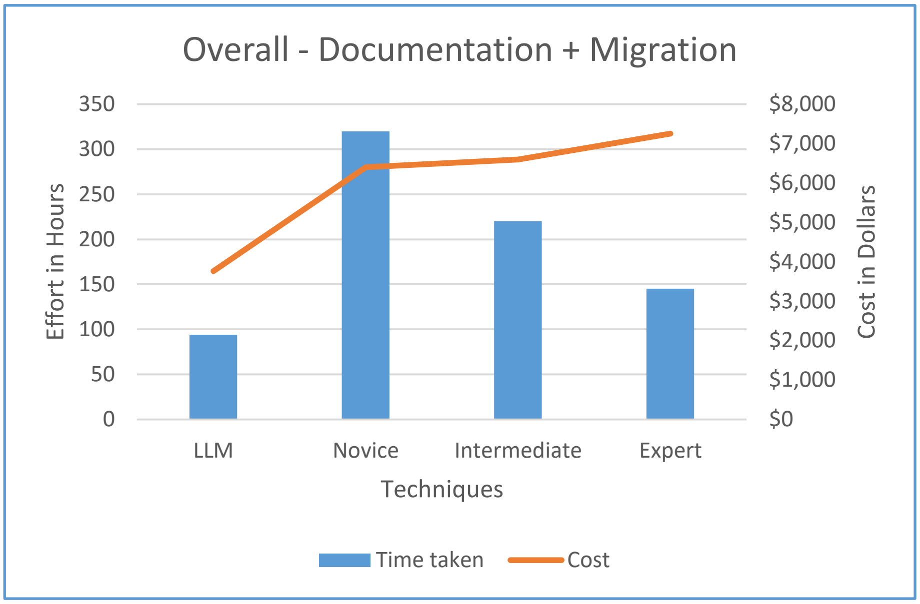


Figure 59
Cost and Effort - End-to-end process

Similarly, the graphs below show the percentage effort and percentage cost savings we get in different steps of documentation and code conversion. Figure 62 also shows the percentage effort and cost savings obtained in the overall process.

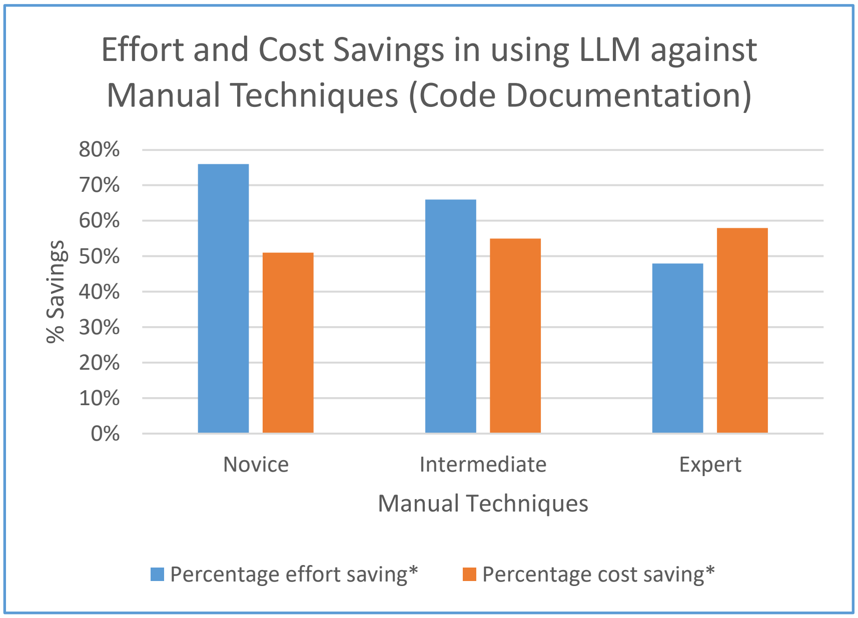


Figure 60
Effort and Cost Savings - Code Documentation

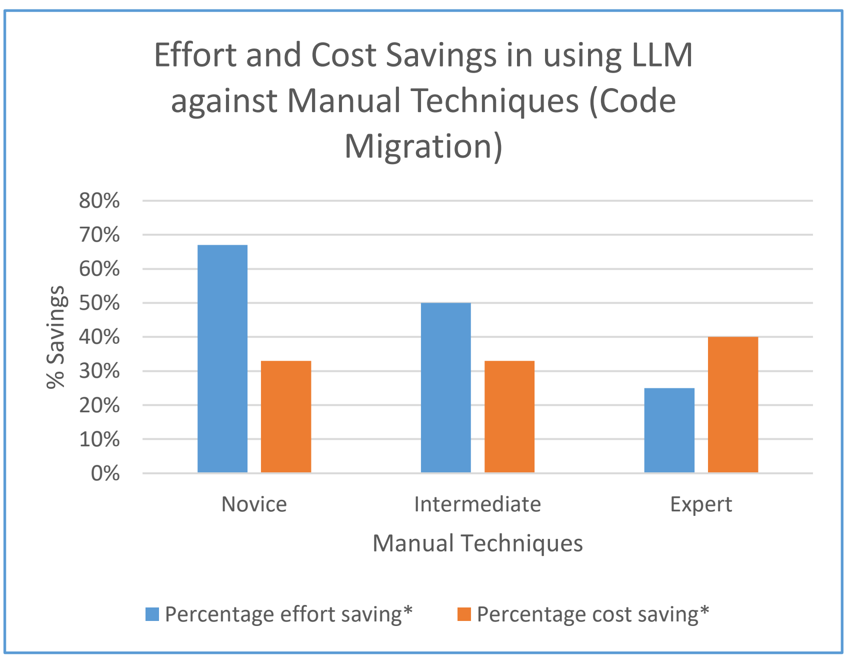


Figure 61
Effort and Cost Savings - Code Migration

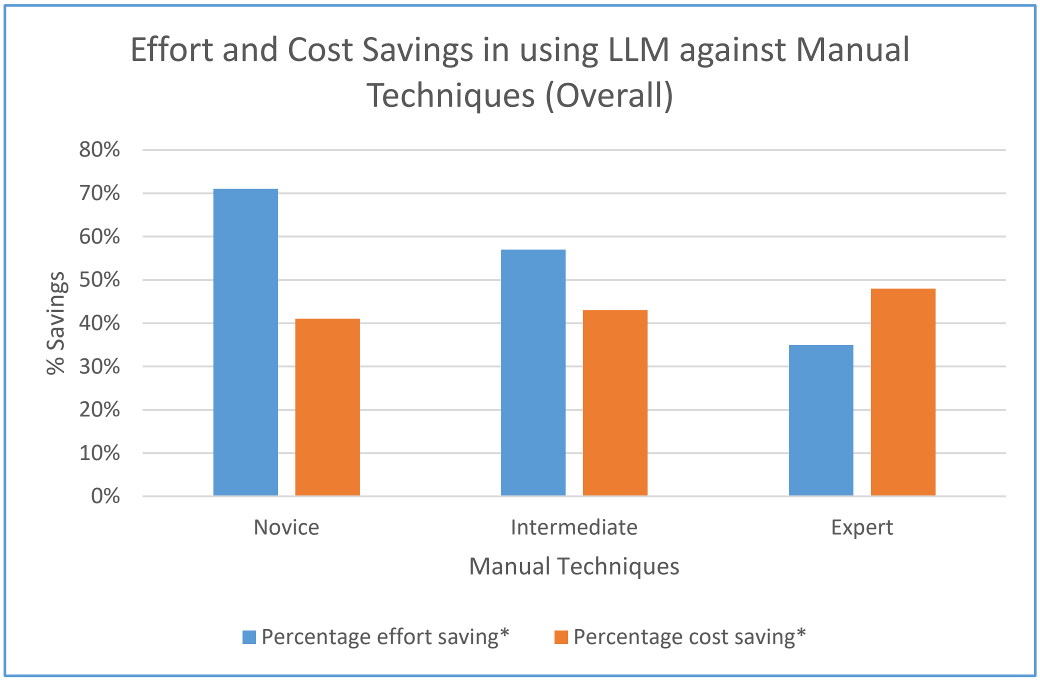


Figure 62
Effort and Cost Savings - Manual Techniques

We have seen that the accuracy of the translated code was almost 80%.

Cost-Benefit Analysis (with respect to Expert programmer)

Table 92
Cost-Benefit Analysis (End-to-end process)

	Manual Code Writing	**Code Generation using AI model**	**Saving**
LOC	1300	1300	
Approximate % code which is generated correctly		80%	80%
Developer’s effort (hours)	145	74	
% Effort saving using AI		49%	49%
Developer’s rate ($ per hour)	50	40	
Developer’s cost to client ($)	7250	2960	
Manual effort involved (correction/validation) in hours		20	

Effective % Effort Saving (Overall)		35%	35%
Cost of manual effort (correction/validation) needed with AI model		800	
Cost of API usage		2	
Total cost to client	7250	3760	48%

As in previous cases, the application for which the code documentation was generated and then migrated to Java was of simple to medium complexity. All the points mentioned in the previous sections apply to this task too when leveraging AI models for translating large and complex applications. In addition, for legacy code migration, a few things add to the complexity. Legacy languages like COBOL are procedural and target languages like Java are object-oriented. Hence, as we discussed in the approach, this is a two-step process – where we first generate documentation and then do the code generation. There will additionally be other steps that will be manual like creating the design diagrams (UML diagrams) and re-architecting the entire target application. These factors also amplify the intricacy of the task in the case of large complex applications. As mentioned earlier, combining different techniques including rule-based and tool-based approaches can provide better efficiency in performing these tasks.

Let us consider the case where a COBOL application with millions of lines of code must be re-architected and transitioned to a Java application.

There are a few points to be taken care of when migrating COBOL to Java:

1. Understanding the business requirements

   Before starting the migration process, it's essential to have a clear understanding of the business requirements that the COBOL system is currently fulfilling. This will help in identifying the critical areas of the system that need to be migrated first and ensure that the new Java system meets the business needs.

2. Mapping COBOL constructs to Java

   COBOL and Java are two different programming languages with different syntaxes, semantics, and libraries. It's important to map COBOL constructs to their Java equivalents, such as COBOL data structures to Java classes, COBOL subprograms to Java methods, and COBOL libraries to Java packages.

3. Handling data types

   COBOL uses a different set of data types than Java, which can lead to compatibility issues during the migration. It's important to understand the data types used in the COBOL system and map them to appropriate Java data types.

4. Handling I/O operations

   COBOL and Java have different ways of handling I/O operations. The migration process should include refactoring the COBOL I/O operations to work with Java equivalents.

5. Testing and validation

   The migration process should include comprehensive testing and validation of the new Java system to ensure that it meets the business requirements and performs as expected.

6. Performance optimization

   COBOL and Java have different performance characteristics. The new Java system should be optimized for performance to ensure that it can handle the same level of workload as the COBOL system.

7. Data security: The migration process should consider the security implications of moving the data from the COBOL system to the Java system. Appropriate measures should be taken to ensure the security and integrity of the data during the migration process.

Let us assume that the COBOL application to be migrated has 40 million lines of code. Considering the above points, the following table shows the cost-benefit analysis for building this application manually versus when assisted with an AI model.

Table 93
Cost-Benefit Analysis (Large Application)

		Manual Code Writing	**Code Generation using AI model**	**Saving**
LOC		40,000,000	40,000,000	
Approximate % code which is translated correctly			40%	40%
# Avg. LOC per person per day		250	1000	
Developer's effort (hours)		1,280,000	320,000	
% Effort saving using AI			75%	75%
Average rate ($ per hour) for developers		40	40	
Developer's cost to client ($)		51,200,000	12,800,000	
Manual effort involved (correction/validation) in hours			768000	
Effective Hours Overall			1,088,000	
Effective % Effort Saving (Overall)			15%	15%
Cost of manual effort (correction/validation) needed with AI model			31000000	
Cost of API usage			10000	
Total cost to client ($)		51,200,000	43,800,000	15%

Thus, we can see the client will see a benefit of around 7 million dollars, which is a 15% reduction in the manual cost.

Conclusion

From these studies, we can conclude that the following factors influence the effort and hence cost saving when leveraging an AI model:

1. The model's capability and understanding of the programming language/s involved.
2. The complexity of the application and interdependencies
3. The effort required to generate code relevant to the problem statement (prompt-engineering effort)
4. Percentage of manual effort involved in correcting/validating the code
5. The effort involved in deploying an open-source model (if required)
6. Cost of infrastructure for hosting the model (if required)
7. The effort involved in fine-tuning the model (if required)

We call this the seven-point framework.

5.4.3 Seven-point framework

Based on the above discussion, we align the points mentioned above in three phases:

A. Model Discovery phase

   This is the phase in which we identify the right AI/ML model that will be most effective in solving the problem. This phase would involve experimenting with different models and reviewing the published metrics and scores.

   The following two points from the seven-point framework fall in this phase:

   1. Complexity of Application

      Understanding the complexity of the application will help ease the process of identifying the right model for the task. There are various sub-parameters that we propose to measure the complexity of the application. Based on the value selected for the sub-parameter, a score will be assigned to the sub-parameter. Then an average score will be calculated and a value will be assigned to the main parameter. The same process will apply to all the parameters and sub-parameters described next.

i. Total Lines of Code (LOC) – As part of this framework, and based on the various experiments conducted, we can define the following values for this sub-parameter:

   a. < 5000
   b. Between 5000 and 50000
   c. > 50000

   This becomes one of the metrics to measure the complexity of the application.

ii. Individual file size – As we have discussed so far, all AI models have some limitations on the maximum lines of code they can read in one API call. This limitation is set by the model's context window. At the time of this study, the largest context window is provided by GPT-4, which is 32k tokens (including input and output). Hence, the size of the individual files involved becomes a determining factor for measuring the complexity of the application. If the individual files are large, then it would involve breaking these files into chunks, thus losing context and increasing complexity. The values we define for this sub-parameter are:

   a. Small
   b. Medium
   c. Large

iii. Frameworks / Components / DB involved – This is another sub-parameter that can be used to measure the complexity of an application. As more and more components are added to it, the application becomes increasingly complex. The values we define here are:

a. None
b. Few
c. Many

iv. Interdependency between code components – This is linked to the fact that AI models have limited context windows. If there is a lot of dependency between different code components, the AI model may not be able to capture the same. Hence, this increases the complexity of the application. The values for this sub-parameter are:

a. None
b. Few
c. Many

By considering the values for each of the sub-parameters, and after calculating the scores and average score, one final value will be assigned to the "Complexity of Application." The value assigned will be one from:

Simple, Medium, or Complex

2. Model Capability

The second important point which is part of the model discovery phase is model capability. After we determine the complexity of the application, we also need to identify the right AI/ML model to be used for the task. Two sub-parameters can be considered to determine the model capability:

i. Published scores – When an AI/ML model is released, along with the task that the model can perform, the scores of the model are also released. These scores will differ based on the task the model performs and the evaluation criteria. In some cases, CodeBLEU score is published, while for some models the accuracy is published. Reviewing the published score is one of the ways of measuring the model's capability. The following values can be defined for this sub-parameter:

a. Very Low – the score published for the model is very low
b. Low – the score is on the lower side
c. Average – the score is average as compared to other models
d. High – the published score is high compared to the scores of other models

ii. Initial testing – In addition to reviewing the published scores, one must perform some experiments related to the programming language and task at hand. Based on the experimentation, the following values can be assigned to this sub-parameter:

a. Poor
b. Average
c. Good
d. High

Based on both the above sub-parameter values, a single value will be assigned to the model capability parameter. The value that will be assigned would be one from:

Poor, Average, Good, or Excellent.

B. The second phase is Setup Consideration

Once we have completed the model discovery, we must now identify the setup considerations for inferencing, deploying, or fine-tuning the model. The following points from the seven-point framework will be part of this phase:

3. Deployment Effort

   Based on the model identified, a model may be open-source which must be deployed before consumption, while in other cases, the model may be deployed on external servers and only available as a service. If the model is available as a service, there will be no deployment effort. However, in other cases, depending on the model size, the deployment effort will vary. Hence, we define the following values to be selected for this parameter:

   a. Small model – if a model is less than 15 GB, the deployment effort will be low
   b. Medium-sized model – if the model is less than 60 GB, an average effort will be required for deploying the model
   c. Large model – for models which are larger than 60 GB, the deployment effort will be considerable
   d. NA – if the model to be used is available only as a service, the value for this sub-parameter will be "NA."

   Depending on the values selected and the calculated scores, the final value for the "Deployment Effort" parameter will be one from:

   NA, Low, Medium, High

4. Fine-tuning effort

   Depending on the capability of the model, the value for this parameter will be auto-calculated. When fine-tuning a model, the effort involved depends on the following sub-parameters:

   i. Labeled data for fine-tuning the model – if the model is having an average understanding, it means the model was not exposed to the programming language as part of its learning process. Hence, considerable size of labeled data will be needed for tuning the model. On the other hand, if the model already has a very good understanding of the programming language, the magnitude of data required to further tune the model will be considerably low.

ii. Data preparation effort – The effort involved in collecting and preparing labeled data will vary depending on the volume of data needed to fine-tune the model.

iii. Similarly, the effort spent on various experiments and hyperparameter tuning will vary depending on the model capability.

Thus, we observe that all these sub-parameters largely depend on the capability of the model. Hence, the value for "Fine-tuning" effort is derived from model capability and would be assigned one of the following values:

NA - if API based model is used,

Not required - if the model capability is excellent

Low – if model understanding of programming language is good

Medium – if the model has an average understanding of the programming language

High – if the model has a poor understanding of the programming language

5. Infrastructure Cost / API cost

The next critical parameter in setup considerations is the "Infrastructure Cost / API Cost" involved. This cost depends on two sub-parameters:

i. Size per GPU

Depending on the size of the model being used for deployment or fine-tuning, the GPU size needed for deploying, inferencing, or fine-tuning the model will vary. This will in turn impact the cost involved. The following values can be assigned to this sub-parameter:

a. <= 20 GB – if a small GPU suffices for fine-tuning or inferencing the model

b. > 20 GB and <= 40 GB – this is suited for medium-sized models

c. > 40 GB – most large models need more than 40 GB GPU RAM

ii. Count of GPUs

Depending on the size of the model and fine-tuning requirements, the number of GPUs needed for inferencing or fine-tuning a model will vary. For inferencing with small GPUs, most times a single GPU is sufficient. However, for fine-tuning a small model, more than one GPU may be needed. This also affects the cost of infrastructure. The following values are defined for this sub-parameter:

a. < 2 – sufficient for inferencing with small GPUs and sometimes for fine-tuning small GPUs

b. Between 2 to 8 – most medium-sized GPUs would need between 2 to 8 for inferencing or finetuning the model

c. > 8 – large models will generally need 8 or more GPUs, only for inferencing

As in the other cases, based on the values selected for these sub-parameters, a score will be assigned to each sub-parameter, an average score calculated and a final value from one of the below values will be assigned to "Infrastructure Cost / API Cost" parameter:

NA, Low, Medium, High

C. Implementation

This is the last phase of the framework. The following two points from the seven-point framework get aligned to this phase:

6. Prompt-tuning effort involved

   The values for this parameter will be auto-calculated from points 1 and 2 in the model discovery phase. The extent of effort needed to tune the prompt for task execution is contingent on the complexity of the application and the model's capabilities. For example, if the application is of simple complexity and the model capability is excellent, then the effort involved in tuning the model will be low. However, if the application is complex and the capability of the model is excellent, the tuning effort will increase. Considering these various possibilities, the following values are auto-assigned to this parameter:

   Low, Average, High, Very High, Opt for Manual

Table 94
Calculating prompt-tuning effort

Model Capability	**Complexity of Application**	**Prompt-tuning effort**
Excellent	Simple	**Low**
	Medium	**Average**
	Complex	**High**
Good	Simple	**Low**
	Medium	**Medium**
	High	**High**
Average	Simple	**Average**
	Medium	**High**
	High	**Very High**
Poor	-	**Opt for Manual**

As it can be observed, if the model capability is poor, it is suggested to opt for a manual approach. This is because if the model does not have an understanding of the programming language, then even for a simple application, the fine-tuning effort will be large. For a complex application, fine-tuning will not only add to the effort but also the cost. Hence, for a "Poor" model capability, performing the task manually or using rule-based techniques, should be preferred.

7. The last parameter from the seven-point framework is "Manual correction." As we have seen, different AI models can perform different tasks and support different programming languages. However, we cannot expect any model to provide a 100% accurate code. These models can provide a template code that must be manually reviewed and corrected. This parameter measures the effort and cost involved in correcting the code generated by the AI model. The value for this parameter, as in the previous case, is auto-calculated. This also depends on points 1 and 2 from the model discovery phase. Similar, to point 6, the values for this parameter are:

   Low, Medium, High, Very High, Opt for Manual

   These values are calculated as follows:

Table 95
Calculating manual correction effort

Model Capability	Complexity of Application	Manual correction effort
Excellent	Simple	**Low**
	Medium	**Medium**
	Complex	**High**
Good	Simple	**Medium**
	Medium	**High**
	High	**Very High**
Average	Simple	**Medium**

	Medium	**High**
	High	**Very High**
Poor	-	**Opt for Manual**

This framework can now be applied to calculate the effort and cost saving for any given task.

Effort saving

Effort saving will be measured by considering the following points:

1. Initial effort saving by using a pre-trained model

   The first effort saving is caused by leveraging the AI model for automated code generation. The template code or documentation generated by the model saves the effort of manually writing the code. By considering the final value from point 6, the initial effort saving can be calculated as follows:

Table 96
Initial Effort Saving Calculation

Prompt-engineering effort	**Initial effort saving**	**Reason**
Low	> 75%	If the prompt-engineering effort required is low, since the model capability is excellent, then the effort required to produce the template code will be far less as compared to writing the code manually. This would result in a high initial effort saving. However, as the model capability reduces and the prompt-engineering effort increases, it would mean, generating even template code will require a large effort and hence reduced initial effort saving.
Average	Between 50% to 75%	
High	Between 25% to 50%	
Very High	< 25%	

2. Effort saving after manual correction

After the initial code has been generated, there will be extra effort involved in manually correcting the code. This will cause the initial effort saving obtain to reduce. The effort saving after manual correction is calculated as shown below:

Table 97
Calculating change in effort saving due to manual correction

Manual correction	**Effort saving**	**Reason**
Low	Reduction in effort by 25% to 45%	The value of the manual correction parameter is auto-calculated based on the complexity of the application and model capability. Both these factors together indicate the accuracy of the model. Thus, if the manual effort value is low, it means that the model accuracy was good. Hence, the effort saving will be reduced by 25% to 45%. However, as the manual effort correction value moves from low to high, the reduction in effort saving will rise gradually. If the manual correction needed is very high, then the effort reduction will be more than 90%. This will considerably reduce the overall effort, even if the initial effort saving is high.
Medium	Reduction in effort by 45% to 70%	
High	Reduction in effort by 70% to 90%	
Very High	Reduction in effort by more than 90%	

3. Deployment effort

When using API-based models like OpenAI GPT-3 or Codex, since these models are already hosted, there is no deployment effort involved. However, when using open-source models, extra effort will be involved in deploying the model. The “Deployment effort” parameter from the framework guides how the effort saving will be affected. The following table indicates the same.

Table 98
Calculating change in effort saving due to deployment effort

Deployment Effort	Effort saving	Reason
NA	No deployment effort	The value of the deployment effort parameter depends on the size of the manual selected during the model discovery phase. If the deployment effort is low, then the reduction in effort saving will be by a smaller percentage of approximately 10%. However, as the deployment effort increases, the effort saving further reduces.
Low	Further reduction in effort by 10%	
Medium	Further reduction in effort by 20%	
High	Further reduction in effort by 30%	

4. Fine-tuning effort

   Similarly, if fine-tuning is needed, then the effort required for the same must also be factored in. This will further reduce the initial effort saving. The “Finetuning effort” point guides the further reduction in effort. The following table indicates how the effort-saving changes due to this parameter.

Table 99

Calculating change in effort saving due to fine-tuning effort involved

Fine-tuning Effort	**Effort saving**	**Reason**
NA	Inference only API-based model	The value of fine-tuning effort parameter is auto-calculated based on the model capability. The value is NA if the model is API-based, large model and fine-tuning is not possible. If the model capability is Excellent, fine-tuning will not be required, hence there will be no reduction in effort saving. For other cases, the effort saving will be reduced as shown in the table.
Not required	No further reduction in effort	
Low	Further reduction in effort by 50%	
Medium	Further reduction in effort by 70%	
High	Further reduction in effort by 90%	

Cost saving

As in the case of effort saving, cost saving is calculated by considering the values of different parameters in the seven-point framework.

Case 1 – No fine-tuning required

1. Cost incurred due to the effort spent in prompt-tuning

   This cost is calculated based on the "Prompt-engineering effort" parameter from the seven-point framework.

Table 100
Cost incurred due to effort spent in prompt-tuning

Prompt-engineering Effort	**Cost incurred**	**Reason**
Low	< 20%	The prompt-engineering effort parameter is auto-calculated by considering both the complexity of the application and the capability of the model. If this parameter value is low, it means, not much effort is required in performing the prompt engineering to perform the desired task. Hence, the cost incurred will be < 20%. This means the cost saving will be significant, ~80%. However, as the value of this parameter moves from low to very high, the cost will increase. When the prompt-engineering effort is very high, it means the cost incurred will be ~70% of the manual cost. As a result, the cost saving will be very less.
Average	Between 20% to 45%	
High	Between 45% to 70%	
Very High	~70%	
Opt for Manual	Same as manual	

2. Additional cost due to manual correction

   As we have seen in effort savings calculation, the code generated by the AI model is not always accurate, and manual validation and correction is always required. Here, we calculate the additional cost incurred due to manual corrections. This cost is calculated by considering the value in the "Manual correction" parameter from the seven-point framework

Table 101
Additional cost incurred due to manual correction

Manual correction	**Additional cost incurred**	**Reason**
Low	~ 20%	The manual correction parameter is auto-calculated by considering both application complexity and model capability, giving an idea of model accuracy. Thus, if the manual correction required is low, then the additional cost incurred will be less, ~20%. However, as the manual correction required grows from low to very high, the additional cost incurred also increases. This will reduce the final cost saving obtained.
Medium	Between 20% to 40%	
High	Between 40% to 60%	
Very High	Between 60% to 80%	
Opt for Manual	Same as manual	

Case 2: Fine-tuning required

When fine-tuning is required, the points to consider when calculating the cost incurred will vary as other costs like infrastructure and deployment costs will need to be considered.

1. Infrastructure cost

   The infrastructure cost will depend on two factors:

   a. The size of compute and number of compute instances required
   b. The duration for which the compute will be used

   Both these factors are determined by “Fine-tuning effort” and “Infrastructure cost” parameters from the framework. Thus, the following table indicates how the infrastructure cost is affected by these parameters.

Table 102
Calculating cost incurred due to infrastructure

Fine-tuning Effort	**Infrastructure cost**	**Cost incurred**	**Reason**
Low	Low	10%	The fine-tuning effort is auto-calculated based on the capability of the model. If the model capability is Poor, the fine-tuning effort required will be high. This means the duration for which the compute will be required, will be more. Similarly, if the model capability is good, the fine-tuning effort will be low, which means the duration for which compute is required will be less. The value for the infrastructure cost parameter depends on the size of the compute and the number of compute instances. Thus, if both fine-tuning effort and infrastructure cost is low, the additional cost incurred will be low, approximately 10%. However, if fine-tuning effort and infrastructure requirements are both high, cost incurred will be approximately 70%.
Low	Medium	20%	
Low	High	30%	
Medium	Low	20%	
Medium	Medium	40%	
Medium	High	55%	
High	Low	30%	
High	Medium	55%	
High	High	70%	

2. Deployment effort cost and fine-tuning cost

   The deployment effort cost and fine-tuning cost will depend on two factors:

   a. The effort involved in deploying the model

b. The effort involved in fine-tuning the model

Both these factors are determined by "Deployment effort" and "Fine-tuning effort" parameters from the framework. The following table indicates how these parameters affect the additional cost incurred.

Table 103
Cost incurred due to deployment effort spent and fine-tuning effort

Deployment Effort	**Fine-tuning Effort**	**Cost incurred**	**Reason**
Low	Low	10%	The deployment effort depends on the size of the model. If the model size is small, the deployment effort is low. However, the deployment effort is high, if the model size is large. Also, the fine-tuning effort is auto-calculated based on the capability of the model. If the model capability is Poor, the fine-tuning effort required will be high. Similarly, if the model capability is good, the fine-tuning effort will be low. Thus, if both deployment effort and fine-tuning effort are low, the additional cost incurred will be low, approximately 10%. However, if deployment effort and fine-tuning effort are both high, the cost incurred will be approximately 70%.
Low	Medium	20%	
Low	High	30%	
Medium	Low	20%	
Medium	Medium	40%	
Medium	High	55%	
High	Low	30%	
High	Medium	55%	
High	High	70%	

3. Cost of inferencing and manual corrections

The effort involved in inferencing the fine-tuned model and performing manual corrections depend on:

a. Complexity of the application
b. Model capability

Table 104
Cost of inferencing fine-tuned model and doing manual corrections

Complexity of application	**Model capability**	**Cost incurred**	**Reason**
Simple	Excellent	10%	If the model capability is good, the fine-tuned model will give better accuracy and hence reduce the manual corrections required. Similarly, if the complexity of the application is small, the effort involved in inferencing the fine-tuned model and hence the associated cost will be less. Thus, we see that if complexity is simple and model capability is good, then the cost of inferencing and manual corrections is less. Similarly, if the application is complex and model capability is average, the time and hence cost of inferencing will be high, and manual corrections effort will also be high. Hence, the cost incurred in this case will be high, approximately 70%.
	Good	20%	
	Average	30%	
Medium	Excellent	30%	
	Good	40%	
	Average	50%	
Complex	Excellent	50%	
	Good	60%	
	Average	70%	

By considering all these factors, the final cost saving incurred will be calculated.

In the appendix, we show sample effort and cost savings obtained when applying this framework to two different scenarios.

We had hypothesized that:

Hypothesis 2

If large language models are to be used optimally, then a human-in-the-loop is necessary for validation of the result generated by the model.

Hypothesis 3

If code automation using large language models is leveraged in the industry, it can improve software quality by reducing errors and increasing consistency.

Hypothesis 4

If large language models are used for code automation, it can improve software development efficiency by reducing the time needed to develop code and provide significant cost and effort benefits to software organizations.

The above results and discussions prove that all three hypotheses are true.

5.5 Research Question Four

What types of jobs will be completely replaced, if any, due to the adoption of AI in the software industry?

In section 4.5, we explored how large transformer models can be leveraged for performing different jobs in the software industry. We have seen that these models help reduce the development effort in various tasks, right from requirements gathering and software development to testing.

We discussed the different tasks LLM can help augment in the different phases of the software development cycle, like

1. Planning and Requirements
2. Design

3. Coding and Testing
4. Software testing and quality assurance
5. Code refactoring and optimization
6. Enhancing technical documentation
7. Teaching programming language

In all the above phases, though AI models can be leveraged for various tasks, validating, and correcting the task done by the model will still be needed. Especially, in the code domain, though these models produce authentic-looking code and documentation, the generated code or documentation will not always be accurate. Hence, manual review and validation will always be required when leveraging these models.

In these tasks, the effort saving primarily comes from the effort spent in searching different sources on the internet, to help solve a problem in the software development life cycle. Large language models provide pointed answers to the task and for code-related tasks can generate template code or documentation for the user, which must further be refined and optimized. Thus, though capable of reducing the effort spent in performing a task, AI models cannot replace a job completely. At the same time, the adoption of these models does free up the bandwidth of the developers and allows them to focus on higher levels jobs.

For example, in the requirements phase, a person can now focus on higher-order decision-making jobs like working with stakeholders, defining the scope of the project, and collaborating with different teams. In the design phase, the effort saved by automating certain tasks can be utilized by focusing on other jobs requiring higher-order thinking skills like designing for a scalable, maintainable, and more secure system. Similarly, in the development phase, the developer whose effort is saved due to auto code generation or translation can now focus on jobs like architecting the applications, creating design documents, working on performance improvements, and looking into security constraints. In the testing phase, AI can help automate test case generation. However, validating that the generated test cases are accurate and correcting them will still require manual effort. Moreover, individuals can also focus on other aspects like ensuring that the test cases align with best practices and they meet the quality standards.

Similarly, we have seen that AI models can suggest refactored and optimized code. This saves time for the developers to perform the optimization from scratch. However, the refactored code will need to be reviewed and different factors like performance and scalability would have to be considered. This is where the individual bandwidth saved by automation can be directed.

We also saw that AI models can generate technical documentation. However, the quality of the documentation may not be as per industry standards. This is because the AI models are generic. Hence, they cannot capture or generate documentation that is domain specific. Thus, human writers will still be needed to add more context and domain knowledge to the documentation.

We also experienced as part of the various experiments, that the AI model can help programmers learn a new programming language. The model can explain what a piece of code does with details about the variables used and reasoning. However, as in the other tasks, this information may not always be correct. The AI model can help speed up learning a programming language, but it lacks the domain knowledge which can help cover different aspects of the programming language. For this purpose, an individual with an expert understanding of the programming language will always be needed.

We had hypothesized that:

If code automation using large language models is adopted in the software industry, it can improve software development flexibility by allowing developers to focus on higher-level tasks and by providing more options for code generation.

Based on the results and the above discussion, this hypothesis has been proven true.

5.6 Research Question Five

What is the new set of skills that will emerge as AI is increasingly adopted across the organization for software development?

We discussed in section 4.6 that AI models are rapidly evolving and becoming more efficient to solve a variety of problems in software development. However, working with these models requires a new set of skills to make the model understand and learn better and for the model to produce the desired output with better accuracy.

The skills we discussed are:

- Prompt engineering or prompt tuning

  This is one of the critical skills that will be required when working with AI models. As we have already seen, AI models are well-read and have a lot of knowledge. However, they must be given the right instructions to get the right output. Contextualizing the model by adding more domain knowledge and context in the prompt will be required to get the best results. Knowing how to produce this prompt and tune it further based on the requirement will be one of the critical skills required.

- Analytical and debugging skills

  These skills though not new, would be required and will be critical when working the LLMs. As we have seen in the entire study, after the result is generated by the model, manual review and correction will still be needed. Here, the analytical and debugging skills of individuals will be more important. As we have seen earlier, a generated piece of code that may be appearing correct may still contain errors or bugs.

Analyzing the code, debugging the code, and correcting the code followed by further tuning the prompt to produce better code, will be required in such cases.

- Collaboration and communication skills

  These skills will also become critical as different teams would need to collaborate to get the desired results. When using these AI models, collaboration with teams who have a business understanding of the problem, architecture team, design team, and also teams that automate tasks using rule-based tools may be required. Having a 360-degree view of the problem to be solved can help the individual provide the right context to the model and hence generate better results.

- Creativity and innovation

  As the capability of the model improves, creativity and innovation become important skills that will be needed in the future. By understanding the strengths and limitations of the AI models, individuals should be able to innovate and come up with different ways and techniques how to maximize and apply the strengths of the model across the entire life cycle of software development. Hence, this becomes another critical skill to acquire.

- Model fine-tuning, responsible AI, and data-driven decision making

  While some AI models may be able to solve the problem at hand without further training, there may be some set of problems, for which further improving or fine-tuning the model will be required. Also, for finetuning a model, data understanding and preparation are most critical. The quality of data used in the fine-tuning process will greatly impact the quality of results obtained from the fine-tuned model. At the same time, ensuring that all ethical considerations and responsible AI principles are followed will be equally important. Hence, it will be important to acquire all these skills as we move towards AI-assisted software development.

We hypothesized:

Hypothesis 6: If code automation using large language models is adopted in the industry, then reskilling software professionals is necessary.

As we see from the above discussion, this hypothesis is true.

Our last hypothesis was:

Hypothesis 7

If code automation continues to increase in popularity and effectiveness, it will have a significant impact on the software industry and software professionals.

We discussed in section 4.2.2 how large language models will have a significant impact on the software industry and software professionals.

We observed how AI-assisted software development will impact the workforce, business, and ways of working in both services-based and product-based industries. As AI models become more efficient, there may be a decrease in entry-level developers as some of the tasks may be automated. There will also be a shift in the skills required to work with these models, as discussed in the previous section.

We have also seen from the results of various experiments conducted in this study, how LLMs can provide significant effort and cost savings in different phases of software development. We also suggested a seven-point framework, which can be used by both services-based as well as product-based companies to measure the effort and cost savings they can get by automating different tasks.

The way of working will also be impacted, for both categories of industries. There will be a need to train employees to make use of AI in an intelligent and responsible manner. A more collaborative and hybrid approach will emerge as the adoption of large language models grows in the industry.

CHAPTER VI:

SUMMARY, IMPLICATIONS, AND RECOMMENDATIONS

6.1 Summary

As part of this research, we aimed to study the impact large language models will have on the software industry and software professionals. We listed the research questions and hypotheses.

We did various experiments in a controlled environment and noted the results of the study. Based on the various results obtained, we see that efficient LLMs like Codex, and ChatGPT can perform many code-related tasks. They can also provide significant effort and cost savings if the application complexity is simple. We also studied that, when industries do not want to use API-based models due to data privacy issues, other open-source models like CodeGee, GPT-Neox, and Bloom may be utilized.

We present a seven-point framework, which we derive based on our studies and experiments. This framework takes into account seven points that can help assess the cost and effort impact LLMs can have when leveraged during the software development lifecycle. The key points of this framework are…..

We also identified the new skillset that will be required when AI-assisted programming must be used. We also discussed how the way of working will be impacted and a hybrid approach will evolve.

6.2 Implications

The cost of software development and the lack of skilled talent are two major challenges that software industries face today. Traditional techniques for code automation are not very efficient. Training a machine learning model using traditional approaches like neural machine translation, requires large amounts of data and compute. This study focuses on assessing the impact that large transformer models have in the area of code automation and measuring their impact on the software industry.

The research findings present an illuminating perspective on the capabilities of large pre-trained models from the context of code automation. The study throws light on the various software development tasks that can be automated by using LLMs. These tasks are not limited to code development alone but can belong to any phase of the software development lifecycle. The study also indicates how the capability of LLMs in performing these tasks is either comparable to or even better than traditional approaches. These findings can help industries envision different areas where they would like to apply large transformer models for automating diverse tasks.

The findings also indicate that manual validation and correction will always be required due to the probabilistic nature of these models. Hence, these models can only be used to augment developers and not replace them. The study findings also give an indication of the amount of cost and effort saving one can get by leveraging these models. The seven-point framework presented in the study can help software organizations perform a cost-benefit analysis that they can get for different tasks, based on the complexity of the problem and the capability of the ML model. This can help them choose the right model for the problem to be solved and estimate the savings.

The seven-point framework will prove useful to assess the impact of problem complexity on automation. It would help an organization assess whether the cost of finetuning a model for a particular problem is worth the effort. It will also help them identify solutions that could involve combining multiple techniques along with LLM to save their effort and cost.

The findings from the study give insights into the skills that will be required in the future as the adoption of LLMs in the industry increases. Entry-level development work can be automated using LLMs and developers would need to upskill and reskill themselves. This study will help developers understand why such reskilling will be needed and how it will help them in their course of work.

Limitations

The study currently assessed how LLMs can be used for code automation using techniques like prompt-tuning and inferencing. Also, the problem statements that were used were of medium complexity and mostly related to modern programming languages. The aspect of fine-tuning a pre-trained model for a custom problem or training a transformer-based model from scratch on a custom dataset was not explored. These activities require large amounts of compute and hence, were kept out of scope for this research.

6.3 Recommendations for Future Research

Considering the limitations listed above, it is recommended to conduct future studies on smaller models, fine-tune them for custom problem statements and custom datasets, and apply the seven-point framework to arrive at the cost-benefit analysis in such cases. The study also mostly targets modern programming languages. Future studies must be done on legacy programming languages and assess the capability and impact that these models can bring in this space.

The case studies considered during the course were of medium complexity. Building on this study, further research is recommended to be carried out, to measure the impact of code automation when the task is complex. This study can help provide insights into the benefit these models can bring when there are a large number of files, where each file is large with many interdependencies on other components or frameworks. Such a study will help in assessing whether fine-tuning a model in such a case will provide the necessary benefit.

Also, the current study included only the task of requirements classification from the requirements elicitation phase. A similar study must be done for other tasks in both the requirements identification as well as the design phases. This study will help in getting a better understanding of the model's capabilities in performing various tasks, other than those mentioned in this study.

6.4 Conclusion

In this study, we aimed to assess the impact large pre-trained models will have on the software development industry. Our findings prove, that the revolution has already started. These models, which are pre-trained on code, can perform many code-related tasks and thus provide multiple benefits. The study assessed the efficiency of these models and also measured the cost and effort savings obtained by leveraging them. The study showed that these models can provide effort savings of more than 50% when the task is of simple to medium complexity and involves modern programming languages. For tasks involving legacy language like COBOL, the study saw an effort saving of around 30%. Similarly, based on the complexity of the task and the programming language involved, the cost savings will vary.

One pivotal outcomes of the research study was the suggested seven-point framework. This framework can serve as a guiding principle for organizations to assess and measure the benefits they will get when leveraging these models, based on the task and complexity of the application. The study also indicates that LLMs will only help augment software development and not replace their jobs. On the contrary, the freed-up bandwidth of the developers will help them focus on higher-order decision-making skills. Also, as AI-assisted programming becomes mainstream and more widely adopted by businesses, the study reveals a new set of skills that will emerge. Since the models can generate code, analyzing and debugging the generated code will be crucial. Hence, having strong analytical and debugging skills will be required. In addition, prompt-tuning, model fine-tuning, and collaboration and communication skills to effectively communicate with different teams to get the desired outcome, will be equally important.

To conclude, the research shows that as large language model adoption grows, the software development industry will see great benefits. Both service-based and product-based industries will see a change in the way of working, the workforce, and the business. As more efficient code automation models get released, the industry can see a much bigger and better impact.

APPENDIX A

SURVEY RESULTS

The following questions were asked to the participants during the online survey:

Sample Survey Questions:

1. Describe your role: Student, Programmer, Manager, Other
2. Rate your programming skills on a scale of 1 to 4
3. What programming language are you comfortable with:

   Java, C++, Python, .NET, Javascript, C, C#, Go, R, Swift, PHP, Perl, Ruby, Scala, Other
4. Time (in mins) you would take to code a simple Java program
5. Time (in mins) you would take to code a simple Python program
6. Time (in mins) you would take to code a simple C++ program
7. Time (in mins) you would take to code a medium complexity program
8. Time (in mins) you would take to code a complex program
9. Select reason for slow development time
10. Would you prefer to write code from scratch compared to using templates?
11. Would you prefer to make use of an automatic code generation utility for code completion?

The survey recorded 1,777 responses. The screenshots below indicate the responses received.

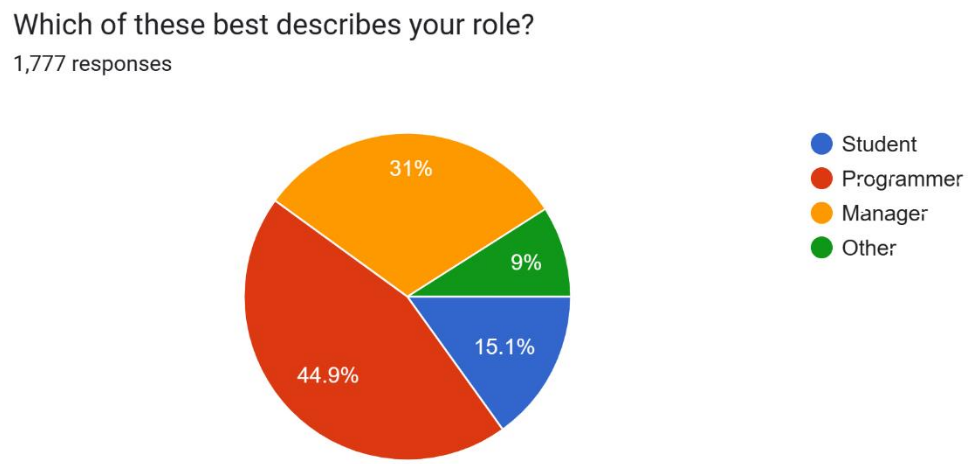


Figure 63
Q1 - Role of the participant

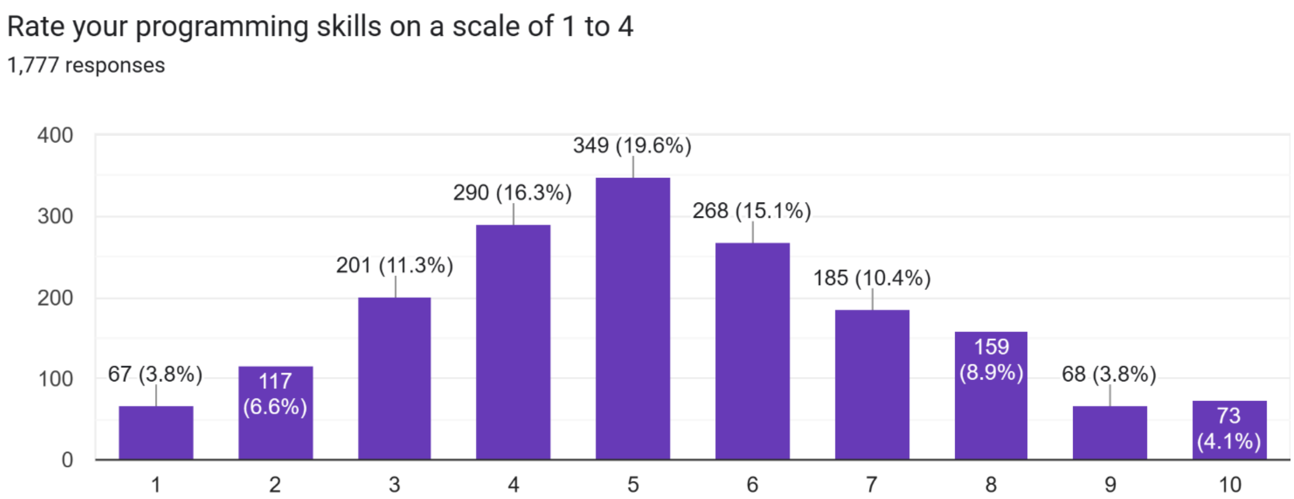


Figure 64
Q2 - Programming skills

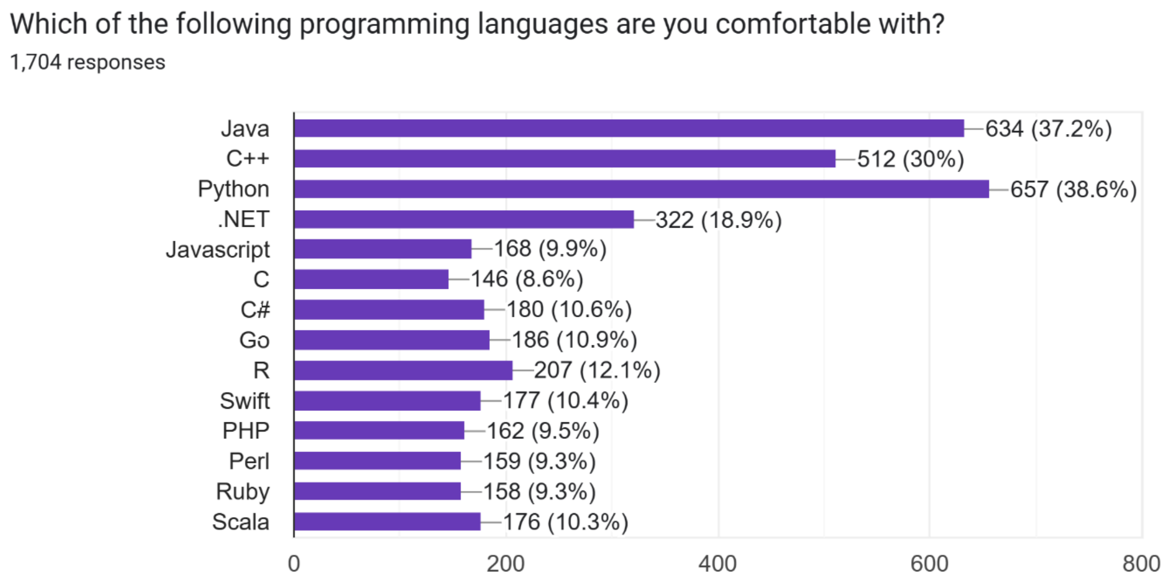


Figure 65
Q3 – Programming language preference

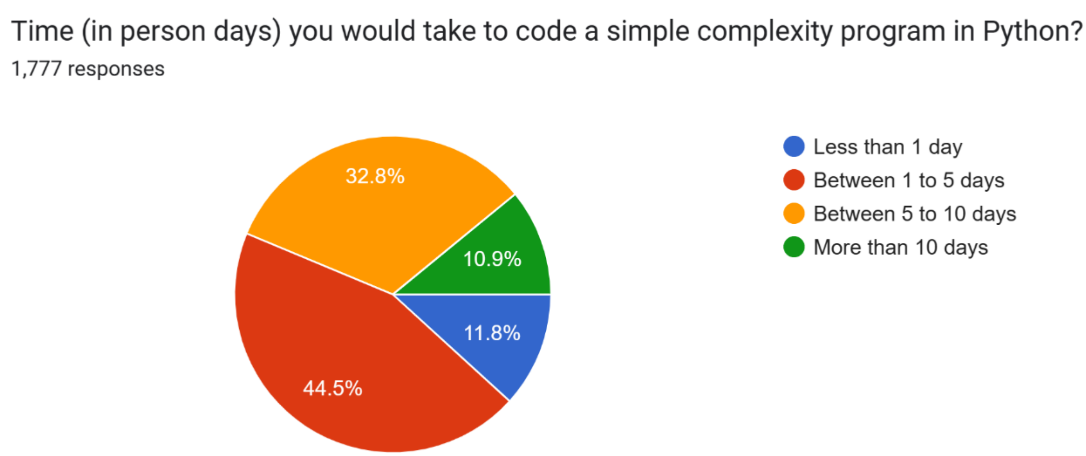


Figure 66
Q4 – Effort needed for simple complexity program in Python

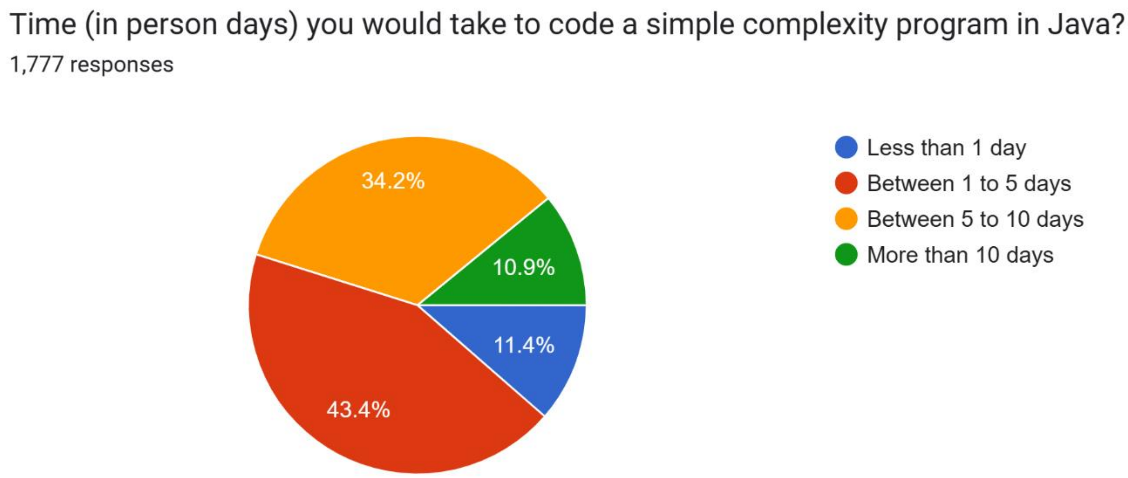


Figure 67
Effort needed for simple complexity program in Java

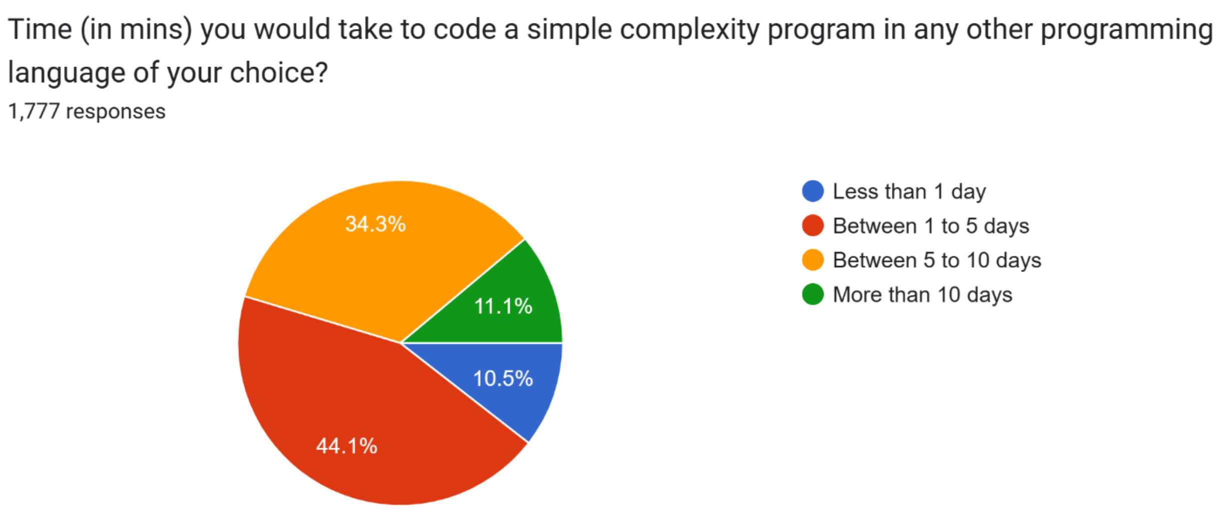


Figure 68
Effort to code simple complexity program in PL of choice

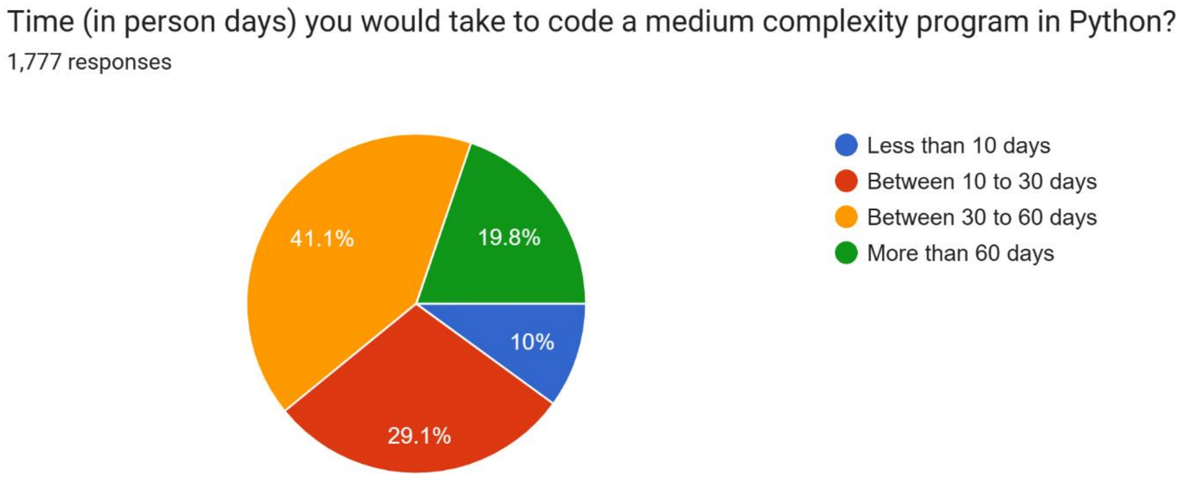


Figure 69
Effort needed to code medium complexity program in Python

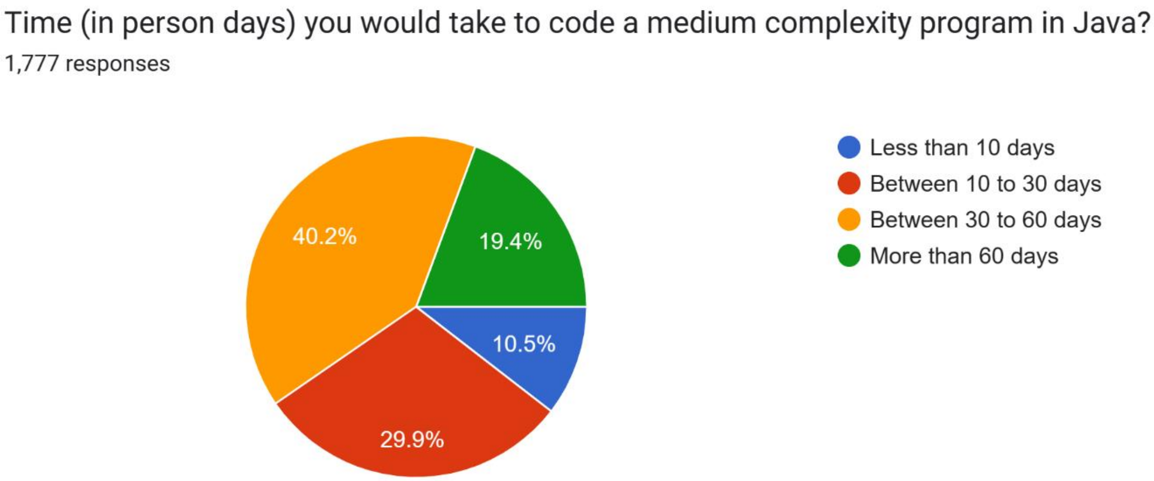


Figure 70
Effort needed to code medium complexity program in Java

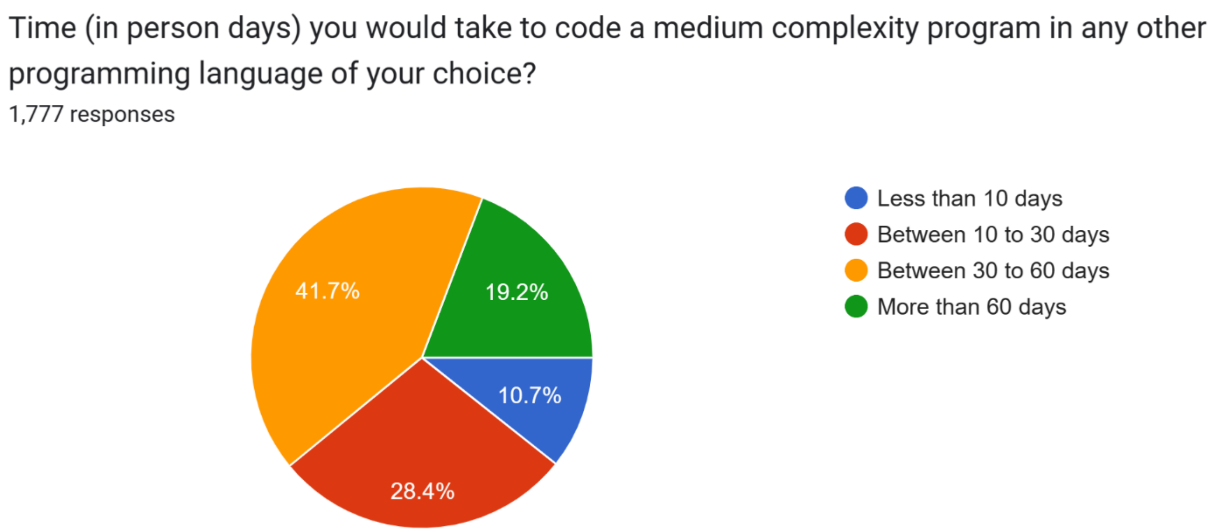


Figure 71
Effort to code medium complexity program in PL of choice

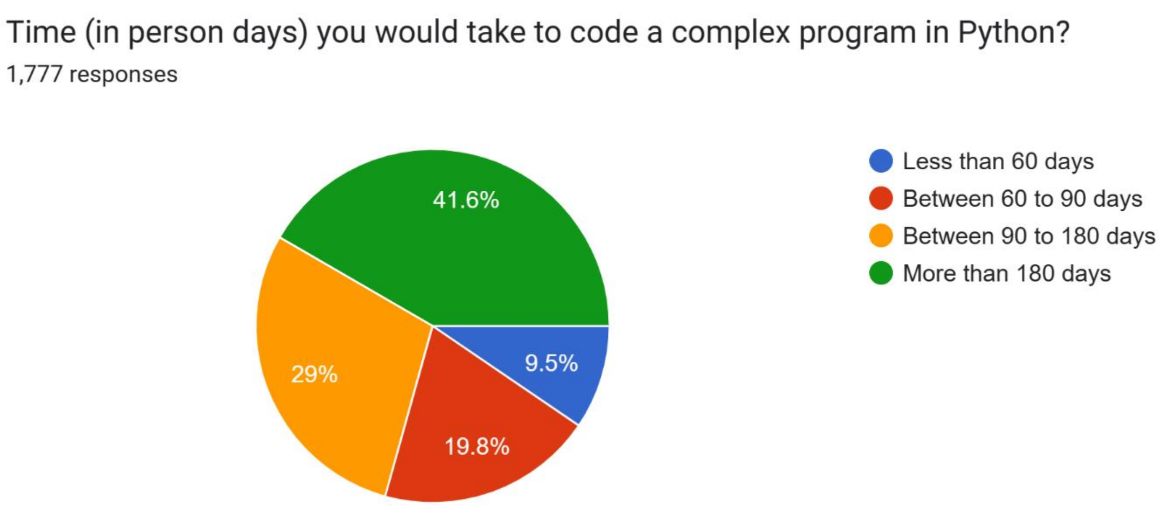


Figure 72
Time (in person days) to code complex program in Python

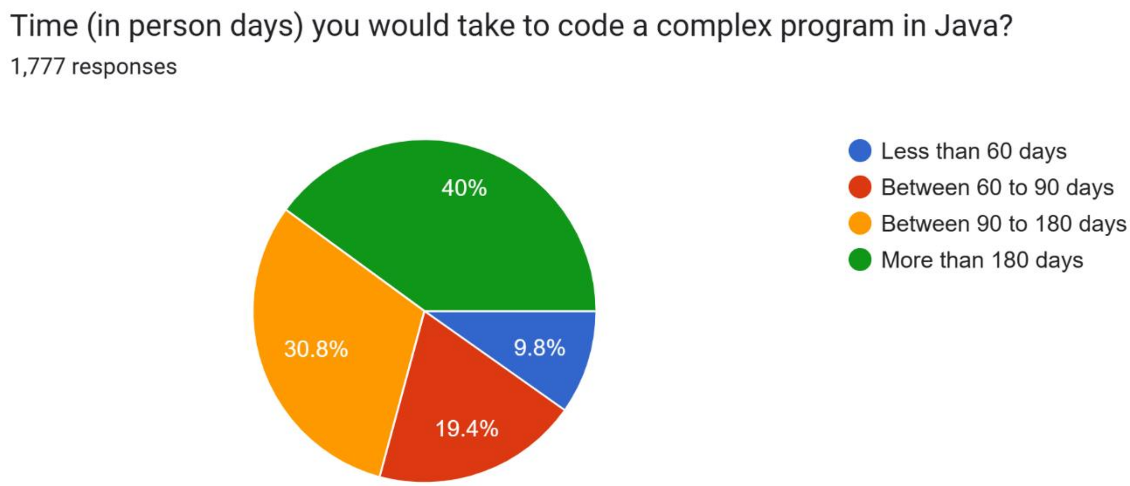


Figure 73
Time (in person days) to code complex program in Java

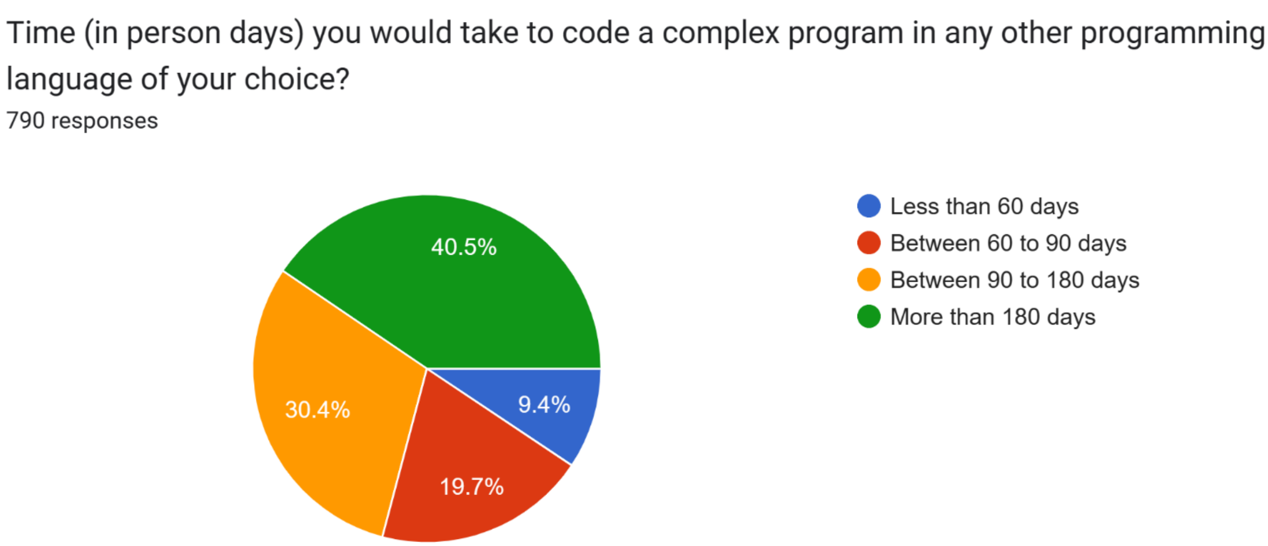


Figure 74
Time (in person days) to code complex program in PL of choice

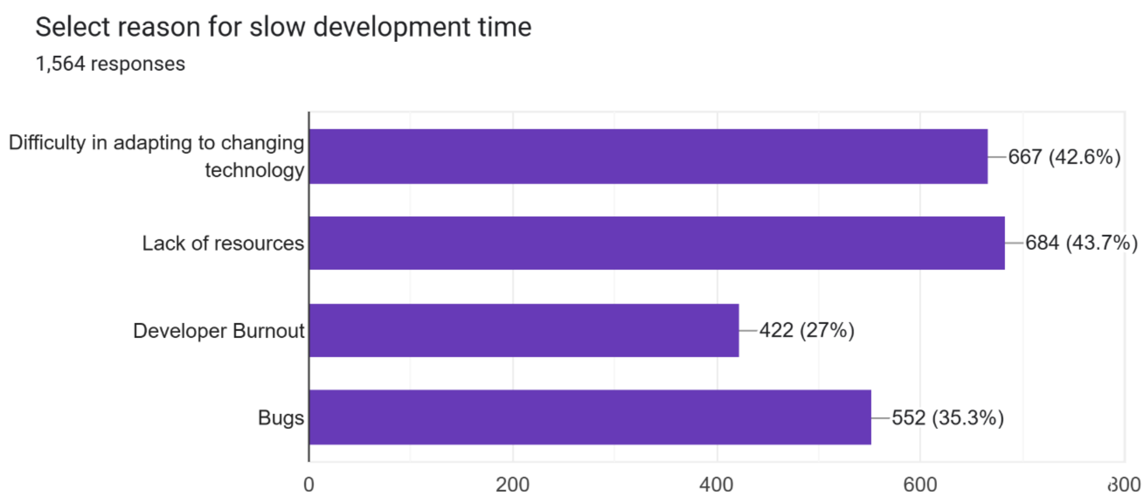


Figure 75
Reason for slow development

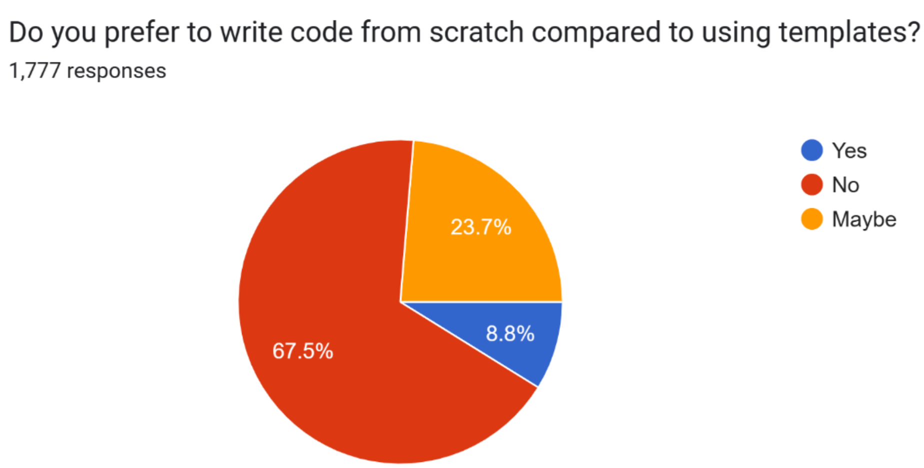


Figure 76
Code writing preference

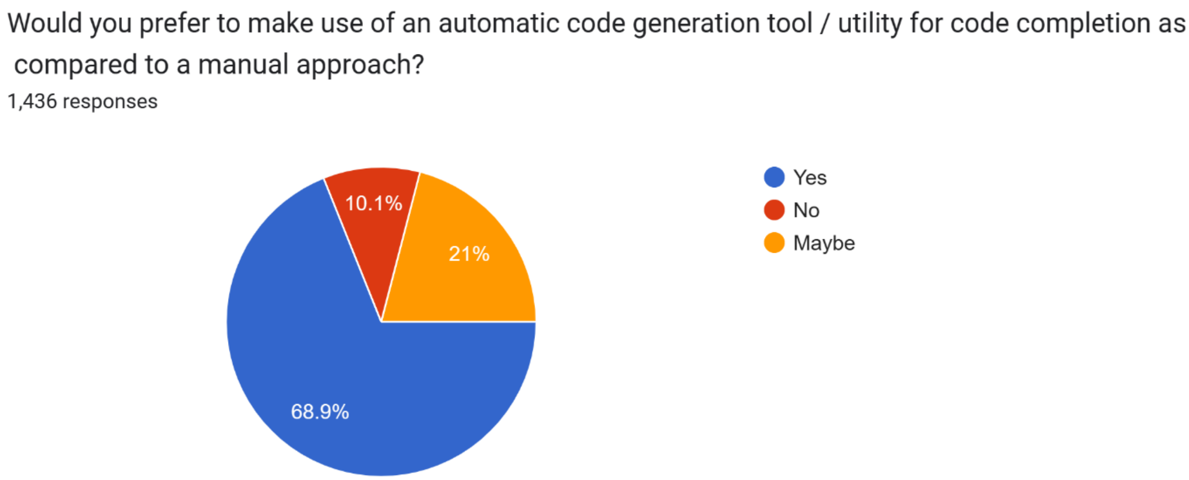


Figure 77
Preference to utility compared to manual approach

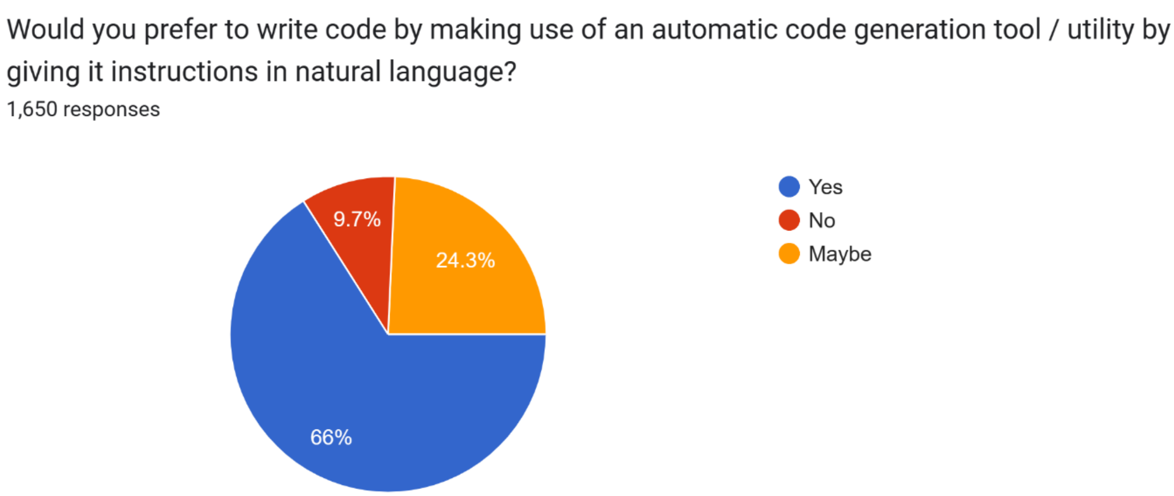


Figure 78
Preference for auto code generation

APPENDIX B

CODE SAMPLES

Table 105

Code generation capability of GPT-J, GPT-3, and Codex

GPT-J generated code	GPT-3 generated code	Codex generated code

import sys
 	
def main():
    name = input("Enter your name: ")
    age = input("Enter your age: ")
    rollno = input("Enter your rollno: ")
    with open('file.txt', 'w') as f:
        f.write(name + "\n" + age + "\n" + rollno + "\n")
	f = open("student.txt", "w")
	while True:
	    print("*" * 100)
	    name = input("Enter student's name: ")
	    f.write("{}\n".format(name))
	    age = input("Enter student's age: ")
	    f.write("{}\n".format(age))
	    rollno = input("Enter student's rollno: ")
	    f.write("{}\n".format(rollno))
	    print("*" * 100)
	    option = input("Do you want to continue? (Y/N): ")
	    if option.upper() == "N":
	        break
	f.close()
		# Taking input from user
		name = input("Enter your name: ")
		age = input("Enter your age: ")
		rollno = input("Enter your rollno: ")
		
		# Writing to file
		with open("student.txt", "w") as f:
		    f.write(f"Name: {name}\n")
		    f.write(f"Age: {age}\n")
		    f.write(f"RollNo: {rollno}\n")

```
"""
Find business logic in Python program
"""

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        dummy = ListNode(0)
        current, carry = dummy, 0

        while l1 or l2:
            val = carry
            if l1:
                val += l1.val
                l1 = l1.next
            if l2:
                val += l2.val
                l2 = l2.next
            carry, val = divmod(val, 10)
            current.next = ListNode(val)
            current = current.next

        if carry == 1:
            current.next = ListNode(1)

        return dummy.next

s = Solution()
l1 = ListNode(4)
l2 = ListNode(2)
print(s.addTwoNumbers(l1,l2).val)

Business Logic:
    1. Add two numbers
    2. Return the sum of two numbers
    3. Return the sum of two numbers as a linked list
```

Figure 79
Code Documentation using Codex

addnums.py - E:/Personal/ssbm_dba/research_design/addnums.py (3.7.8)

File Edit Format Run Options Window Help

```
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        dummy = ListNode(0)
        current, carry = dummy, 0

        while l1 or l2:
            val = carry
            if l1:
                val += l1.val
                l1 = l1.next
            if l2:
                val += l2.val
                l2 = l2.next
            carry, val = divmod(val, 10)
            current.next = ListNode(val)
            current = current.next

        if carry == 1:
            current.next = ListNode(1)

        return dummy.next

s = Solution()
l1 = ListNode(4)
l2 = ListNode(2)
print(s.addTwoNumbers(l1,l2).val)
```

Figure 80
Python Code (To be translated to Java)

```java
class ListNode {
    public int val;
    public ListNode next;
    public ListNode(int x) {
        val = x;
        next = null;
    }
}

public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;
        int carry = 0;
        while (l1 != null || l2 != null) {
            int val = carry;
            if (l1 != null) {
                val += l1.val;
                l1 = l1.next;
            }
            if (l2 != null) {
                val += l2.val;
                l2 = l2.next;
            }
            carry = val / 10;
            val = val % 10;
            current.next = new ListNode(val);
            current = current.next;
        }
        if (carry == 1) {
            current.next = new ListNode(1);
        }
        return dummy.next;
    }
}
```

Figure 81
Java Code (Translated using Codex)

```java
public static void main(String args[])
{
    Solution s = new Solution();
    ListNode l1 = new ListNode(2);
    ListNode l2 = new ListNode(4);
    ListNode r = s.addTwoNumbers(l1, l2);
    System.out.println(r.val);
}
```

Figure 82
Lines of the program added to test the program

```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummyHead = new ListNode(0);
    ListNode p = l1, q = l2, curr = dummyHead;
    int carry = 0;
    while (p != null || q != null) {
        int x = (p != null) ? p.val : 0;
        int y = (q != null) ? q.val : 0;
        int sum = carry + x + y;
        carry = sum / 10;
        curr.next = new ListNode(sum % 10);
        curr = curr.next;
        if (p != null) p = p.next;
        if (q != null) q = q.next;
    }
    if (carry > 0) {
        curr.next = new ListNode(carry);
    }
    return dummyHead.next;
}
```

Figure 83
Code written manually by Java programmer

```
<!DOCTYPE html>
<html>
 <head>
  <title>Eco-Friendly Store</title>
  <link rel="stylesheet" href="style1.css" />
 </head>
 <body>
  <div class="container">
   <header>
    <div class="logo">Logo Here</div>
    <nav>
     <ul>
      <li>Home</li>
      <li>About</li>
      <li>Products</li>
      <li>Contact</li>
     </ul>
    </nav>
    <form>
     <input type="search" placeholder="Search.." />
    </form>
   </header>
   <main>
    <section class="hero">
     <h1>Welcome to Our Store!</h1>
     <div class="featured-products">
      <div class="product-card">
       <img src="product-1.jpg" alt="Featured Product" />
       <div class="product-info">
        <h3>Product Name</h3>
        <p>Description Here</p>
        <h4>Price: $XX.XX</h4>
       </div>
      </div>
      <div class="product-card">
       <img src="product-2.jpg" alt="Featured Product" />
       <div class="product-info">
        <h3>Product Name</h3>
        <p>Description Here</p>
        <h4>Price: $XX.XX</h4>
       </div>
      </div>
      <div class="product-card">
       <img src="product-3.jpg" alt="Featured Product" />
       <div class="product-info">
        <h3>Product Name</h3>
        <p>Description Here</p>
        <h4>Price: $XX.XX</h4>
       </div>
      </div>
     </div>
    </section>
    <section class="category-list">
     <h2>Categories</h2>
     <ul>
      <li>Category 1</li>
      <li>Category 2</li>
      <li>Category 3</li>
      <li>Category 4</li>
      <li>Category 5</li>
     </ul>
    </section>
   </main>
   <footer>
    <ul>
     <li><a href="#">Facebook</a></li>
     <li><a href="#">Twitter</a></li>
     <li><a href="#">Instagram</a></li>
     <li><a href="#">About</a></li>
     <li><a href="#">Contact</a></li>
    </ul>
   </footer>
  </div>
 </body>
</html>
```

Figure 84
GPT-3 generated webpage

```css
CSS:

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  background-color: #f3f3f3;
}

.container {
  width: 90%;
  max-width: 1200px;
  margin: 0 auto;
}

header {
  background-color: white;
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 1rem;
}

.logo {
  font-size: 1.5rem;
  font-weight: bold;
}

nav ul {
  list-style: none;
  display: flex;
}

nav ul li {
  padding: 0 1rem;
}

form {
  position: relative;
}

form input {
  font-size: 1rem;
  width: 200px;
  padding: 0.5rem;
  border: 1px solid #ccc;
  border-radius: 3px;
}

main {
  padding: 1rem;
}

.hero {
  background-color: #00bfa5;
  text-align: center;
  color: white;
  padding: 2rem;
}

.featured-products {
  display: flex;
  justify-content: space-between;
  margin-top: 1rem;
}

.product-card {
  width: 300px;
  margin-bottom: 1rem;
}

.product-info {
  text-align: center;
  font-size: 0.9rem;
  padding: 1rem;
}

.category-list {
  background-color: white;
  padding: 1rem;
}

.category-list h2 {
  font-size: 1.2rem;
  font-weight: bold;
  margin-bottom: 0.5rem;
}

.category-list ul {
  list-style: none;
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}

.category-list ul li {
  padding: 0.5rem;
  background-color: #ccc;
  border-radius: 3px;
}

footer {
  background-color: #00bfa5;
  color: white;
  padding: 1rem;
  text-align: center;
}

footer ul {
  list-style: none;
  display: flex;
  justify-content: center;
}

footer ul li {
  padding: 0 0.5rem;
}
```

Figure 85
GPT-3 generated stylesheet

html Copy code

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Eco-Friendly Online Store</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <header>
      <div class="logo">Eco-Friendly Online Store</div>
      <nav>
        <ul>
          <li><a href="#">Home</a></li>
          <li><a href="#">Products</a></li>
          <li><a href="#">About Us</a></li>
          <li><a href="#">Contact Us</a></li>
        </ul>
      </nav>
      <div class="search-bar">
        <input type="text" placeholder="Search Products...">
        <button type="submit">Search</button>
      </div>
    </header>
    <main>
      <section class="hero">
        <h1>Welcome to Eco-Friendly Online Store</h1>
        <p>Find the best eco-friendly products for your home and lifestyle.</p>
        <a href="#" class="button">Shop Now</a>
      </section>
      <section class="categories">
        <h2>Shop by Category</h2>
        <ul>
          <li>
```

Figure 86
ChatGPT generated webpage-1

```
          <li>
            <a href="#">
              <img src="category1.jpg" alt="Category 1">
              <span>Category 1</span>
            </a>
          </li>
          <li>
            <a href="#">
              <img src="category2.jpg" alt="Category 2">
              <span>Category 2</span>
            </a>
          </li>
          <li>
            <a href="#">
              <img src="category3.jpg" alt="Category 3">
              <span>Category 3</span>
            </a>
          </li>
        </ul>
      </section>
    </main>
    <footer>
      <ul>
        <li><a href="#">About Us</a></li>
        <li><a href="#">Contact Us</a></li>
        <li><a href="#">Privacy Policy</a></li>
        <li><a href="#">Terms of Service</a></li>
      </ul>
      <div class="social-links">
        <a href="#"><i class="fab fa-facebook"></i></a>
        <a href="#"><i class="fab fa-twitter"></i></a>
        <a href="#"><i class="fab fa-instagram"></i></a>
      </div>
    </footer>
  </body>
</html>
```

Figure 87
ChatGPT generated webpage-2

```css
/* General styles */
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

body {
  font-family: Arial, sans-serif;
  font-size: 16px;
  line-height: 1.5;
  background-color: #f1f1f1;
  color: #555;
}

a {
  color: #007f5f;
  text-decoration: none;
}

a:hover {
  text-decoration: underline;
}

ul {
  list-style: none;
}

button {
  border: none;
  background-color: #007f5f;
  color: #fff;
  padding: 10px 20px;
```

Figure 88
ChatGPT generated stylesheet-1

```css
  padding: 10px 20px;
  font-size: 16px;
  cursor: pointer;
}

button:hover {
  background-color: #005f4e;
}

/* Header styles */
header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 20px;
  background-color: #fff;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

.logo {
  font-size: 24px;
  font-weight: bold;
  color: #007f5f;
}

nav ul {
  display: flex;
}

nav li {
  margin-left: 20px;
}
```

Figure 89
ChatGPT generated stylesheet-2

```css
/* Search bar styles */
.search-bar {
  display: flex;
}

.search-bar input[type="text"] {
  width: 300px;
  padding: 10px;
  border: 2px solid #007f5f;
  border-right: none;
  border-radius: 0;
  font-size: 16px;
  font-weight: bold;
}

.search-bar button[type="submit"] {
  border-radius: 0;
}

/* Hero section styles */
.hero {
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 500px;
  background-image: url('hero.jpg');
  background-size: cover;
  background-position: center;
  color: #fff;
}
```

Figure 90
ChatGPT generated stylesheet-3

```css
.hero h1 {
  font-size: 48px;
  font-weight: bold;
  text-align: center;
  margin-bottom: 20px;
  text-shadow: 0 0 10px rgba(0, 0, 0, 0.5);
}

.hero p {
  font-size: 24px;
  text-align: center;
  margin-bottom: 40px;
  text-shadow: 0 0 10px rgba(0, 0, 0, 0.5);
}

.hero .button {
  margin-top: 40px;
  background-color: #007f5f;
  border-radius: 30px;
  padding: 15px 30px;
  font-size: 24px;
}

/* Categories section styles */
.categories {
  padding: 40px;
}
```

Figure 91
ChatGPT generated stylesheet-4

```css
.categories h2 {
  font-size: 36px;
  font-weight: bold;
  margin-bottom: 40px;
  text-align: center;
}

.categories ul {
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
}

.categories li {
  margin: 20px;
  text-align: center;
}

.categories li a {
  display: flex;
  flex-direction: column;
  align-items: center;
  color: #555;
}
```

Figure 92
ChatGPT generated stylesheet-5

Figure 93
Android app code - 1

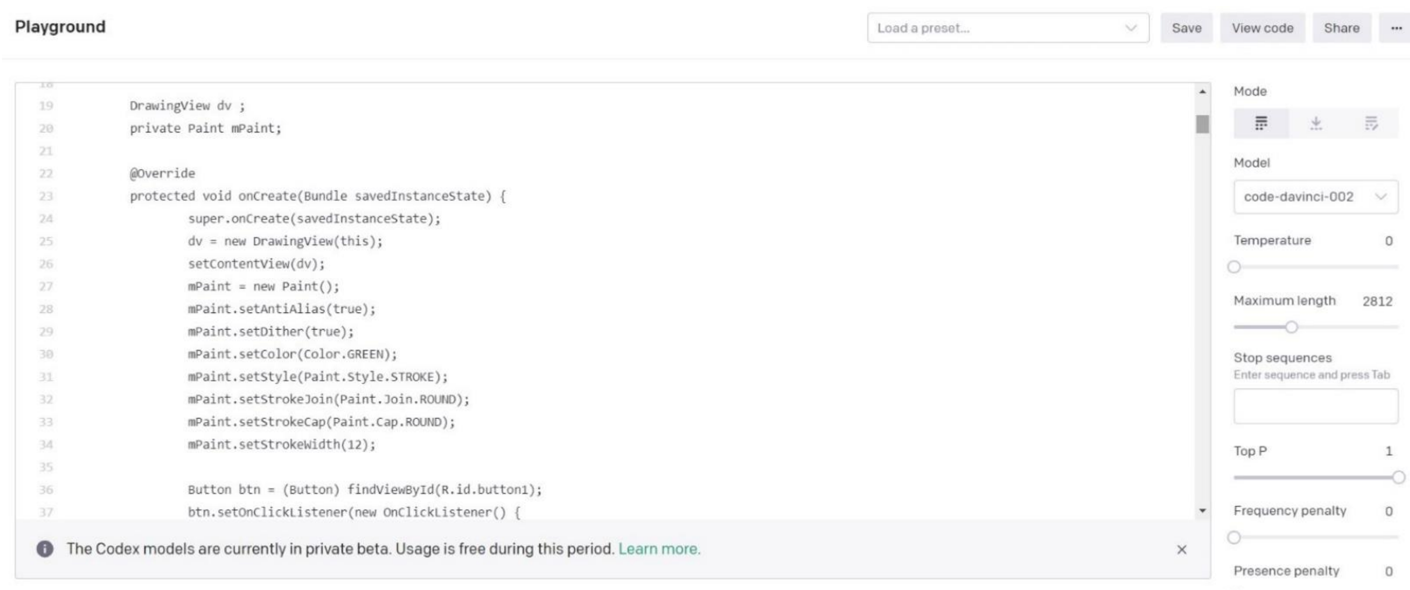


Figure 94
Android app code - 2

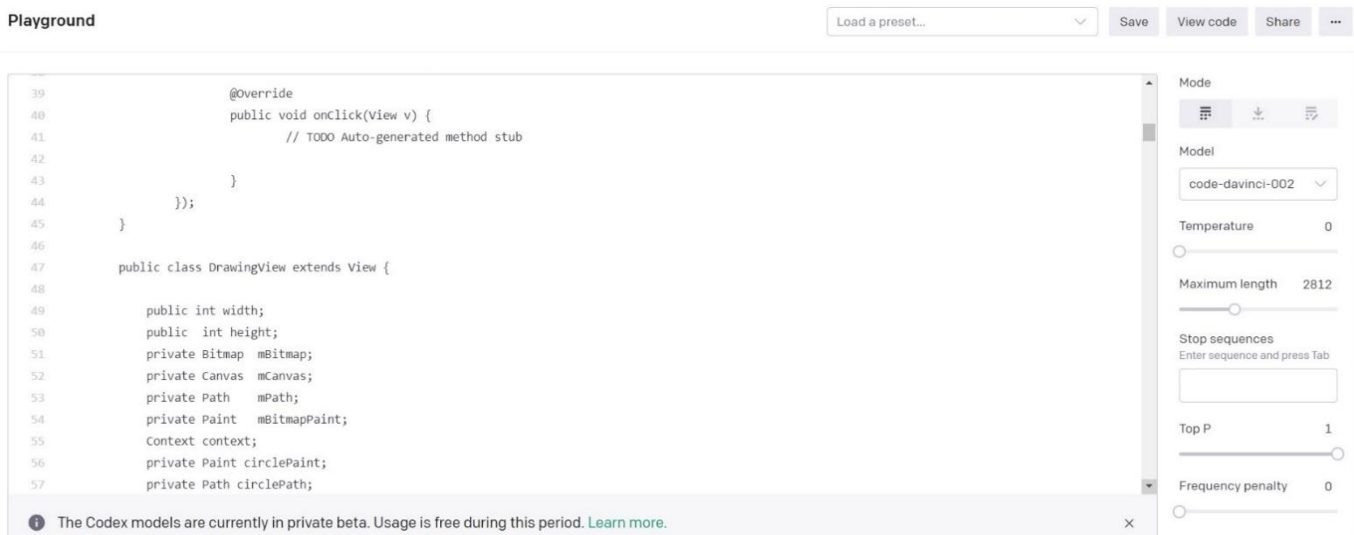


Figure 95
Android app code - 3

Figure 96
Android app code - 4

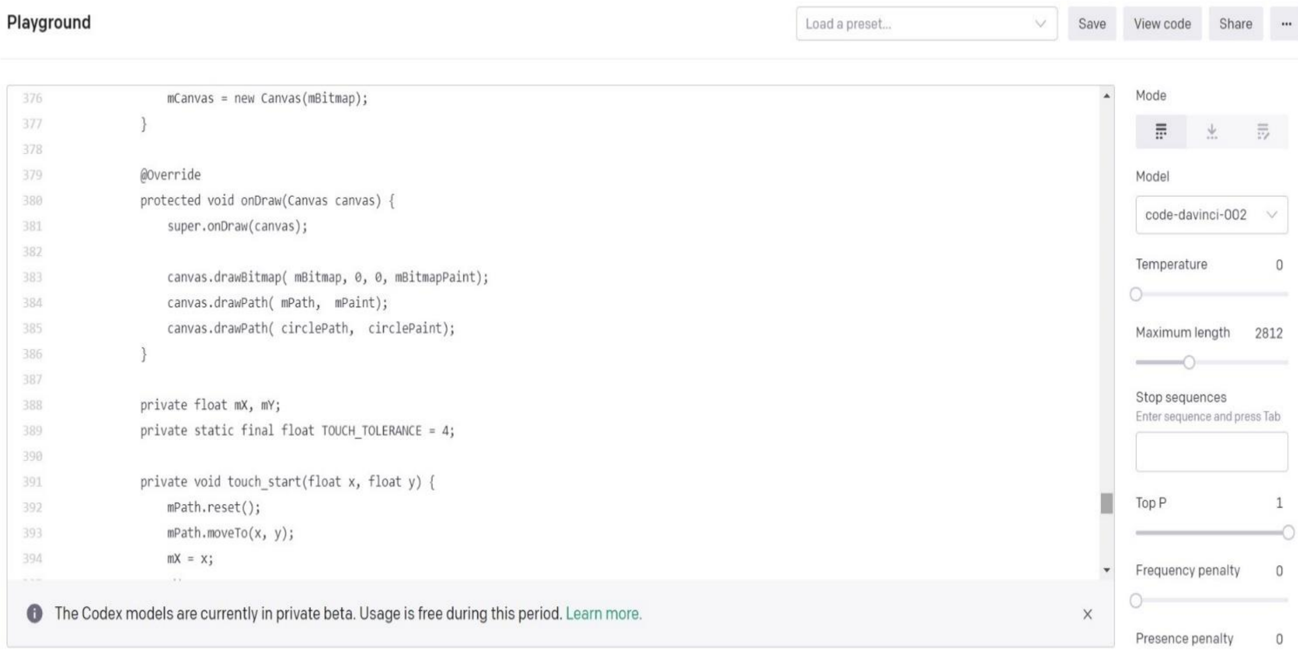


Figure 97
Android app code - 5

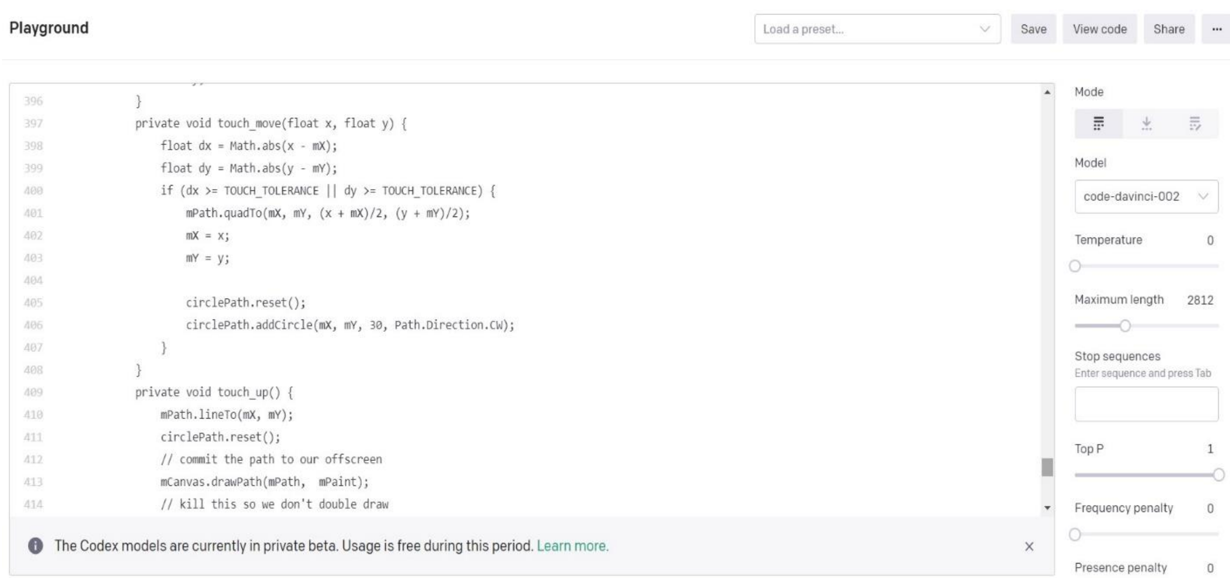


Figure 98
Android app code - 6

Figure 99
Android app code - 7

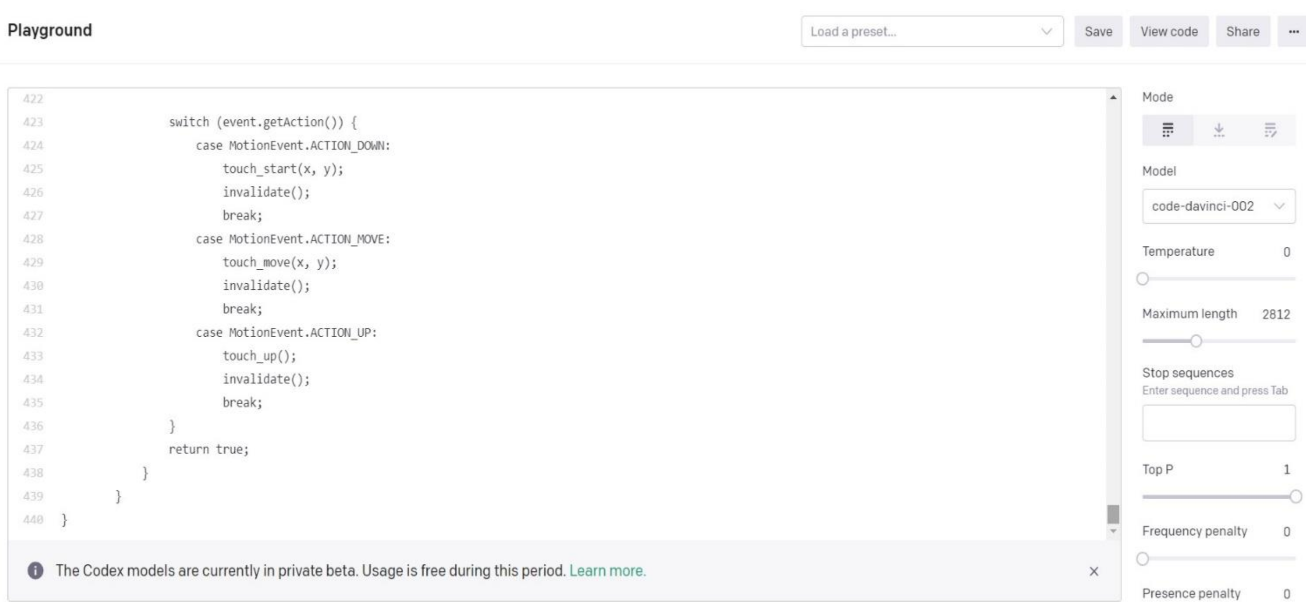


Figure 100
Android app code - 8

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data;
using MySql.Data.MySqlClient;
using System.Data.SqlClient;
using System.Configuration;

namespace WebApplication9
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                BindGrid();
            }
        }

        private void BindGrid()
        {
            string constr = ConfigurationManager.ConnectionStrings["travelConnectionString"].ConnectionString;
            using (MySqlConnection con = new MySqlConnection(constr))
            {
                using (MySqlCommand cmd = new MySqlCommand("SELECT * FROM User"))
                {
                    using (MySqlDataAdapter sda = new MySqlDataAdapter())
                    {
                        cmd.Connection = con;
                        sda.SelectCommand = cmd;
                        using (DataTable dt = new DataTable())
                        {
                            sda.Fill(dt);
                            GridView1.DataSource = dt;
                            GridView1.DataBind();
                            ViewState["CurrentTable"] = dt;
                        }
                    }
                }
            }
        }

        protected void Insert(int id, string name, string email, string phone, string address)
        {
            string constr = ConfigurationManager.ConnectionStrings["travelConnectionString"].ConnectionString;
            using (MySqlConnection con = new MySqlConnection(constr))
            {
                using (MySqlCommand cmd = new MySqlCommand("INSERT INTO User VALUES (@Id, @Name, @Email, @Phone,
                @Address)"))
                {
                    cmd.Parameters.AddWithValue("@Id", id);
                    cmd.Parameters.AddWithValue("@Name", name);
                    cmd.Parameters.AddWithValue("@Email", email);
                    cmd.Parameters.AddWithValue("@Phone", phone);
                    cmd.Parameters.AddWithValue("@Address", address);
                    cmd.Connection = con;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    con.Close();
                }
            }
            this.BindGrid();
        }
```

Figure 101
ASP.NET application -1

```
protected void OnRowEditing(object sender, GridViewEditEventArgs e)
{
    GridView1.EditIndex = e.NewEditIndex;
    this.BindGrid();
}

protected void OnRowUpdating(object sender, GridViewUpdateEventArgs e)
{
    GridViewRow row = GridView1.Rows[e.RowIndex];
    int userId = Convert.ToInt32(GridView1.DataKeys[e.RowIndex].Values[0]);
    string name = (row.FindControl("txtName") as TextBox).Text;
    string email = (row.FindControl("txtEmail") as TextBox).Text;
    string phone = (row.FindControl("txtPhone") as TextBox).Text;
    string address = (row.FindControl("txtAddress") as TextBox).Text;
    string constr = ConfigurationManager.ConnectionStrings["travelConnectionString"].ConnectionString;
    using (MySqlConnection con = new MySqlConnection(constr))
    {
        using (MySqlCommand cmd = new MySqlCommand("UPDATE User SET User_Name = @Name, User_Email =
        @Email, User_Phone = @Phone, User_Address = @Address WHERE User_Id = @UserId"))
        {
            cmd.Parameters.AddWithValue("@UserId", userId);
            cmd.Parameters.AddWithValue("@Name", name);
            cmd.Parameters.AddWithValue("@Email", email);
            cmd.Parameters.AddWithValue("@Phone", phone);
            cmd.Parameters.AddWithValue("@Address", address);
            cmd.Connection = con;
            con.Open();
            cmd.ExecuteNonQuery();
            con.Close();
        }
    }
    GridView1.EditIndex = -1;
    this.BindGrid();
}

protected void OnRowCancelingEdit(object sender, EventArgs e)
{
    GridView1.EditIndex = -1;
    this.BindGrid();
}

protected void OnRowDeleting(object sender, GridViewDeleteEventArgs e)
{
    int userId = Convert.ToInt32(GridView1.DataKeys[e.RowIndex].Values[0]);
    string constr = ConfigurationManager.ConnectionStrings["travelConnectionString"].ConnectionString;
    using (MySqlConnection con = new MySqlConnection(constr))
    {
        using (MySqlCommand cmd = new MySqlCommand("DELETE FROM User WHERE User_Id = @UserId"))
        {
            cmd.Parameters.AddWithValue("@UserId", userId);
            cmd.Connection = con;
            con.Open();
            cmd.ExecuteNonQuery();
            con.Close();
        }
    }
    this.BindGrid();
}

protected void OnRowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow && e.Row.RowIndex != GridView1.EditIndex)
    {
        (e.Row.Cells[5].Controls[0] as LinkButton).Attributes["onclick"] = "return confirm('Do you want
        to delete this row?');";
    }
}
```

Figure 102
ASP.NET application -2

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebForm1.aspx.cs" Inherits="WebApplication9.WebForm1" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<!--    <style type="text/css">
        body
        {
            font-family: "Segoe UI";
            font-size: 14px;
        }
        .auto-style1 {
            height: 31px;
        }
    </style>-->
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.3/css/all.min.css">
    <style type="text/css">
        body {
            background-image: url('bkgrnd_img1.jpg');
            background-size: cover;
            background-color: #f8f8f8;
            font-family: Arial, sans-serif;
            font-size: 14px;
            line-height: 1.5;
        }
        .container {
            max-width: 80%;
            margin: 0 auto;
            padding: 20px;
            background-color: #fff;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.2);
        }
        h1 {
            font-family: "Algerian", Arial, sans-serif;
            font-size: 36px !important;
            text-align: center;
            margin-top: 50px;
            margin-bottom: 20px;
            }
        .card {
            margin-bottom: 20px;
            border: 1px solid #ccc;
            border-radius: 5px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            overflow: hidden;
        }
        .card-header {
            background-color: #f1f1f1;
            padding: 10px;
            font-weight: bold;
            border-bottom: 1px solid #ccc;
        }
        .card-body {
            padding: 10px;
        }
        table {
            width: 100%;
            border-collapse: collapse;
        }
        th, td {
            padding: 10px;
            text-align: left;
            vertical-align: middle;
            border: none;
        }
        th {
            font-weight: bold;
        }
        .btn {
            display: inline-block;
            padding: 10px 20px;
            background-color: #4CAF50;
            color: #fff;
            text-decoration: none;
            border-radius: 5px;
        }
        .btn:hover {
            background-color: #3e8e41;
        }
```

Figure 103
ASP.NET application -3

```
        .btn-danger {
            background-color: #f44336;
        }
        .btn-danger:hover {
            background-color: #da190b;
        }
    </style>
</head>
<body>
    <form id="form1" runat="server">
        <h1>Travel Application</h1>
        <div class="container">
        <div class="card">
            <div class="card-header">
                <i class="fas fa-users"></i> Passenger Details
            </div>
            <div class="card-body">
            <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="false" DataKeyNames="User_Id"
            OnSelectedIndexChanged="GridView1_SelectedIndexChanged"
                OnRowEditing="OnRowEditing"
                OnRowCancelingEdit="OnRowCancelingEdit"
                OnRowUpdating="OnRowUpdating" OnRowDeleting="OnRowDeleting" OnRowDataBound="OnRowDataBound"
                OnRowCommand="GridView1_RowCommand">
                <Columns>
                    <asp:TemplateField HeaderText="Name">
                        <ItemTemplate>
                            <asp:Label ID="lblName" runat="server" Text='<%# Eval("User_Name") %>'></asp:Label>
                        </ItemTemplate>
                        <EditItemTemplate>
                            <asp:TextBox ID="txtName" runat="server" Text='<%# Eval("User_Name") %>'></asp:TextBox>
                        </EditItemTemplate>
                    </asp:TemplateField>
                    <asp:TemplateField HeaderText="Email">
                        <ItemTemplate>
                            <asp:Label ID="lblEmail" runat="server" Text='<%# Eval("User_Email") %>'></asp:Label>
                        </ItemTemplate>
                        <EditItemTemplate>
                            <asp:TextBox ID="txtEmail" runat="server" Text='<%# Eval("User_Email")
                            %>'></asp:TextBox>
                        </EditItemTemplate>
                    </asp:TemplateField>
                    <asp:TemplateField HeaderText="Phone">
                        <ItemTemplate>
                            <asp:Label ID="lblPhone" runat="server" Text='<%# Eval("User_Phone") %>'></asp:Label>
                        </ItemTemplate>
                        <EditItemTemplate>
                            <asp:TextBox ID="txtPhone" runat="server" Text='<%# Eval("User_Phone")
                            %>'></asp:TextBox>
                        </EditItemTemplate>
                    </asp:TemplateField>
                    <asp:TemplateField HeaderText="Address">
                        <ItemTemplate>
                            <asp:Label ID="lblAddress" runat="server" Text='<%# Eval("User_Address")
                            %>'></asp:Label>
                        </ItemTemplate>
                        <EditItemTemplate>
                            <asp:TextBox ID="txtAddress" runat="server" Text='<%# Eval("User_Address")
                            %>'></asp:TextBox>
                        </EditItemTemplate>
                    </asp:TemplateField>
                    <asp:TemplateField HeaderText="Show Account">
                        <ItemTemplate>
                            <asp:Button ID="btnShowAcc" runat="server" Text="Show Accomodation"
                                CommandName="ShowAccomodation" CommandArgument='<%# Eval("user_id") + "," +
                                Eval("User_Name") %>' />
                        </ItemTemplate>
                    </asp:TemplateField>
                    <asp:CommandField ShowEditButton="true" />
                    <asp:CommandField ShowDeleteButton="true" />
                    <asp:CommandField ShowSelectButton="true" />
                </Columns>
            </asp:GridView>
            <asp:Button ID="btnAddNewRow" runat="server" Text="Add New Row" OnClick="AddNewRowToGrid" />
        </div>
        </div>
        </div>
    </form>
</body>
</html>
```

Figure 104
ASP.NET application -4

Figure 105
Tic-Tac-Toe 1

Figure 106
Tic-Tac-Toe 2

Figure 107
Tic-Tac-Toe 3

Figure 108
Tic-Tac-Toe 4

Figure 109
Tic-Tac-Toe 5

Figure 110
Tic-Tac-Toe 6


```
        case 5:
            if (board[1][1]=='-'){
                board[1][1]={'O'};
                clr();
                System.out.println("\t\t\t\t\t\t\t"+board[0][0]+" | "+board[0][1]+" | "+board[0][2]+"\n"
                    +"\t\t\t\t\t\t\t"+"-- --- --- \n"
                    +"\t\t\t\t\t\t\t"+board[1][0]+" | "+board[1][1]+" | "+board[1][2]+"\n"
                    +"\t\t\t\t\t\t\t"+"-- --- --- \n"
                    +"\t\t\t\t\t\t\t"+board[2][0]+" | "+board[2][1]+" | "+board[2][2]+"\n");
                    return true;
            }
            else {
                System.out.print("That position is already occpied, enter a non occupied cell: ");
                goto specialCase;
            }
            break;

        case 6:
            if (board[1][2]=='-'){
                board[1][2]={'O'};
                clr();
                System.out.println("\t\t\t\t\t\t\t"+board[0][0]+" | "+board[0][1]+" | "+board[0][2]+"\n"
```

Figure 111
Tic-Tac-Toe 7


```
                System.out.println("\t\t\t\t\t\t\t"+board[0][0]+" | "+board[0][1]+" | "+board[0][2]+"\n"
                    +"\t\t\t\t\t\t\t"+"-- --- --- \n"
                    +"\t\t\t\t\t\t\t"+board[1][0]+" | "+board[1][1]+" | "+board[1][2]+"\n"
                    +"\t\t\t\t\t\t\t"+"-- --- --- \n"
                    +"\t\t\t\t\t\t\t"+board[2][0]+" | "+board[2][1]+" | "+board[2][2]+"\n");
                    return true;
                }
            else {
                System.out.print("That position is already occpied, enter a non occupied cell: ");
                goto specialCase;
            }
            break;

        case 7:
            if (board[2][0]=='-'){
                board[2][0]={'O'};
                clr();
                System.out.println("\t\t\t\t\t\t\t"+board[0][0]+" | "+board[0][1]+" | "+board[0][2]+"\n"
                    +"\t\t\t\t\t\t\t"+"-- --- --- \n"
                    +"\t\t\t\t\t\t\t"+board[1][0]+" | "+board[1][1]+" | "+board[1][2]+"\n"
                    +"\t\t\t\t\t\t\t"+"-- --- --- \n"
                    +"\t\t\t\t\t\t\t"+board[2][0]+" | "+board[2][1]+" | "+board[2][2]+"\n");
```

Figure 112
Tic-Tac-Toe 8

Figure 113
Tic-Tac-Toe 9

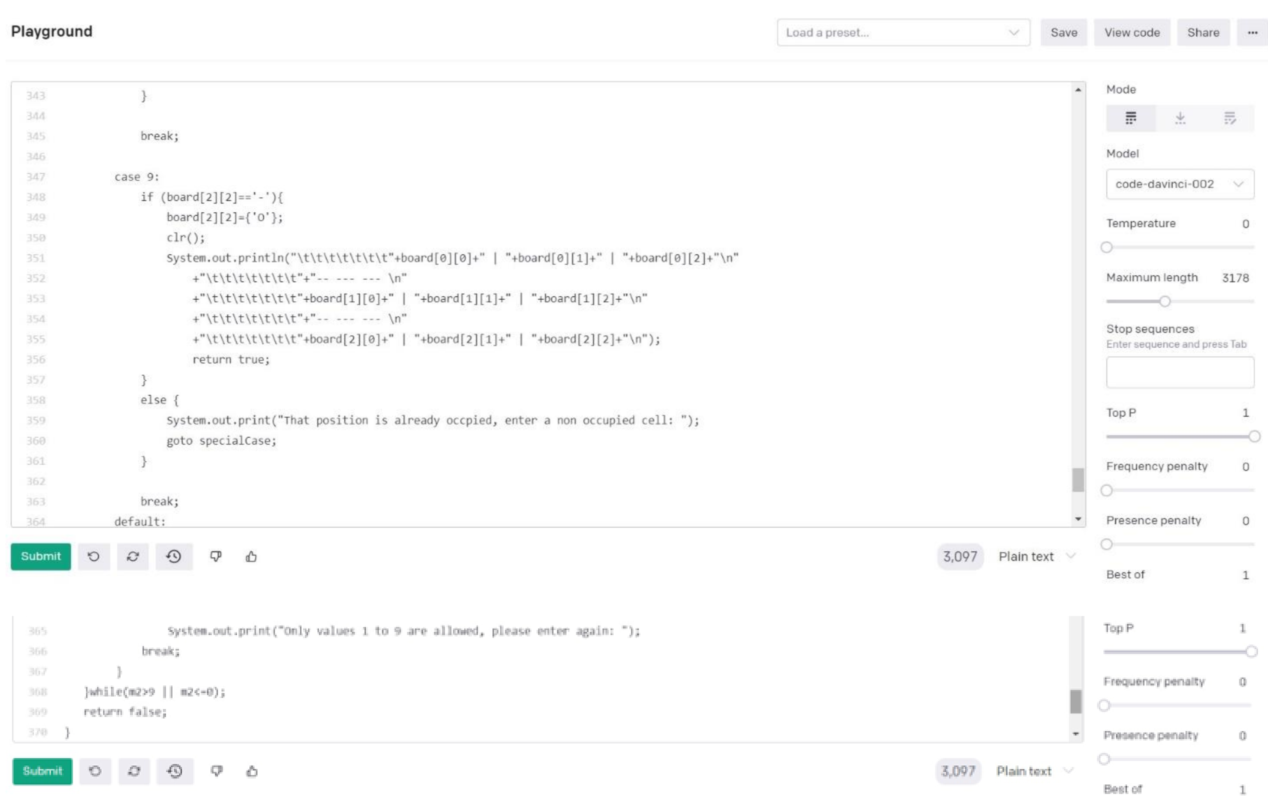


Figure 114
Tic-Tac-Toe 10

```
CONFIRMATION-PARA.
   DISPLAY(1 1) ERASE.
   OPEN INPUT CONFIRMATIONFILE.
   DISPLAY "ENTER CODE :".
   ACCEPT CCONID.
   DISPLAY(1 1) ERASE.
   READ CONFIRMATIONFILE INVALID KEY
        GO TO ERROR-CONFIRMATION-PARA.
   DISPLAY(1 1) " CONFIRMATION CODE :" CCONID.
   DISPLAY(2 1) " EMPLOYEE CODE    :" CEMPID.
   DISPLAY(3 1) " CONFIRMATION DATE :" CCDATE.
   CLOSE CONFIRMATIONFILE.
   DISPLAY(20 10)
    "PRESS ENTER TO RETURN TO HRMS READ MENU".
   STOP ' '.
   GO TO MAIN-PARA.
```

```
GRADE-PARA.
   DISPLAY(1 1) ERASE.
   OPEN INPUT GRADEFILE.
   DISPLAY "ENTER GRADE NO. :".
   ACCEPT GR.
   DISPLAY(1 1) ERASE.
   PERFORM GR-READ-PARA UNTIL FSG = 10.
GR-READ-PARA.
   READ GRADEFILE AT END GO TO GR-EXIT-PARA.
   IF GGRADE = GR
   DISPLAY(1 1) " GRADE NO.   :" GGRADE.
   DISPLAY(2 1) " DESIGNATION :" GDESIGN.
GR-EXIT-PARA.
   CLOSE GRADEFILE.
   DISPLAY ' '.
   DISPLAY ' '.
   DISPLAY "PRESS ENTER TO RETURN TO HRMS READ MENU".
   STOP ' '.
   GO TO MAIN-PARA.
```

```
TRANSFER-PARA.
   DISPLAY(1 1) ERASE.
   OPEN INPUT TRANSFERFILE.
   DISPLAY "ENTER TRANSFER CODE :".
   ACCEPT TTRFID.
   DISPLAY(1 1) ERASE.
   READ TRANSFERFILE INVALID KEY GO TO ERROR-TRANSFER-PARA.
   DISPLAY(1 1) " TRANSFER CODE   :" TTRFID.
   DISPLAY(2 1) " EMP CODE        :" TEMPID.
   DISPLAY(3 1) " OLD BRANCH CODE :" TOBRID.
   DISPLAY(4 1) " TRANSFER DATE   :" TTRFDT.
   CLOSE TRANSFERFILE.
   DISPLAY(20 10)
    "PRESS ENTER TO RETURN TO HRMS READ MENU".
   STOP ' '.
   GO TO MAIN-PARA.

EMPPERSONAL-PARA.
   DISPLAY(1 1) ERASE.
   OPEN INPUT EMPPERSONALFILE.
   DISPLAY "ENTER EMP CODE :".
   ACCEPT EPEMPID.
   DISPLAY(1 1) ERASE.
   READ EMPPERSONALFILE INVALID KEY
          GO TO ERROR-EMPPERSONAL-PARA.
   DISPLAY(1 1) " EMPLOYEE CODE     :" EPEMPID.
   DISPLAY(2 1) " TEMPORARY ADDRESS :" EPTADD.
   DISPLAY(3 1) " PHONE             :" EPTPH.
   DISPLAY(4 1) " DOB               :" EPDOB.
   DISPLAY(5 1) " POB               :" EPPOB.
   DISPLAY(6 1) " LANGUAGE KNOWN    :" EPLANG.
   DISPLAY(7 1) " BLOOD GROUP       :" EPBLOOD.
   DISPLAY(8 1) " WEIGHT            :" EPWEIGHT.
   DISPLAY(9 1) " HEIGHT            :" EPHEIGHT.
   DISPLAY(10 1) " VISION           :" EPVISION.
   DISPLAY(11 1) " FATHER'S NAME    :" EPFATHER.
   DISPLAY(12 1) " DOB OF FATHER    :" EPDOBF.
   DISPLAY(13 1) " MOTHER'S NAME    :" EPMOTHER.
```

Figure 115
COBOL Code

```
  DISPLAY(14 1) " DOB OF MOTHER  :" EPDOBM.
  DISPLAY(15 1) " SPOUSE NAME   :" EPSPOUSE.
  DISPLAY(16 1) " CHILD NAME    :" EPCHILD.
  DISPLAY(17 1) " DOB OF CHILD  :" EPDOBC.
  CLOSE EMPPERSONALFILE.
  DISPLAY(20 10)
   "PRESS ENTER TO RETURN TO HRMS READ MENU".
  STOP ' '.
  GO TO MAIN-PARA.

ERROR-EMP-PARA.
  CLOSE EMPFILE.
  DISPLAY(1 1) ERASE.
  DISPLAY(12 30) "INVALID CODE".
  DISPLAY(20 10)
   "PRESS ENTER TO RETURN TO HRMS READ MENU".
  STOP ' '.
  GO TO MAIN-PARA.

ERROR-LEAVE-PARA.
  CLOSE LEAVEFILE.
  DISPLAY(1 1) ERASE.
  DISPLAY(12 30) "INVALID CODE".
  DISPLAY(20 10)
   "PRESS ENTER TO RETURN TO HRMS READ MENU".
  STOP ' '.
  GO TO MAIN-PARA.

ERROR-BRANCH-PARA.
  CLOSE BRANCHFILE.
  DISPLAY(1 1) ERASE.
  DISPLAY(12 30) "INVALID CODE".
  DISPLAY(20 10)
   "PRESS ENTER TO RETURN TO HRMS READ MENU".
  STOP ' '.
  GO TO MAIN-PARA.

ERROR-DEPARTMENT-PARA.

CLOSE DEPARTMENTFILE.
DISPLAY(1 1) ERASE.
DISPLAY(12 30) "INVALID CODE".
DISPLAY(20 10)
 "PRESS ENTER TO RETURN TO HRMS READ MENU".
STOP ' '.
GO TO MAIN-PARA.
```

Figure 116
COBOL Code Continued

Point	Parameters	Sub-parameters			Score	Avg. Score	Final Value
		Model Discovery					
1	Complexity of Application	Total LOC	< 5000, 5000 - 50000, > 50000	< 5000	1	1	**Simple**
		Individual files - Small / Large	Small, Medium, Large	Small	1		
		Frameworks / Components / DB involved	None, Few, Many	None	1		
		Interdependency between code components	None, Few, Many	None	1		
2	Model Capability	Published scores	Very Low, Low, Average, High	High	3	3.5	**Excellent**
		Initial testing	Poor, Average, Good, Excellent	Excellent	4		
		Setup Consideration					
3	Deployment Effort	Deployment Effort	Depends on model size: Small, medium, Large	NA	NA	NA	**NA**
4	Finetuning Effort	Labeled Data - (in GB) for fine-tuning	Depends on point 2, if model capability is poor, very high fine-tuning effort is required, else it is low	NA	NA	NA	**NA**
		Data preparation effort					
		More iterations, more hyperparameter tuning					
5	Infrastructure Cost / API Cos	Size per GPU	<=20 GB, >20 GB <=40GB, >40GB	NA	NA	NA	**NA**
		Count of GPUs	<2, 2,-8, >8	NA	NA		
		Implementation					
6	Prompt engineering effort	Prompt-engineering effort involved (depends on 1 and 2) - low to high	Low, Average, High, Very High, Opt for Manu	Low	1	1	**Low**
7	Manual correction	Depends on 1 & 2	Low, Medium, High, Very High, Opt for man	Low	1	1	**Low**

Effort Saving	Initial effort saving due to prompt-engineering	>75%	75%
	After Manual correction	Between 41% to 56%	49%
	Deployment effort	No deployment effort	49%
	Fine-tuning effort	Inference only API-based model	49%
Cost (no-finetuning)	Cost of doing prompt-engineering as percentage of original cost	<20%	20%
	Cost saving	~80%	80%
	Additional cost increase due to manual corrections	~20%	20%
	Final cost saving		60%

Figure 117
Seven-point Framework Case 1

Point	Parameters	Sub-parameters			Score	Avg. Score	Final Value
		Model Discovery					
1	Complexity of Application	Total LOC	< 5000, 5000 - 50000, > 50000	Between 500	2	1.5	**Medium**
		Individual files - Small / Large	Small, Medium, Large	Small	1		
		Frameworks / Components / DB involved	None, Few, Many	Few	2		
		Interdependency between code components	None, Few, Many	None	1		
2	Model Capability	Published scores	Very Low, Low, Average, High	Average	2	3	**Good**
		Initial testing	Poor, Average, Good, Excellent	Excellent	4		
		Setup Consideration					
3	Deployment Effort	Deployment Effort	Depends on model size: Small, medium, Large	Small model	1	1	**Low**
4	Finetuning Effort	Labeled Data - (in GB) for fine-tuning	Depends on point 2, if model capability is	1	1	1	**Low**
		Data preparation effort	poor, very high fine-tuning effort is				
		More iterations, more hyperparameter tuning	required, else it is low				
5	Infrastructure Cost / API Cos	Size per GPU	<=20 GB, >20 GB <=40GB, >40GB	>20 GB and <	2	1.5	**Medium**
		Count of GPUs	<2, 2,-8, >8	<2	1		
		Implementation					
6	Prompt engineering effort	Prompt-engineering effort involved (depends on 1 and 2) - low to high	Low, Average, High, Very High, Opt for Manu	Average	2	2	**Average**
7	Manual correction	Depends on 1 & 2	Low, Medium, High, Very High, Opt for man	High	3	3	**High**

Effort Saving	Initial effort saving due to prompt-engineering	Between 50% to 75%	63%
	After Manual correction	Between 5% to 23%	14%
	Deployment effort	Further reduction in effort by 10%	13%
	Fine-tuning effort	Further reduction in effort by 50%	6%

Cost (with finetuning)	Infrastructure cost	20%	
	Deployment effort cost and finetuning cost	10%	
	Cost of inferencing and manual corrections	40%	
	Final cost saving	30%	

Figure 118
Seven-point Framework Case 2

REFERENCES

Abad, Z. S. H. *et al.* (2017) *What Works Better? A Study of Classifying Requirements*. 2017 IEEE 25th International Requirements Engineering Conference (RE), Lisbon, Portugal. doi: 10.1109/RE.2017.36.

Aggarwal, G. (2021) *Impact of AI on software development - implications of ICC, The Architect Coach*. Available at: https://thearchitectcoach.com/impact-of-ai-on-software-development-implications-of-icc/.

AI21labs (2021) *Announcing AI21 Studio and Jurassic-1 Language Models, Ai21.com*. Available at: https://www.ai21.com/blog/announcing-ai21-studio-and-jurassic-1. (Accessed: November 16, 2021).

Alford, A. (2022) *Meta open-sources 175 billion parameter AI language model OPT, InfoQ*. Available at: https://www.infoq.com/news/2022/06/meta-opt-175b/ (Accessed: April 9, 2023).

Alkashri, Z., Siyam, N. and Alqaryouti, O. (2020) "A detailed survey of Artificial Intelligence and Software Engineering: Emergent Issues," in *2020 Fourth International Conference on Inventive Systems and Control (ICISC)*. IEEE, pp. 666–672.

Asiroglu, B. *et al.* (2019) "Automatic HTML code generation from mock-up images using machine learning techniques," in *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT)*. IEEE, pp. 1–4.

Bahdanau, D., Cho, K. and Bengio, Y. (2014) "Neural machine translation by jointly learning to align and translate," *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/1409.0473.

Benbya, H. and McKelvey, B. (2006) "Toward a complexity theory of information systems development," *Information Technology and People*, 19(1), pp. 12–34.

Bendor-Samuel, P. (2022) *Spiking Attrition Impact On IT And Engineering Services, Forbes*. Available at: https://www.forbes.com/sites/peterbendorsamuel/2022/05/31/spiking-attrition-impact-on-it-and-engineering-services/?sh=1fe9e7217c99 (Accessed: August 20, 2022).

Bessen, J. and Frick, W. (2018) *How Software Is Helping Big Companies Dominate*, *Harvard Business Review*. Available at: https://hbr.org/2018/11/how-software-is-helping-big-companies-dominate.

BigScience (2022) *BLOOM*, *Huggingface.co*. Available at: https://bigscience.huggingface.co/blog/bloom (Accessed: April 9, 2023).

Black, D. (2020) *AI is eating software development: Automation for overburdened programmers*, *EnterpriseAI*. Available at: https://www.enterpriseai.news/2020/02/25/how-ai-is-automating-and-improving-software-programming (Accessed: November 7, 2021).

Breaux, T. and Moritz, J. (2021) "The 2021 software developer shortage is coming," *Communications of the ACM*, 64(7), pp. 39–41.

Brooks, F., Jr (1987) "No silver bullet essence and accidents of software engineering," *IEEE Computer*, 20(4), pp. 10–19. doi: 10.1109/mc.1987.1663532.

Brown, T. B. *et al.* (2020) "Language Models are Few-Shot Learners," *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/2005.14165.

Bundy, A. *et al.* (eds.) (2012) "AI meets Formal Software Development (Dagstuhl Seminar 12271)." Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik.

Casamayor, A., Godoy, D. and Campo, M. (2010) "Identification of non-functional requirements in textual specifications: A semi-supervised learning approach," *Information and software technology*, 52(4), pp. 436–445. doi: 10.1016/j.infsof.2009.10.010.

CHAOS REPORT 2015 (no date) *Standishgroup.com*. Available at: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.

Chen, M. *et al.* (2021) "Evaluating large language models trained on code," *arXiv [cs.LG]*. Available at: http://arxiv.org/abs/2107.03374.

Chowdhery, A. *et al.* (2022) "PaLM: Scaling language modeling with Pathways," *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/2204.02311 (Accessed: April 10, 2023).

Chui, M., Manyika, J. and Miremadi, M. (2016) *Where machines could replace humans---and where they can't (yet)*, *Mckinsey.com*. McKinsey & Company. Available at: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/where-machines-could-replace-humans-and-where-they-cant-yet (Accessed: December 18, 2021).

Ciniselli, M. *et al.* (2021) “An empirical study on the usage of BERT models for code completion.” Available at: https://arxiv.org/abs/2103.07115.

Code Translation - Papers with Code (no date). Available at: https://paperswithcode.com/task/code-translation.

“Code-T5 Dataset” (no date). Available at: https://console.cloud.google.com/storage/browser/sfr-codet5-data-research/data/translate;tab=objects?pageState=(%22StorageObjectListTable%22:(%22f%22:%22%255B%255D%22))&prefix=&forceOnObjectsSortingFiltering=false&pli=1.

Code.org 2016 Annual Report (no date) *Code.org*. Available at: https://code.org/about/2016 (Accessed: April 12, 2023).

“CodeXGLUE” (2020) *CodeXGLUE - GitHub*. Available at: https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans.

Cullum, S. (2007) *The effect of automatic code generation on developer job satisfaction, Open.ac.uk*. Available at: http://computing-reports.open.ac.uk/2007/TR2007-19.pdf.

Dam, H. K. (2019) “Artificial intelligence for software engineering,” *XRDS Crossroads The ACM Magazine for Students*, 25(3), pp. 34–37. doi: 10.1145/3313117.

Dias Canedo, E. and Cordeiro Mendes, B. (2020) “Software Requirements classification using machine learning algorithms,” Entropy (Basel, Switzerland), 22(9), p. 1057. doi: 10.3390/e22091057.

EleutherAI - text generation testing UI (no date) *Eleuther.ai*. Available at: https://6b.eleuther.ai/ (Accessed: December 18, 2021).

enterpriseitworld (2018) *Artificial Intelligence- Fruits of A long Journey, Enterprise IT World*. Available at: https://www.enterpriseitworld.com/artificial-intelligence-fruits-of-a-long-journey/ (Accessed: November 3, 2022).

Fabrício Filho, J., Rodriguez, L. G. A. and da Silva, A. F. (2018) “Yet Another Intelligent Code-Generating System: A Flexible and Low-Cost Solution. Journal of Computer Science and Technology,” *Journal of Computer Science and Technology*, 33(5), pp. 940–965.

Feng, Z. *et al.* (2020) “CodeBERT: A pre-trained model for programming and natural languages.” Available at: http://arxiv.org/abs/2002.08155.

Fong, V. L. (2019) *Software requirements classification using word embeddings and convolutional neural networks*. California Polytechnic State University, San Luis Obispo. doi: 10.15368/theses.2018.89.

Galanis, N.-I. *et al.* (2021) "Machine Learning Meets Natural Language Processing -- The story so far." Available at: https://www.researchgate.net/publication/351046768_Machine_Learning_Meets_Natural_Language_Processing_--_The_story_so_far.

Gamma, E. *et al.* (1994) *Design patterns: Elements of reusable object-oriented software*. Boston: Addison-Wesley Professional.

Gandhi, D. (2020) "Using abstarct syntax tree with CodeBERT to improve BLUE score in zero-shot code documentation generation," *Pace.edu*. Available at: https://csis.pace.edu/~scha/MID2020/Abstract/a19.pdf.

GPT-NeoX (no date) *Huggingface.co*. Available at: https://huggingface.co/docs/transformers/model_doc/gpt_neox (Accessed: November 12, 2022).

Grano, G. *et al.* (2017) "app_reviews." Available at: https://huggingface.co/datasets/app_reviews.

Guelfi, N. (2018) "Please ... . draw me a Software Engineer," in *Proceedings of the 51st Hawaii International Conference on System Sciences*. Hawaii International Conference on System Sciences. doi: 10.24251/hicss.2018.701.

Hammond, J. (2020) *Predictions 2021: Software developers face mounting pressure*, *Forrester*. Available at: https://www.forrester.com/blogs/predictions-2021-software-developers-face-mounting-pressure/.

Han, K. *et al.* (2020) "A Survey on Vision Transformer," *arXiv [cs.CV]*. Available at: http://arxiv.org/abs/2012.12556.

Harsh (2022) *TicTacToe*, *GitHub*. Available at: https://github.com/ash-dodek/TicTacToe.

Harvey Nash Group (2021) *Growth in global tech threatened as skills crisis reaches all-time high, finds world's largest digital leadership survey*, *www.harveynash.com*. Available at: https://www.harveynash.com/latest-news/growth-in-global-tech-threatened-as-skills-crisis-reaches-all-time-high-finds-worlds-largest-digital-leadership-survey.

Hung, D. (2021) *Navigate Your Codebase Faster with Find Related Code - Code Faster with Kite*, *kite.com*. Available at: https://www.kite.com/blog/product/navigate-your-codebase-faster-with-find-related-code/ (Accessed: November 7, 2021).

Huston, I. (2018) *AI Is Not the End of Software Developers*, *Vmware.com*. Available at: https://tanzu.vmware.com/content/built-to-adapt/ai-is-not-the-end-of-software-developers (Accessed: September 23, 2021).

Imam, A. T., Rousan, T. and Aljawarneh, S. (2014) "An expert code generator using rule-based and frames knowledge representation techniques," in *2014 5th International Conference on Information and Communication Systems (ICICS)*. IEEE, pp. 1–6.

Introducing ChatGPT (2022) *openai.com*. Available at: https://openai.com/blog/chatgpt (Accessed: December 9, 2022).

kamyu (2020) *LeetCode - Solutions*, *GitHub*. Available at: https://github.com/kamyu104/LeetCode-Solutions.

Karmakar, A. and Robbes, R. (2021) "What do pre-trained code models know about code?," *arXiv [cs.SE]*. Available at: http://arxiv.org/abs/2108.11308.

Kaur, P. (ed.) (no date) "Unit 1: Introduction and Overview," in *Artificial Intelligence | DCAP506*. Lovely Professional University, p. 9. Available at: https://eslm.lpude.in/computer_application/mca/term_4/DCAP506_ARTIFICIAL_INTELLIGENCE/files/basic-html/page9.html (Accessed: November 9, 2021).

Kazman, R. (2017) "Software Engineering," *Computer*, 50(7), pp. 10–11. doi: 10.1109/mc.2017.184.

Kejeiri, M. L. (2020) *Most commonly used Java Design pattern*, *GitHub*. Available at: https://github.com/mkejeiri/Java-Design-Pattern.

Korzeniowski, Ł. and Goczyla, K. (2019) "Artificial intelligence for software development --- the present and the challenges for the future," *Bulletin of the Military University of Technology*, 68(1), pp. 15–32. doi: 10.5604/01.3001.0013.1464.

Krasner, H. (2018) *The cost of poor software quality in the US: A 2020 report*, *Consortium for IT Software Quality*. Available at: https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report.htm.

Kulkarni, R. H. and Padmanabham, P. (2017) “Integration of artificial intelligence activities in software development processes and measuring effectiveness of integration,” *IET software*, 11(1), pp. 18–26. doi: 10.1049/iet-sen.2016.0095.

Kurtanovic, Z. and Maalej, W. (2017) “Automatically classifying functional and non-functional requirements using supervised machine learning,” in *2017 IEEE 25th International Requirements Engineering Conference (RE)*. IEEE, pp. 490–495.

LeClair, A. and McMillan, C. (no date) *Funcom: Java Function Parallel Corpus*, *FunCom*. Available at: http://leclair.tech/data/funcom/.

Li, Z. *et al.* (2020) “The metric for automatic code generation,” *Procedia computer science*, 166, pp. 279–286. doi: 10.1016/j.procs.2020.02.099.

Lins, S. *et al.* (2021) “Artificial intelligence as a service: Classification and research directions,” *Business & information systems engineering*, 63(4), pp. 441–456. doi: 10.1007/s12599-021-00708-w.

Lu, S. *et al.* (2021) “CodeXGLUE: A machine learning benchmark dataset for code understanding and generation,” *arXiv [cs.SE]*. Available at: http://arxiv.org/abs/2102.04664.

Lyon, A. (2021) *The Impact GitHub Copilot Will Have on Software Development*, *GitHub Copilot Review: Is Copilot a Threat? | Rightpoint*. Available at: https://www.rightpoint.com/thought/articles/2021/08/25/the-impact-github-copilot-will-have-on-software-development.

Mashhadi, E. and Hemmati, H. (2021) “Applying CodeBERT for automated program repair of java simple bugs,” *arXiv [cs.SE]*. Available at: http://arxiv.org/abs/2103.11626 (Accessed: May 21, 2022).

Mithas, S., Kude, T. and Whitaker, J. (2018) “Artificial intelligence and IT professionals,” *IT professional*, 20(5), pp. 6–13. doi: 10.1109/mitp.2018.053891331.

Nagulapati, V., Rapelli, S. and Fiaidhi, J. (2020) “Automating Software Development using Artificial Intelligence.” doi: 10.36227/techrxiv.12089139.

Narasimhan, A., Venkatesha Rao, K. P. A. and M B, V. (2021) “CGEMs: A Metric Model for Automatic Code Generation using GPT-3.” Available at: https://www.researchgate.net/publication/354087810_CGEMs_A_Metric_Model_for_Automatic_Code_Generation_using_GPT-3.

National Research Council of Italy. (no date) “Natural Language Requirements Dataset.” Available at: http://fmt.isti.cnr.it/nlreqdataset/.

Navarro-Almanza, R., Juarez-Ramirez, R. and Licea, G. (2017) “Towards supporting software engineering using deep learning: A case of software requirements classification,” in *2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, pp. 116–120.

Núñez, P. V. M. (no date) Software Requirements Classification Using Machine Learning Algorithms | IV Convención Científica Internacional UCLV 2023, IV International Scientific Convention UCLV 2023. Available at: https://convencion.uclv.cu/event/simposio-transferencia-de-conocimiento-en-tecnologias-de-la-informacion-113/track/software-requirements-classification-using-machine-learning-algorithms-4345.

OpenAI (2022) *OpenAI API, platform.openai.com*. Available at: https://platform.openai.com/docs/models/gpt-3 (Accessed: December 21, 2022).

OpenAI (2023) *GPT-4, openai.com*. Available at: https://openai.com/research/gpt-4 (Accessed: April 9, 2023).

Ott, D. (2013) “Automatic requirement categorization of large natural language specifications at Mercedes-Benz for review improvements,” in *Requirements Engineering: Foundation for Software Quality*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 50–64. doi: 10.1007/978-3-642-37422-7_4.

Paik, I. and Wang, J.-W. (2021) “Improving text-to-code generation with features of code graph on GPT-2,” *Electronics*, 10(21), p. 2706. doi: 10.3390/electronics10212706.

Pan, C., Minyan, L. and Biao, X. (2021) “An Empirical Study on Software Defect Prediction Using CodeBERT Model,” *Applied Sciences*, 11(11), p. 4793. doi: 10.3390/app11114793.

Parvez, M. R. *et al.* (2021) “Retrieval augmented code generation and summarization,” *arXiv [cs.SE]*. Available at: http://arxiv.org/abs/2108.11601.

Peng, X., Xing, Z. and Sun, J. (2019) “AI-boosted software automation: learning from human pair programmers,” *Science China Information Sciences*, 62(10). doi: 10.1007/s11432-018-9854-3.

Perez, L., Ottens, L. and Viswanathan, S. (2021) “Automatic Code Generation using Pre-Trained Language Models.” Available at:

https://www.researchgate.net/publication/349521999_Automatic_Code_Generation_using_Pre-Trained_Language_Models.

Piotrowski, P. T. (2012) "Hired personnel and software development costs." Available at: https://www.researchgate.net/publication/238400303_Hired_personnel_and_software_development_costs.

Project - Human Resource Management System. - COBOL Projects Source Code in COBOL (no date) *Sourcecodesworld.com*. Available at: https://www.sourcecodesworld.com/source/show.asp?ScriptID=469 (Accessed: April 14, 2023).

Project Success, Quick reference card (2021). Available at: https://hennyportman.files.wordpress.com/2021/01/project-success-qrc-standish-group-chaos-report-2020.pdf.

PROMISE Software Engineering Repository (no date) *Uottawa.ca*. Available at: http://promise.site.uottawa.ca/SERepository/ (Accessed: December 28, 2022).

Qiu, X. *et al.* (2020) "Pre-trained models for natural language processing: A survey," *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/2003.08271.

Ramel, D. (2020) *IntelliCode advances with first deep learning model for python in VS code*, *Visual Studio Magazine*. Available at: https://visualstudiomagazine.com/articles/2020/08/24/intellicode-deep-learning.aspx.

Schatsky, D. and Bumb, S. (2020) *AI is helping to make better software, AI-assisted software development | Deloitte Insights*. Available at: https://www2.deloitte.com/us/en/insights/focus/signals-for-strategists/ai-assisted-software-development.html. (Accessed: November 7, 2021).

Simplified Wrapper and Interface Generator (no date). Available at: https://www.swig.org/.

Stillman, J. (2017) *21 future jobs the robots are actually creating*, *Inc*. Available at: https://www.inc.com/jessica-stillman/21-future-jobs-robots-are-actually-creating.html (Accessed: November 7, 2021).

Svyatkovskiy, A. *et al.* (2020) "IntelliCode compose: code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, pp. 1433–1443.

Synced (2019) *Deep TabNine: A powerful AI code autocompleter for developers*, *SyncedReview*. Available at: https://medium.com/syncedreview/deep-tabnine-a-powerful-ai-code-autocompleter-for-developers-70454a5953fe.

The curious case of the CHAOS Report 2009 (no date) *Project Smart*. Available at: https://www.projectsmart.co.uk/it-project-management/the-curious-case-of-the-chaos-report-2009.php (Accessed: April 12, 2023).

Turing, A. (2004) "Computing Machinery and Intelligence (1950)," in *The Essential Turing*. Oxford University Press. doi: 10.1093/oso/9780198250791.003.0017.

Vaswani, A. *et al.* (2017) "Attention is all you need," *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/1706.03762.

Venkatesh, G. (2017) "Automation to kill 70% of IT jobs," *The Hindu Business Line*, 14 November. Available at: https://www.thehindubusinessline.com/info-tech/automation-to-kill-70-of-it-jobs/article9960555.ece.

Vinugayathri (no date) *How will artificial intelligence impact software development?*, *Clariontech.com*. Available at: https://www.clariontech.com/blog/how-will-artificial-intelligence-impact-software-development (Accessed: December 12, 2021).

Wang, D. *et al.* (2021) "How much automation does a data scientist want?," *arXiv [cs.LG]*. Available at: http://arxiv.org/abs/2101.03970 (Accessed: May 21, 2023).

Wang, Y. *et al.* (2021) "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/2109.00859.

Wilson, D. (no date) *IBM Garage Methodology*. Available at: https://www.ibm.com/garage/method/practices/code/tool_lint/.

Wilson, H. J. and Daugherty, P. R. (2018) "Collaborative intelligence: humans and AI are joining forces." Harvard Business Review, 96(4), pp. 114–123.

Xu, F. F. et al. (2022) "A systematic evaluation of large language models of code," in Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. New York, NY, USA: ACM.

Xu, F. F., Vasilescu, B. and Neubig, G. (2021) "In-IDE code generation from natural language: Promise and challenges," *arXiv [cs.SE]*. Available at: http://arxiv.org/abs/2101.11149.

Yang, S., Wang, Y. and Chu, X. (2020) "A survey of deep learning techniques for Neural Machine Translation," *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/2002.07526.

Your AI pair programmer (no date) *GitHub Copilot · Your AI pair programmer*. Available at: https://copilot.github.com/.

Zaremba, W., Brockman, G. and OpenAI (2021) *OpenAI Codex, OpenAI*. Available at: https://openai.com/blog/openai-codex/.

Zargar, S. A. (2021) "Introduction to Sequence Learning Models: RNN, LSTM, GRU." doi: 10.13140/RG.2.2.36370.99522.

Zheng, Q. *et al.* (2023) "CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X," *arXiv [cs.LG]*. Available at: http://arxiv.org/abs/2303.17568.

Zhou, W. *et al.* (2021) "Improving code autocompletion with transfer learning," *arXiv [cs.SE]*. Available at: http://arxiv.org/abs/2105.05991.

Zohair, L. M. A. (2018) "The future of software engineering by 2050s: Will AI replace software engineers?," *International Journal of Information Technology and Language Studies*, 2(3). Available at: https://journals.sfu.ca/ijitls/index.php/ijitls/article/view/23.