ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

COMPUTER SCIENCE DEPARTMENT

BACHELORS THESIS

«UTILIZATION OF LARGE LANGUAGE MODELS FOR THE DEVELOPMENT OF AUTOMATED SOFTWARE ACCEPTANCE TESTING»

PLATIAS KONSTANTINOS

UNIVERSITY ID: P3200157

SUPERVISOR OF THE THESIS:

DIAMANTIDIS NIKOLAOS

ATHENS, AUGUST 2024

ABSTRACT

In modern software development, large language models play a crucial role in automating code and managing requirements. These models, which are based on artificial intelligence techniques, can understand and generate human language with high accuracy, thus improving software quality.

Behavior-Driven Development (BDD) is a methodology that focuses on collaboration between developers and end users to understand requirements by creating scenarios in natural language. Cucumber is a tool that supports BDD by writing and automating these scenarios.

This thesis wishes to objectively evaluate large language models in the generation of automated code through BDD. The study assesses the GPT-3.5, GPT-4, GPT-4o, and GitHub Copilot models, focusing on the creation of Step Definitions and understanding variables. The methodology includes four phases of experiments, each involving different amounts of prior knowledge and different techniques for presenting information to the model, with the goal of finding the optimal way to communicate with it.

The findings show that GPT-4o excels in producing Step Definitions and understanding requirements, with GitHub Copilot following to a lesser extent. Strategies such as presenting BDD scenarios in a single prompt and instructing the model to first generate code for Domain classes/Data Access Objects/Services proved to be very effective, while adding information about the Domain classes improved the quality of the results.

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ABSTRACT **1**
LIST OF FIGURES **6**
LIST OF TABLES **7**
1.INTRODUCTION **8**
1.1 Large Language Models, BDD, and Code Automation 8
1.2 Purpose of the Present Thesis 8
1.3 Structure 9
2.Large Language Models **10**
2.1 What is a Neural Network and How Does It Work 10
2.1.1 Definition of Neural Networks 10
2.1.2 Definition of Gradient Descent 11
2.1.3 Definition of the Backpropagation Algorithm 13
2.2 What is a Large Language Model 14
2.2.1 Definition of a Large Language 14
2.2.2 What a Transformer Does 14
2.2.3 What is Attention Layer 15
2.2.4 What is the Feed Forward Step 16
2.3 History and Evolution of Large Language Models 16
2.3.1 The ELIZA Chatbot 16
2.3.2 The Rise of Neural Networks 17
2.3.3 Creation of LSTMs 18
2.3.4 Creation of Gated Recurrent Network 18
2.3.5 The Rise of the Attention Mechanism 18
2.3.6 The Invention of Transformers 19
2.3.7 Emergence of Large Language Models 19
2.4 Main Applications and Uses 20
2.4.1 Audio Analysis 20
2.4.2 Content Creation 20
2.4.3 Customer Support 20
2.4.4 Language Translation 20
2.4.5 Education 21
2.4.6 Cybersecirity 21

2.5 Examples of Large Language Models ... 21
2.5.1 BERT ... 21
2.5.2 GEMINI ... 21
2.5.3 GPT –3 ... 21
2.5.4 GPT -3.5 και GPT –3.5 Turbo ... 22
2.5.5 GPT –4 ... 22
2.5.6 GPT –4o ... 22
3.LARGE LANGUAGE MODELS IN SOFTWARE DEVELOPMENT ... 23
3.1 Role of Large Language Models in Software Development ... 23
3.1.1 Introduction to the Concept of Language Models in Software Development ... 23
3.1.2 Examples of Large Language Models Usage in Software Development ... 23
3.1.3 Code Analysis ... 23
3.1.4 Automatic Code Checks Generation ... 24
3.1.5 Feedback During Code Writing ... 24
3.1.6 Code Documentation ... 24
3.2 Uses in Different Stages of Software Development ... 25
3.2.1 Requirements Analysis Stage ... 25
3.2.2 Design Stage ... 25
3.2.3 Coding Stage ... 25
3.2.4 Testing and Optimization Stage ... 26
3.3 Advantages and Limitations ... 26
3.3.1 Advantages ... 27
3.3.2 Limitations ... 27
3.4 Future Prospects and Development Opportunities ... 28
3.4.1 Collaboration of Large Language Models ... 28
3.4.2 Understanding Different Types of Input ... 28
3.4.3 Adaptation to User Needs ... 28
3.4.4 Improvement in Existing Tasks ... 28
4.BEHAVIOR-DRIVEN DEVELOPMENT (BDD) ... 29
4.1 Test-Driven Development και Behavior-Driven Development ... 29
4.2 The Importance of Collaboration in BDD ... 31
4.2.1 Ubiquitous Language ... 31
4.2.2 Scenarios and User Stories in BDD ... 31
4.3 The Three Practises of BDD ... 32

4.4 Collaboration Analysis Techniques 33
4.5 Test Automation with Cucumber 34
4.5.1 Introduction to Cucumber 34
4.5.2 Gherkin Syntax 35
4.5.3 Creating Feature Files 36
4.5.4 Developing Step Definitions 36
4.6 Cucumber Reports 38
4.7 Live Documentation 38
4.8 Best Practises for writing with Cucumber 39
5.USE OF LARGE LANGUAGE MODELS FOR BDD 39
5.1 Methodology 39
5.1.1 The library Problem 40
5.1.2 Converting Use Cases into Cucumber Scenarios 40
5.1.3 Development of BDD Tests 43
5.1.4 Execution and Verification of BDD Tests 46
5.1.5 Selection of Specific Large Language Models 46
5.1.6 Designing Different Conversations Based on Prior Knowledge 47
5.2 Evaluation Criteria 48
5.3 Empirical Evaluation 53
5.3.1 Phase 1 Evaluation 53
5.3.2 Phase 2 Evaluation 55
5.3.3 Phase 3 Evaluation 57
5.3.4 Phase 4 Evaluation 59
6.CONCLUSIONS 62
7.BIBLIOGRAPHY 64
8.APPENDICES 66
8.1 Phase 1 66
8.1.1 GPT -3.5 Chat 1 66
8.1.2 GPT -3.5 Chat 2 66
8.1.3 GPT -3.5 Chat 3 66
8.1.4 GPT -3.5 Chat 4 67
8.1.5 GitHub Copilot Chat 1 67
8.1.6 GitHub Copilot Chat 2 68
8.1.7 GitHub Copilot Chat 3 69

8.1.8 GitHub Copilot Chat 4 69
8.1.9 GPT -4 Chat 1 69
8.1.10 GPT -4 Chat 2 69
8.1.11 GPT -4 Chat 3 70
8.1.12 GPT -4 Chat 4 70
8.1.13 GPT -4o Chat 1 70
8.1.14 GPT -4o Chat 2 71
8.1.15 GPT -4o Chat 3 71
8.2 Phase 2 72
8.2.1 GPT -3.5 Chat 1 72
8.2.2 GPT -3.5 Chat 2 73
8.2.3 GPT -3.5 Chat 3 73
8.2.4 GitHub Copilot Chat 1 74
8.2.5 GitHub Copilot Chat 2 74
8.2.6 GitHub Copilot Chat 3 74
8.2.7 GitHub Copilot Chat 4 75
8.2.8 GitHub Copilot Chat 5 75
8.2.9 GPT -4 Chat 1 76
8.2.10 GPT -4 Chat 2 76
8.2.11 GPT -4 Chat 3 76
8.2.12 GPT -4 Chat 4 76
8.2.13 GPT -4o Chat 1 77
8.2.14 GPT -4o Chat 2 77
8.2.15 GPT -4o Chat 3 78
8.3 Phase 3 78
8.3.1 GPT -3.5 Chat 1 78
8.3.2 GPT -3.5 Chat 2 79
8.3.3 GPT -3.5 Chat 3 80
8.3.4 GPT -3.5 Chat 4 80
8.3.5 GitHub Copilot Chat 1 81
8.3.6 GitHub Copilot Chat 2 81
8.3.7 GitHub Copilot Chat 3 82
8.3.8 GitHub Copilot Chat 4 82
8.3.9 GPT -4 Chat 1 83

8.3.10 GPT -4 Chat 2 ... 83
8.3.11 GPT -4 Chat 3 ... 83
8.3.12 GPT -4o Chat 1 ... 84
8.3.13 GPT -4o Chat 2 ... 84
8.3.14 GPT -4o Chat 3 ... 84
8.4 Phase 4 ... 84
8.4.1 GPT -3.5 Chat 1 ... 84
8.4.2 GPT -3.5 Chat 2 ... 85
8.4.3 GPT -3.5 Chat 3 ... 85
8.4.4 GPT -3.5 Chat 4 ... 86
8.4.5 GitHub Copilot Chat 1 ... 86
8.4.6 GitHub Copilot Chat 2 ... 86
8.4.7 GitHub Copilot Chat 3 ... 87
8.4.8 GPT -4 Chat 1 ... 87
8.4.9 GPT -4 Chat 2 ... 87
8.4.10 GPT -4 Chat 3 ... 88
8.4.11 GPT -4o Chat 1 ... 88
8.4.12 GPT -4o Chat 2 ... 88
8.4.13 GPT -4 Chat 3 ... 89

LIST OF FIGURES

Figure 1. Example of a simple neural network ... 11
Figure 2. Example Representation of Gradient Descent ... 12
Figure 3. Example of Mathematical Representation of the Gradient ... 13
Figure 4. Example of Transformer Communication in a Large Language Model ... 15
Figure 5. An example chat with chatbot ELIZA ... 17
Figure 6. Comparison of the Performance of Attention and RNNs ... 19
Figure 7. The Development Cycle of TDD ... 30
Figure 8. Example of a user story ... 31
Figure 9. The three practices of BDD ... 33
Figure 10. The Three Amigos Meeting ... 34
Figure 11. Example of a scenario in the form of Given-When-Then ... 35
Figure 12. Example of a scenario in the form of Given-When-Then ... 36

Figure 13. Example of the structure of a feature file 36
Figure 14. Example of the structure of Step Definitions 37
Figure 15. Example of a report with Cucumber 38
Figure 16. Example of the Borrowing Use Case in the Library System........ 41
Figure 17. Example of Cucumber Scenarios for the Loaning Items Use Case........ 42
Figure 18. Example Java Code for Scenario Testing 44

LIST OF TABLES

Table 1. Evaluation criteria of large language models........ 49
Table 2. Phase 1 evaluation of Large Language Models 53
Table 3. Phase 2 evaluation of Large Language Models 57
Table 4. Phase 3 evaluation of Large Language Models 59
Ταβλε 5. Phase 4 evaluation of Large Language Models 61

1. INTRODUCTION

1.1 Large Language Models, BDD, and Code Automation

In the modern era of technology, artificial intelligence and large language models such as OpenAI's GPT-3.5, GPT-4, and GPT-4o have become essential tools across numerous fields in everyday life and computer science technology. Specifically, large language models have radically transformed the way people approach software development, revolutionizing natural language processing by offering capabilities that were inconceivable just a few years ago. Through their training on vast datasets, these models have developed natural language understanding skills to a level that now closely mirrors human communication and interaction.

The potential these large language models present in enhancing the software development process is particularly noteworthy. Specifically, their use in generating automated code within Behavior-Driven Development (BDD) scenarios has garnered significant interest, as it can drastically reduce development time and considerably improve code quality. Behavior-Driven Development (BDD) is a software development methodology focused on understanding and verifying system behavior from the user's perspective. It requires a clear and comprehensible description of system requirements and functions while fostering collaboration among various stakeholders of a software system.

Despite the significant advances made by large language models, their application within specific development methodologies, such as BDD, may still face challenges. These models must be capable of understanding and producing precise, functional code that meets BDD requirements, including Step Definitions and BDD specifications. In particular, these challenges involve accurately interpreting user requirements and generating code that aligns with the provided BDD scenarios.

1.2 Purpose of the Present Thesis

HThis thesis addresses the issue of evaluating the performance of large language models in generating automated code for BDD scenarios, with each scenario being presented to the system with a different level of prior knowledge. The amount of this prior knowledge is divided into four phases, with each phase adding additional information compared to the previous one. Additionally, this thesis aims to determine an optimal methodology for interacting with large language models and presenting knowledge to them in a way that yields the best results in the shortest possible time.

4o

1.3 Structure

Initially, an analysis of the concept of large language models is presented, describing their principles, architecture, operation, and historical development up to the present day. Following this, the technology behind their operation is examined, including their training and the algorithms used in these highly popular models.

Additionally, the use of large language models in the field of software development is explored, covering applications such as code analysis, automated test generation, feedback during code writing, and code documentation. Examples of their applications across different software stages are reviewed, along with their advantages, challenges, and future improvement prospects.

The thesis then provides an in-depth description of the Behavior-Driven Development (BDD) methodology, explaining its primary purpose, characteristics, principles, and application within software development. Moreover, the use of the Cucumber tool in conjunction with BDD is demonstrated, detailing how to create automated tests with this tool, the necessary theoretical knowledge required for its adoption, and real-world examples.

Finally, the use of large language models to support BDD is discussed. Specifically, the methodology followed in conducting the experiment and evaluating the large language models is presented, along with the criteria used for this evaluation and the conclusions regarding the final results. (link of the thesis on github)

2. Large Language Models

2.1 What is a Neural Network and How Does It Work

2.1.1 Definition of Neural Networks

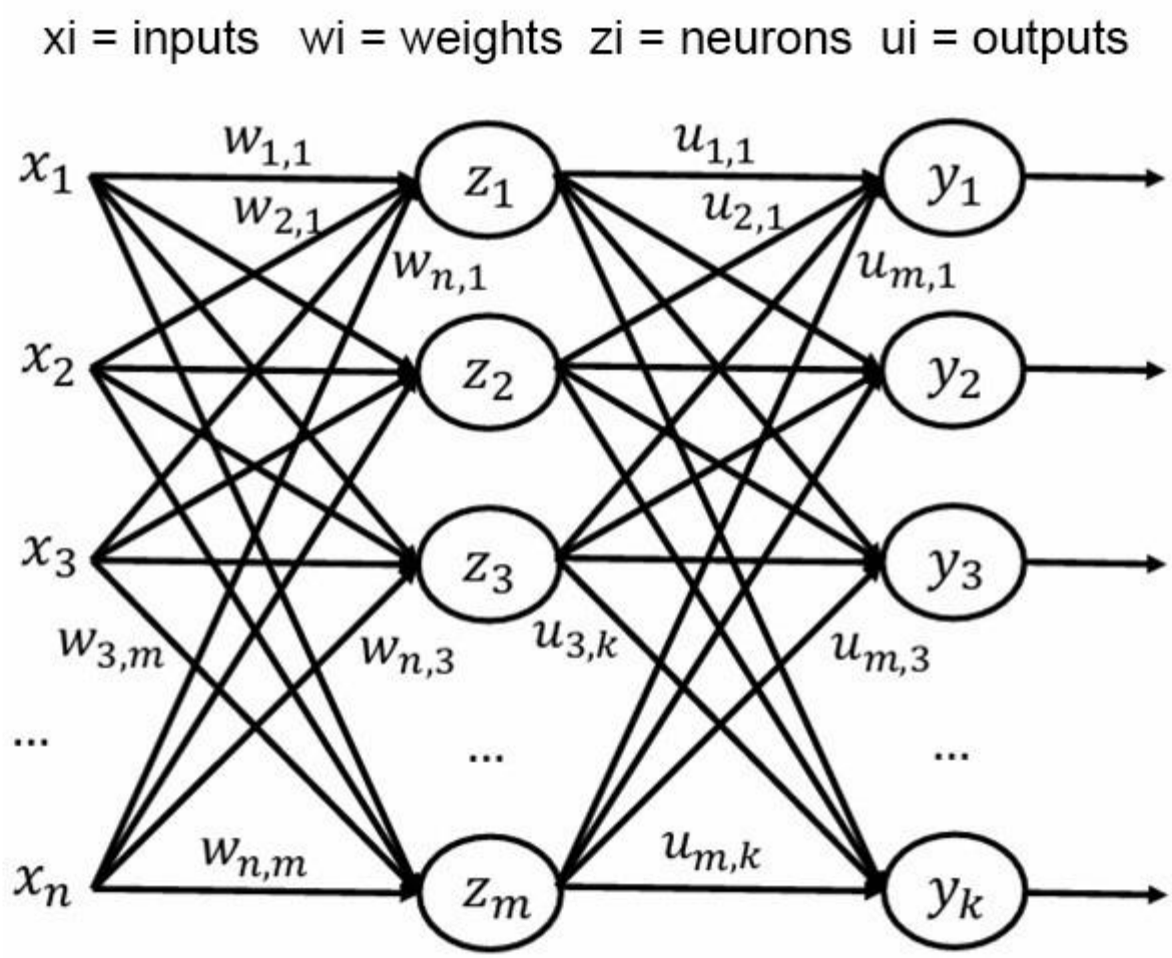
Starting a discussion on large language models and their applications inevitably requires an understanding of their foundational component, the “neural network,” and the ways in which it operates. A neural network, as its name suggests, is a network of many interconnected neurons, organized into different "layers." Each neuron can be thought of as a unit that receives certain inputs, performs simple computations on them, and then produces an output, a number, which in turn is passed as an input to subsequent neurons.

These simple computations involve the sum of the inputs multiplied by a number called a “weight,” which the neural network learns from thousands of data points during its training. Once a mathematical transformation is applied, the result is passed to the neurons in the next layer as input, and this process continues until the final layer of neurons in the network.

For example, a neuron might pass a zero to the neurons in the next layer if the inputs it received were negative or close to zero, and a one in the opposite case. This is known as a “sigmoid activation function.”

Deep learning models, in particular, use millions of neurons organized into numerous layers with billions of weights and complex neuron arrangements, though the basic concept remains the same (Androutsopoulos, 2024). In Figure 1, a simplified form of a neural network is shown, consisting of an input layer, a hidden layer, and an output layer.

Figure 1. Example of a simple neural network



Note: Retrieved from "Artificial Intelligence and Large Language Models," I. Androutsopoulos, ***OPA News****, Athens University of Economics and Business Newspaper, Issue 51, 2024, p. 8,,(link).*

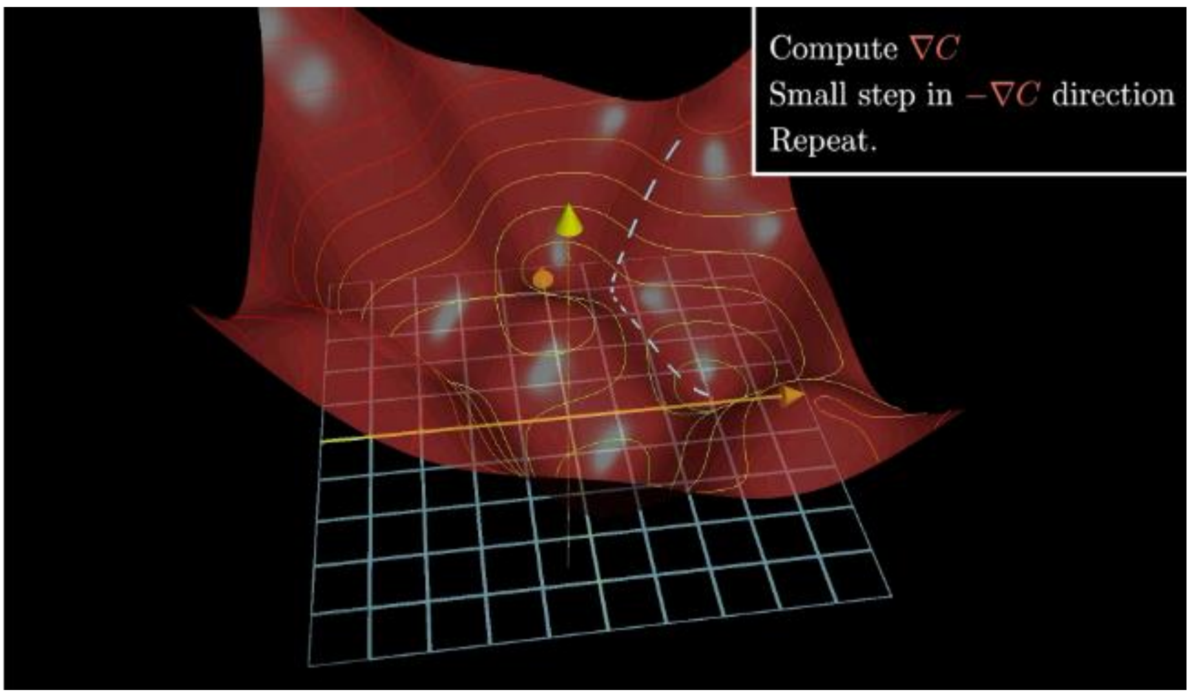
2.1.2 Definition of Gradient Descent

What makes neural networks, and consequently machine learning, so intriguing is that there is no explicitly written algorithm defining what a neural network should do or how it should produce results. Instead, an algorithm is written that, by processing millions of examples and data with their correct "labels" (i.e., the desired outcomes), adjusts the millions of weights within the network to perform better on these examples.

These data are called "training data," and, together with "test data" (examples the neural network has never encountered before, used to evaluate its effectiveness), form the basis for the system's overall evaluation. The task of adjusting the weights is essentially solving the problem of finding the minimum of a "cost function."

This cost function is calculated based on the results produced by the neural network compared to the desired outcomes for each example. Consequently, the function has a high value when the results significantly deviate from the desired ones and a low value when they are close. The algorithm's goal is to use the average of all these cost values for each example to adjust the weights so that the cost function reaches a local minimum.

Figure 2. Example Representation of Gradient Descent

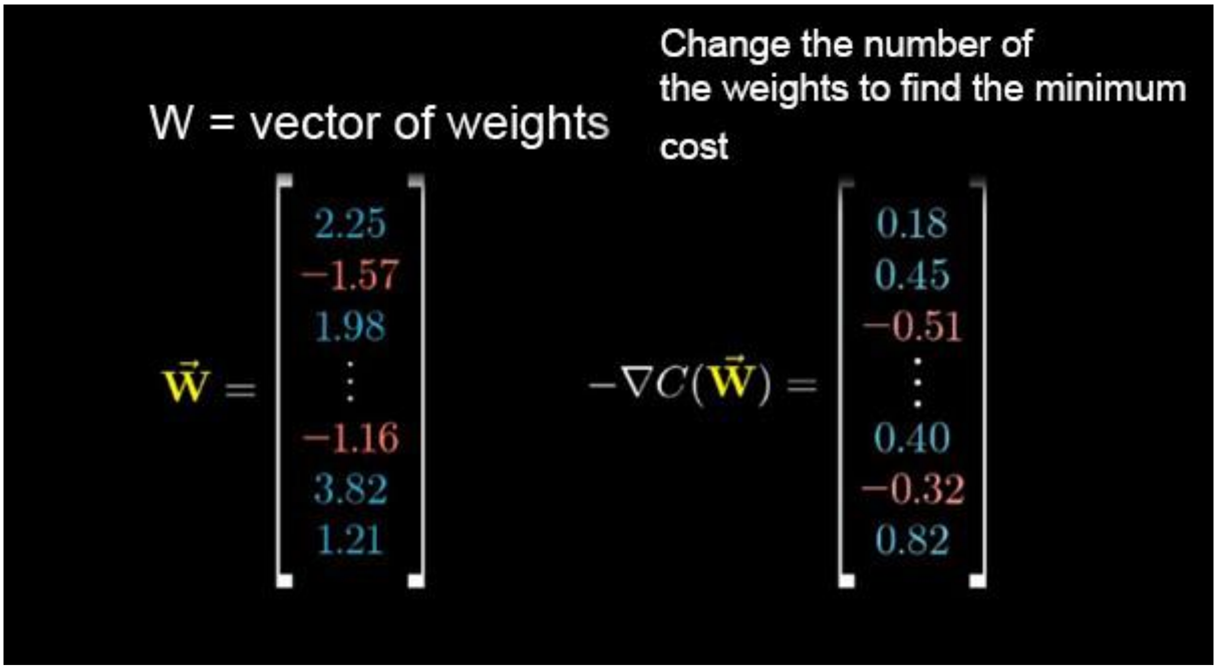


Note: Retrieved from «Gradient descent, how neural networks learn», G. Sanderson-3blue1brown,2017 (link).

This technique is called "**gradient descent**," as it seeks to find a local minimum of a function with thousands of variables (inputs and weights). If represented in a vector space, the function resembles a "geographic landscape" where the goal is to locate a local "lowest point," as illustrated in **Figure 2.**

Using mathematical concepts, the process ultimately involves finding the negative of the "gradient" (∇) of the cost function. The gradient points in the direction of the steepest ascent, so its negative indicates the direction of the steepest decrease in the function. (Sanderson, 2017).

Figure 3. Example of Mathematical Representation of the Gradient



Note: Retrieved from «Gradient descent, how neural networks learn», G. Sanderson--3blue1brown,2017 (link).

Consequently, this section will briefly define the algorithm designed to achieve the aforementioned goal of computing the local minimum of the cost function, known as the "**backpropagation algorithm.**"

2.1.3 Definition of the Backpropagation Algorithm

The backpropagation algorithm is an optimization method that finds the local minimum by calculating the negative gradient of a cost function. Initially, the algorithm initializes all weights in the neural network with random small values. For a given input or training example, it computes the total error or cost function at the output layer by comparing the actual output with the desired output.

Then, the error propagates backward from the output layer to the input layer, calculating derivatives with respect to each individual weight using the chain rule. Each weight is updated in the direction that reduces the error, following the gradient descent approach.

This process is repeated for every training example provided to the neural network in iterations called "epochs." The training process ends either when the system surpasses a maximum number of epochs or when the total error is reduced to a desired level (Ανδρουτσόπουλος, 2023).

2.2 What is a Large Language Model

2.2.1 Definition of a Large Language

As noted by Androutsopoulos (2024), a large language model is a neural network that takes words, or more accurately "tokens," as input in numerical form, representing an incomplete text. It outputs all possible next words, selecting the one with the highest likelihood of being correct.

Based on this principle, a large language model can generate a new sentence by selecting the most probable word. This generated sentence can then be reintroduced as input, prompting the model to produce another possible next word. After several iterations of this process, a coherent and meaningful response is generated.

2.2.2 What a Transformer Does

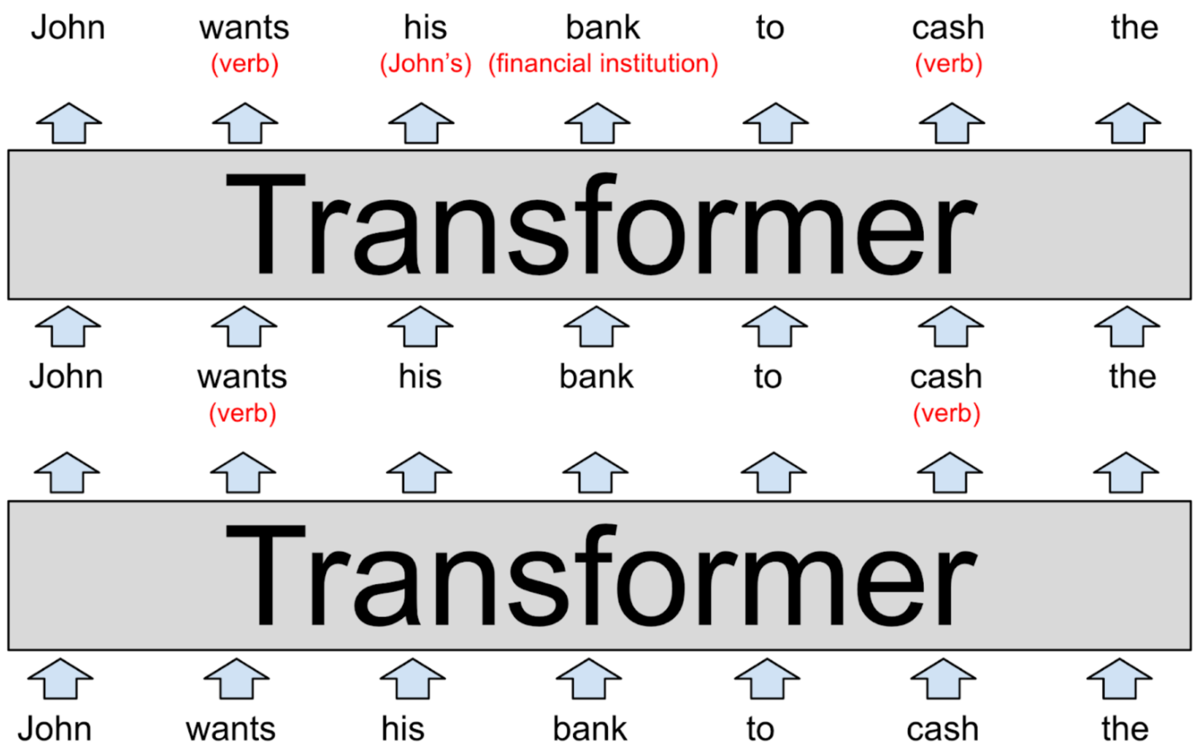
More specifically, large language models break down the input text into distinct tokens, which are not necessarily whole words but can also be parts of words. These tokens are then represented as vectors, consisting of thousands of variables and numbers. These vectors are mapped into a multi-dimensional space where words with similar meanings, such as "cat" and "dog," are located close to each other.

The use of vectors allows large language models to perform mathematical operations that reveal relationships between words. For instance, it has been observed that subtracting the vector for "big" from the vector for "bigger" and adding the vector for "small" yields the vector for "smaller."

Large language models can also represent words with different vectors depending on their context. This is achieved using a neural network architecture known as a **Transformer**, which updates the word vectors through multiple layers.

A large language model comprises many interconnected Transformer layers designed to enrich each token with contextual information, refining its vector to accurately reflect the meaning derived from the surrounding context. Each Transformer layer adds information to the representation, enhancing the model's understanding of the input.

Figure 4. Example of Transformer Communication in a Large Language Model



Note: Retrieved from «Large Language Models, explained with a minimum of math and jargon», T. Lee and S. Trott, 2023, (σύνδεσμος).

And refines the vectors up to the final layer (Lee, 2023).

2.2.3 What is Attention Layer

A key component that allows the transformer to enrich word embeddings is the **'attention layer**.' Specifically, this component enables tokens to exchange information with one another and enrich each other, resulting in the correct meaning of each word. For instance, the word 'model' can have different meanings depending on the context, such as in the phrases 'mathematical model' and 'Hollywood model.' The function of the attention layer is to make the correct distinction for each token based on the surrounding words. This process allows large language models to accurately predict the next word in any given text. (Lee, 2023)

2.2.4 What is the Feed Forward Step

After the information transfer between word embeddings by the attention heads, the transformer includes another component, the 'feed-forward layer.' This layer 'examines' each word embedding and attempts to predict the next word. At this stage, there is no exchange of information between words; the feed-forward layer analyzes each word individually. However, the feed-forward layer has access to any information copied from all the previous attention heads, enabling it to more effectively predict the next word (Lee, 2023).

2.3 History and Evolution of Large Language Models

2.3.1 The ELIZA Chatbot

Starting with a historical overview of the evolution of large language models, reference is made to the chatbot "**ELIZA**," which was created in 1996 and is considered the first chatbot developed by humans. Its creator, Joseph Weizenbaum, developed it at MIT. ELIZA operated by creating the illusion of conversation through the technique of rephrasing users' statements into questions. During that time, several variations of the chatbot were created, which functioned in a similar manner, one of the most famous being "DOCTOR," which responded like a psychotherapist. This concept laid the foundations for further research in the field of chatbots and natural language processing. (Pi, 2024)

Figure 5. An example chat with chatbot ELIZA

```
  Welcome to
                    EEEEEE  LL      IIII   ZZZZZZ   AAAAA
                    EE      LL       II       ZZ   AA   AA
                    EEEEE   LL       II     ZZZ    AAAAAAA
                    EE      LL       II    ZZ      AA   AA
                    EEEEEE  LLLLLL  IIII ZZZZZZ    AA   AA

  Eliza is a mock Rogerian psychotherapist.
  The original program was described by Joseph Weizenbaum in 1966.
  This implementation by Norbert Landsteiner 2005.


ELIZA: Is something troubling you ?
YOU:   Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU:   They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU:   Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU:   He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU:   It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

Note: Retrieved from «ELIZA», In Wikipedia, The Free Encyclopedia (2024, July 15), (σύνδεσμος).

2.3.2 The Rise of Neural Networks

As Pi (2024) mentions in his article, towards the end of the 20th century, neural networks emerged, deeply inspired by the human brain, as evidenced by their name and architecture with interconnected neurons. The year 1986 is noted as the time when **Recurrent Neural Networks (RNNs)** appeared. Unlike traditional feedforward neural networks, where information flows in only one direction, RNNs had the ability to "remember" previous inputs and respond based on a broader context. This allowed them to be trained to process and transform a sequence of input data into a corresponding sequence of output data. However, RNNs had a significant limitation in their "memory," similar to the **context size** in large language models like ChatGPT today, making them seem as though they "forgot" information from earlier messages.

2.3.3 Creation of LSTMs

In 1997, **Long Short-Term Memory (LSTM)** networks emerged, a specialized form of RNN designed to address the issue of retaining information over long sequences. Specifically, this tool had a unique architecture consisting of input gates, forget gates, and output gates, allowing for the retention and management of information over longer periods of time για μεγαλύτερα χρονικά διαστήματα (Pi, 2024).

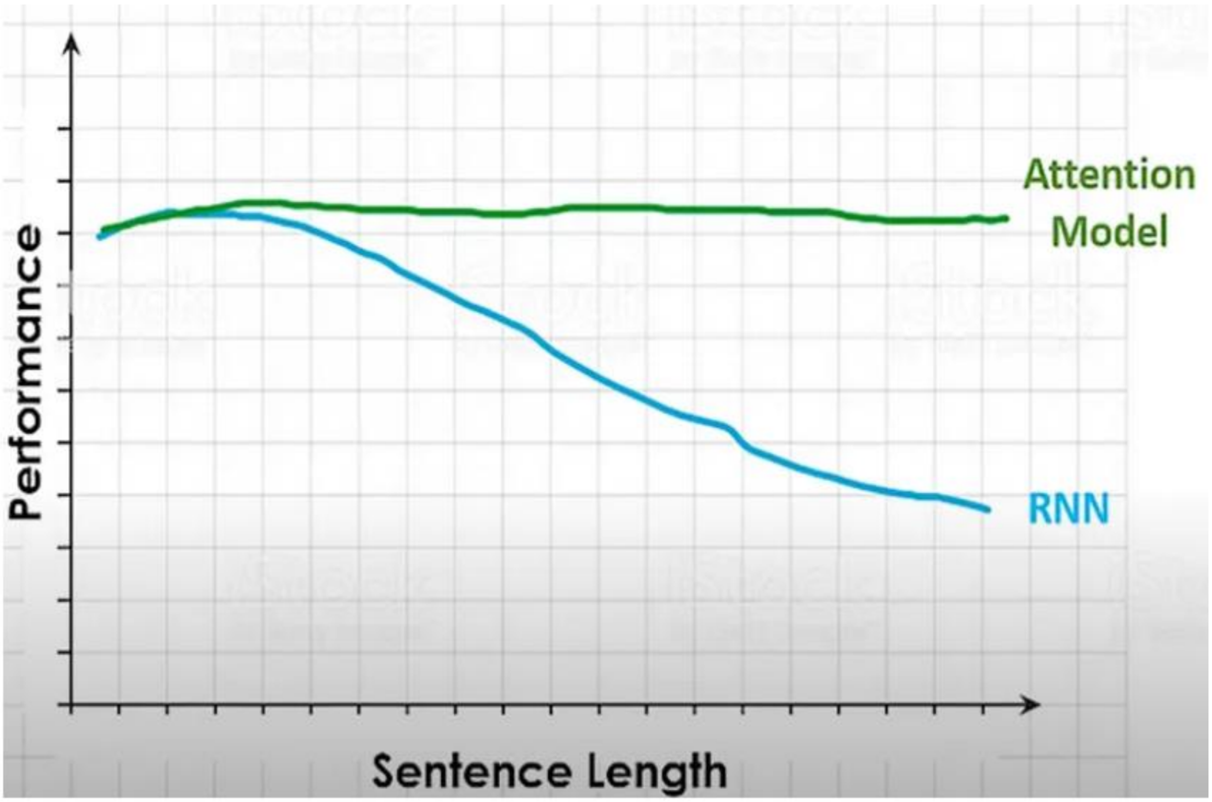
2.3.4 Creation of Gated Recurrent Network

In 2014, **Gated Recurrent Units (GRU)** appeared, designed to solve the same problems as LSTMs but with a simpler structure. GRUs used two gates: an update gate and a reset gate. (Pi, 2024)

2.3.5 The Rise of the Attention Mechanism

Ultimately, technologies such as RNNs, LSTMs, and GRUs proved less effective in maintaining context when it was extensively expanded. This led to the creation of the **Attention mechanism**, which brought a new perspective to large language models. Specifically, attention allowed the model to "look" at all parts of the input sequence simultaneously, improving its ability to capture relationships between distant words or tokens. (Pi, 2024)

Figure 6. Comparison of the Performance of Attention and RNNs



Noted: Retrieved from «Brief Introduction to the History of Large Language Models (LLMs)», W. P,2024 (link)

2.3.6 The Invention of Transformers

In 2017, the concept of **Transformers** was introduced in the paper *"Attention is All You Need"* by Vaswani and his colleagues at Google. This new architecture, as explained in the previous section, utilized the **attention mechanism** as its core tool to process input data and was capable of handling sequences in parallel, divided into multiple layers. This approach laid the groundwork for later models such as **ChatGPT** (Pi, 2024).

2.3.7 Emergence of Large Language Models

With the great success of Transformers, the next logical step was to scale them up. This began with the **BERT** model by Google in 2018 and continued with the release of **GPT-2** in 2019, **GPT-3** in 2020, and newer versions such as **GPT-3.5**, **GPT-4**, and **GPT-4o** (Pi, 2024).

2.4 Main Applications and Uses

With the tremendous evolution of large language models, they have become everyday tools for many people, saving time due to their vast data size and complexity, enabling instant responses without delay. Their cost is minimal, as most are available for free in simpler yet highly effective forms, and their use requires minimal expertise. These factors have driven large language models to find numerous applications in various fields. As Sumrak(2024) mentions, some of the most popular use cases for large language models include audio analysis, content creation, customer support, language translation, education, and cybersecurity.

2.4.1 Audio Analysis

Large language models have revolutionized how people handle audio data. Specifically, they have the ability to listen to long discussions and generate effective summaries, as well as answer questions regarding lengthy meetings. Moreover, based on a large volume of call data, they can extract complex results and provide improvement advice (Sumrak, 2024).

2.4.2 Content Creation

Large language models are effectively used by writers and marketers for drafting initial outlines, suggesting various changes, and quickly finding articles and references online. These tools significantly accelerate productivity, allowing users to focus more on the more demanding and creative aspects of their work while leaving the mechanical elements to the models (Sumrak, 2024).

2.4.3 Customer Support

One area where large language models have seen rapid growth and widespread use is customer support. Many telecommunications companies, public sector pages, and large businesses have adopted customer service models powered by large language models. These models are available continuously without human intervention, providing ongoing assistance to users around the world at a very low cost (Sumrak, 2024).

2.4.4 Language Translation

Large language models help overcome language barriers, enabling businesses to reach customers and hire people from all countries. These models provide accurate real-time translation services, making websites, applications, and digital content globally accessible (Sumrak, 2024).

2.4.5 Education

In the field of education, large language models are used to provide personalized learning by adapting content to individual students' needs, generating comprehension questions, and offering detailed explanations tailored to the students' needs (Sumrak, 2024).

2.4.6 Cybersecirity

Large language models can be used to analyze and interpret large amounts of cybersecurity data, predicting, detecting, and responding to potential security threats. Their targeted training allows faster and more accurate detection and response to threats, enhancing business security (Sumrak, 2024).

2.5 Examples of Large Language Models

To conclude the analysis of large language models, it is important to mention some of the most widely used models and their main features and differentiations.

2.5.1 BERT

The first large language model developed was BERT (Bidirectional Encoder Representations from Transformers), created by Google in 2018. BERT uses a transformer architecture with numerous interconnected layers and has 342 million parameters for processing inputs. It was trained on vast amounts of data to produce natural language responses that are understandable by humans (Lutkevich, 2024).

2.5.2 GEMINI

Following Google, the updated version of their model is called GEMINI, named after the company's chatbot. GEMINI is a multimodal model capable of processing audio, images, video, and text, in contrast to other language models that are limited to text alone. GEMINI is available in three versions: Ultra, Pro, and Nano, ranging from the most powerful to the smallest and least capable. Reports suggest that GEMINI outperforms OpenAI's GPT-4 model, which will be discussed nextOpenAI. (Lutkevich, 2024).

2.5.3 GPT –3

GPT-3 (Generative Pre-trained Transformer 3) is the first powerful model from OpenAI, released in 2020, with more than 175 billion parameters (while BERT has 342 million). It incorporates the transformer architecture with interconnected layers and is ten times larger than its predecessor, **GPT-2**. GPT-3 is trained on millions of data points and is the latest model in the series produced by OpenAI, with the parameter count being publicly knownπαραμέτρων που χρησιμοποιούσε (Lutkevich, 2024).

2.5.4 GPT -3.5 και GPT –3.5 Turbo

GPT-3.5 is the updated version of GPT-3, with fewer parameters but superior training, playing a key role in the ChatGPT platform, which significantly changed the popularity and use of large language models in 2023. GPT-3.5 was trained with knowledge up until September 2021 and does not have internet access, unlike other large language models. The most powerful version of GPT-3.5 is **GPT-3.5 Turbo**, used by GitHub Copilot from Microsoft to assist developers with code generation and error detectionκαι ανίχνευση σφαλμάτων (Lutkevich, 2024).

2.5.5 GPT –4

GPT-4 is the largest model in the GPT series from OpenAI, released in 2023. Like the previous models, it is based on the transformer architecture. However, the number of its parameters has not been disclosed, with rumors suggesting over 170 trillion parameters. OpenAI describes GPT-4 as a multimodal model (like GEMINI), meaning it can process text, audio, and images, unlike previous OpenAI models which were limited to text. GPT-4 is said to approach Artificial General Intelligence (AGI), meaning it is as intelligent or even smarter than a human, and importantly, it has internet access, unlike GPT-3.5 (Lutkevich, 2024).

2.5.6 GPT –4o

GPT-4o (GPT-4 Omni) is the successor to GPT-4 and the latest large language model from OpenAI. It offers several improvements over GPT-4, including more natural human interaction for ChatGPT. GPT-4o is also multimodal, but with the added ability to see images or screenshots and ask relevant questions during interaction.

3. LARGE LANGUAGE MODELS IN SOFTWARE DEVELOPMENT

3.1 Role of Large Language Models in Software Development

3.1.1 Introduction to the Concept of Language Models in Software Development

Large language models are advanced artificial intelligence systems that respond to user queries and discussions with human-like language answers. They have the ability to deeply understand context and user needs, making them useful in various fields of daily life, particularly in software development. Language models assist developers by increasing productivity and reducing debugging time, as they have been trained on billions of open-source projects and are continuously updated with new data.

The internal functioning of large language models makes them highly effective in supporting numerous projects and solving difficulties that may arise during the software development process. They significantly increase productivity, reduce debugging time, and thus shorten the time required for project completion. This is due to the fact that large language models can write, correct, and optimize code quickly and accurately, having been trained on billions of open-source projects, such as those on GitHub, and constantly updated with new data.

3.1.2 Examples of Large Language Models Usage in Software Development

Some of the key examples of using large language models in software development include code analysis, automatic generation of code checks, developer feedback during code writing, and code documentation.

Code analysis can save a lot of time and reduce errors, as language models can generate code based on developers' descriptions and suggest improvements. Automatic generation of checks saves time and minimizes human errors, while identifying all edge cases in system checks. Feedback during writing allows developers to immediately detect mistakes and quickly correct errors, and code documentation improves the readability and maintainability of the code (Ozkaya et al., 2023).

3.1.3 Code Analysis

In the field of code analysis, large language models can analyze code on a large scale with high speed and accuracy, helping developers identify complex problems that may be difficult to spot. The ability of large language models to process vast amounts of code makes them extremely useful tools for recognizing inconsistencies and gaps. Furthermore, large language models can suggest improvements and corrections, accelerating the development process and reducing the risk of errors that could affect the functionality and performance of an application (Ozkaya et al., 2023).

3.1.4 Automatic Code Checks Generation

Large language models can also automate the generation of code checks, which can significantly contribute to improving code quality and the efficiency of the development process. These models can generate checks based on predefined standards and requirements, automatically identifying areas of code that require additional testing or review. This reduces the need for manual check creation, minimizes the possibility of human error, and allows developers to focus on more strategic and creative aspects of software development (Ozkaya και συν., 2023).

3.1.5 Feedback During Code Writing

Tools based on large language models, such as GitHub Copilot, provide real-time feedback during code writing, improving developer efficiency. This feedback includes suggestions for syntax improvement, logical error corrections, and advice on proper code creation. The ability of language models to adapt to the specific needs and coding styles of developers enhances the development process by reducing the likelihood of errors and improving the overall quality of the code. This real-time support allows developers to work more efficiently and ensure that their code meets the required standards (Ozkaya et al., 2023).

3.1.6 Code Documentation

Another example of the use of large language models in the software development process is in code documentation, which becomes more efficient and accurate with the use of these models. Language models can automatically generate comments and documentation for the code, based on analyses of the code itself and related documents. This helps improve the readability and maintainability of the code, ensuring that documentation is always up-to-date and accurate. Additionally, language models can summarize and provide answers to questions about specification documents or policies, facilitating the understanding and application of project requirements (Ozkaya και συν., 2023).

3.2 Uses in Different Stages of Software Development

3.2.1 Requirements Analysis Stage

Requirements analysis is a critical process for the successful development of a software system. A common issue during the requirements analysis stage is ambiguity, meaning the same requirement may be interpreted differently by different individuals, leading to significant issues in later stages of software development. According to Hou et al. (2023), ChatGPT excels significantly over other language models. Additionally, as highlighted by a study cited by Hou et al. (2023), where requirements with significant ambiguity were provided, the language model accurately identified errors and corrected them in every instance. Therefore, large language models serve as a vital tool in eliminating ambiguities in the requirements analysis stage, contributing to higher quality outcomes.

3.2.2 Design Stage

The design stage is also a critical point for the success of a software development project. As Hou et al. (2023) point out, while research on large language models in the design domain has not advanced as much compared to other stages like coding and optimization, these models still offer valuable support in various ways. Specifically, language models can improve existing architecture and suggest alternative solutions when necessary. For example, they can propose specific design patterns that better match the needs of the project.

Moreover, large language models can contribute to the creation of diagrams, not by producing them directly but by describing them through messages, in system descriptions and technical specifications. Additionally, large language models can identify areas where performance or security issues might arise and propose improvements, contributing to a more efficient and secure architecture.

3.2.3 Coding Stage

The coding stage is the most well-known and often the most time-consuming part of software development. During this stage, developers write the code that will implement the project's requirements and specifications. Large language models show continuous performance improvements with each new release. According to Hou et al. (2023), models such as GPT-4, GPT-3, GPT-3.5, BERT series, Codex, CodeGen, InCoder, Copilot, and CodeGeeX play a significant role in coding. Their training on vast amounts of data and specific texts allows them to deeply understand natural language and convert it into code, while they excel at generating code from natural language descriptions, providing real-time corrections and suggestions.

These models also contribute to documentation creation, an important yet often overlooked part of software development. Language models can automatically generate comments and documentation for code, facilitating understanding and maintenance by other developers. Moreover, large language models support multiple programming languages, making them useful for multilingual projects that require the use of various languages, as they can recognize the context and adjust their suggestions based on the programming language being used (Zharovskikh, 2023).

3.2.4 Testing and Optimization Stage

The testing and optimization stage is crucial for ensuring the quality and performance of the software. During this stage, developers and testers identify and fix errors, optimize performance, and ensure the software meets user requirements. Testing and optimization often take a long time, as thorough checks of all written code are required. Furthermore, this process continues after the final product is delivered, especially if new requirements arise from users or project stakeholders.

Large language models offer valuable support in this stage by creating automated tests, identifying errors, and optimizing code. Specifically, large language models can automatically generate unit tests and integration tests based on the written code, ensuring that each piece of code functions correctly and efficiently.

Moreover, large language models can analyze large portions of code and detect potential syntax and logical errors. They provide suggestions for fixing these errors, assisting developers in the debugging process. Additionally, these models can identify inefficient code sections that might give the desired outcome but not in the most optimal way, recommending best practices and improvements that lead to higher quality code.

3.3 Advantages and Limitations

The use of large language models in software development offers numerous advantages and new possibilities for developers, but there are also certain limitations regarding their correct use and the control of the results.

3.3.1 Advantages

Firstly, the ability of large language models to generate code at high speed improves efficiency and reduces errors during the coding process, while also acting as an educational tool for developers by providing immediate explanations and answers regarding code corrections (Hurani & Idris, 2024).

Additionally, through real-time corrections and suggestions offered by these models, they accelerate software development and ensure high quality in the generated code. Moreover, their ability to support multiple programming languages makes them particularly useful for multilingual projects, where understanding large volumes of code in different programming languages can be a challenge for developers involved (Zharovskikh, 2023). Finally, the training of large language models on vast datasets and projects enables a deep understanding and application of best practices in software development, making them valuable tools for developers and large development teams.

3.3.2 Limitations

Along with their advantages, the extensive use of large language models in the software development process also entails certain limitations. Specifically, while large language models excel in simple scenarios and projects, they often exhibit limited accuracy in more specialized or complex scenarios due to their potential lack of training on similar data, resulting in insufficient results (Hurani & Idris, 2024).

Moreover, as Hurani and Idris (2024) state, the use of code generated by large language models increases the need for verification and corrections by developers, as the model may not fully understand the requirements or may produce something close to, but not exactly, what was requested. This can significantly increase debugging time compared to code written by the developers themselves. Lastly, due to their construction, large language models are heavily dependent on the dataset they were trained on, meaning that the quality and content of these datasets significantly influence the generated results.

3.4 Future Prospects and Development Opportunities

Large language models, already powerful tools in software development, are continuously evolving, with new and promising applications and improvements that can further expand their capabilities. With these advancements, which will be discussed below, it is expected that these models will offer even more advanced tools and functionalities for developers to leverage.

3.4.1 Collaboration of Large Language Models

One promising prospect that deserves further exploration, as noted by Hou et al. (2023), is the integration of multiple different large language models into a unified system, where the models collaborate to solve complex tasks in software development. This collaboration would leverage the unique strengths of each model, creating an exceptionally efficient system.

3.4.2 Understanding Different Types of Input

Another significant opportunity for the development of large language models in software development is enhancing their understanding of different types of input, such as images, diagrams, and voice, as well as the automatic generation of complex diagrams like UML. While multimodal models like GPT-4 have made progress in these areas, they have not yet reached a fully promising level to be considered a reliable tool for implementing the above-mentioned tasks (Hou et al., 2023).

3.4.3 Adaptation to User Needs

Additionally, as OpenAI has mentioned, a major evolution is expected to be integrated into its already existing large language models. These models will possess more advanced learning and adaptation capabilities, enabling them to learn from interactions with users and adjust to their specific needs and preferences. These abilities could enhance the accuracy of code suggestions and provide more personalized solutions that better meet project requirements, as the language model would focus on specific domains.

3.4.4 Improvement in Existing Tasks

However, it is equally important for large language models to continue improving in the tasks they already perform well, through training on more and higher-quality data. This includes further enhancing their ability to generate code based on natural language requirements, accurately propose corrections and completions, and reduce errors during development. Strengthening these areas will improve the overall efficiency of these tools and contribute to their better integration and acceptance in the software development process.

4. BEHAVIOR-DRIVEN DEVELOPMENT (BDD)

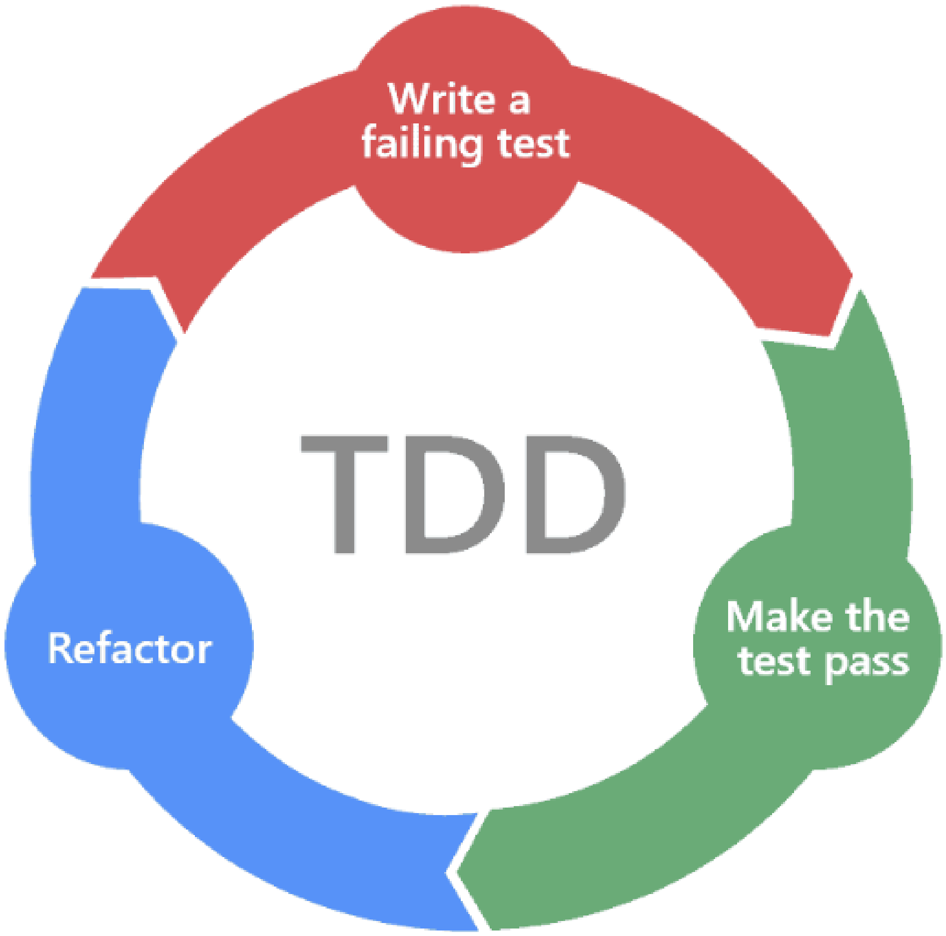
This chapter focuses on the methodology of Behavior-Driven Development (BDD), an evolution of Test-Driven Development (TDD) that aims to promote collaboration between developers, quality assurance specialists, and clients. This methodology enhances collaboration and communication within a software development team, thereby improving the quality and clarity of both the software requirements and functionality.

4.1 Test-Driven Development και Behavior-Driven Development

To begin the chapter, a brief overview of the TDD methodology and its principles will be provided, to later explain the concept of BDD, which is an evolution of TDD.

Test-Driven Development (TDD) is a software development methodology that emphasizes creating tests before the coding process begins, which contrasts with traditional practices that place testing as the final step in the software development process.
TDD asserts that tests should initially be designed with the understanding that they will fail, as the necessary code to pass the tests has not yet been written. The code for the new unit or functionality is then developed with the simplest solution to pass the test. Once the code is implemented, the tests are rerun to verify that the implementation meets the test requirements, and finally, necessary modifications are made to improve the quality of the unit's test implementation (Test-driven development, 2024).

Figure 7. The Development Cycle of TDD



Σημείωση: Retrieved from «Why Test-Driven Development (TDD)», MARNSER,(link)

The key to Test-Driven Development (TDD) is that code coverage is not the final goal but rather the result of the correct application of the TDD methodology. TDD promotes code quality and continuous improvement through iterative cycles of testing and implementation.

The Behavior-Driven Development (BDD) methodology is primarily based on the idea of TDD and represents an evolution of it, influenced by various other Agile methodologies. Its main objective is to encourage collaboration among all stakeholders in the software development process of a project. Specifically, as Fitzgibbons (2021) states, BDD promotes the documentation and design of an application based on the expected behaviors that the user will experience, encouraging developers to focus on the essential behaviors the system should exhibit. This approach helps avoid the creation of unnecessary code and excessive features.

4.2 The Importance of Collaboration in BDD

BDD achieves its goal of fostering collaboration between all stakeholders in a project through the use of a shared, comprehensible language, known as **Ubiquitous Language**. By using a common language, the understanding of the project requirements and the quality of the produced software improve, as all participants, regardless of their role, can contribute to the final outcome.

4.2.1 Ubiquitous Language

A core aspect of collaboration in BDD is the use of a language that is understood by all involved team members, regardless of their specific domain. This language is called **Ubiquitous Language** and is used in all communications regarding the software. An example of such a language, commonly used in the Cucumber tool (to be discussed later), is **Gherkin**. Gherkin is a structured language used to create tests in BDD, and it uses a specific syntax that is easy to read by all, written in natural language (Cucumber, n.d.).

4.2.2 Scenarios and User Stories in BDD

The first process in any project developed using BDD methodology is defining the system's behavior from the perspective of the end users. This includes the creation of **user stories** and **scenarios** that describe how the system should behave in various situations it might encounter during operation.

User stories are written in natural language and follow the format: "As a [user role], I want to [user action with the system], so that [desired result, goal, or benefit]." An example is shown in **Figure 8**, where a hypothetical user story in a library system is presented. In this case, the librarian (user) wants to be able to efficiently manage the borrowing process so that borrowers can easily borrow books from the library. (Cucumber, n.d.)

Figure 8. Example of a user story

```
User Story: As a librarian,
I want to efficiently manage the loaning process of books to registered borrowers
So that borrowers can easily borrow books from the library.
```

Many scenarios can belong to a common feature of the system, referred to as a "**feature**." For example, for the "User Registration" feature, there could be scenarios that examine what happens in the case of successful registration, when the user already has an account, and many other possibilities.

The scenarios are written using the **Ubiquitous Language Gherkin**, as mentioned earlier, and follow a specific structure, typically in the form of "**Given-When-Then**" (this sequence is usually followed). This structure will be explained further below.

Scenarios written with this structure are easily understandable by all stakeholders and help ensure that the user requirements are fully met by the system.

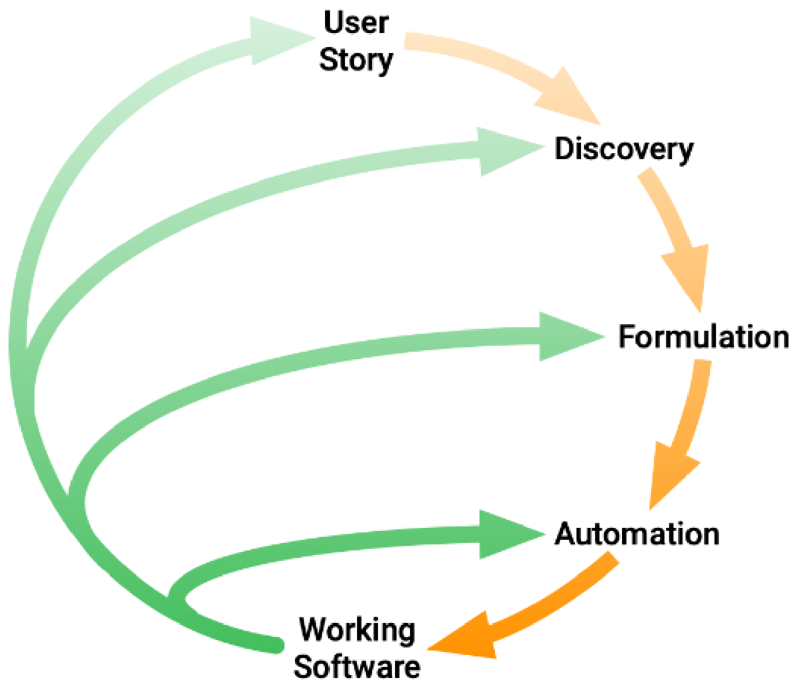
ενδιαφερόμενους και βοηθούν στην εξασφάλιση ότι οι απαιτήσεις των χρηστών ικανοποιούνται πλήρως από το σύστημα (Cucumber, n.d.).

4.3 The Three Practises of BDD

As mentioned in the official Cucumber documentation, BDD, like TDD, encourages working in **rapid iterations**, breaking down user problems into smaller pieces to be solved quickly and correctly.
In essence, the daily workflow of a project developed with BDD consists of three main processes:

1. **Example Analysis** : This starts with a small upcoming change, a user story as described earlier, and breaks it down into real examples (scenarios).
2. **Example Automation** : The examples are then modified to be automatable as **Gherkin** scenarios
3. **Code Implementation** : The system code is implemented to fulfill the behaviors described by these examples.

Figure 9. The three practices of BDD



Noted: Retrieved from «Behavior-Driven Development», Cucumber (link)

4.4 Collaboration Analysis Techniques

A key aspect of the smooth operation of the **BDD methodology** is a meeting known as the **"Three Amigos Meeting"**. As explained below, this meeting is a "discussion" among different stakeholders to understand and create requirements or scenarios.

The **Three Amigos Meeting** primarily aims to convert user stories into understandable and comprehensive **Gherkin scenarios** that are agreed upon by all stakeholders. It involves at least three different roles:

1. **Product Owner** - Focuses on the scope of the application and transforms user stories into features, deciding what is included and what is excluded within the scope.
2. **Tester** - Creates scenarios and edge cases, ensuring that all user stories are covered, and identifies any gaps in test coverage.
3. **Developer** - Adds steps to the scenarios and examines the application's implementation details, such as execution and potential obstacles during code development.

These discussions lead to the creation of high-quality tests, as each participant views the system from a different perspective. For this reason, it is essential that all the mentioned roles participate in the meeting to discover new use cases (Cucumber, n.d.)

Figure 10. The Three Amigos Meeting

Note: Retrieved from «The 3 Amigos Meeting», AWH, 2020, Medium. (link)

4.5 Test Automation with Cucumber

4.5.1 Introduction to Cucumber

To **Cucumber** is a software tool that supports the Behavior-Driven Development (BDD) methodology, leveraging the use of the Gherkin language, as briefly introduced above. **Gherkin** allows for the description of system requirements and the automation of tests. Specifically, Cucumber plays a key role in automating system tests by providing a common language that is understandable to all stakeholders.προσφέροντας μία κοινή γλώσσα που είναι κατανοητή από όλους τους ενδιαφερόμενους.

4.5.2 Gherkin Syntax

Gherkin is a structured language used by the Cucumber tool to create tests in BDD. Its syntax is simple and understandable, enabling both technical and non-technical users to write scenarios. The language follows the "Given-When-Then" format, which allows easy writing of scenarios that are comprehensible to all team members.

These key words serve as the main writing structure for feature scenarios, and each one has a distinct meaning, as described in the official Cucumber documentation:

1. The keyword **Given**, defines the general context of the scenario, setting the system in a specific state so that the scenario can unfold in the subsequent steps. It often involves preparing the system, creating data, or ensuring that the system is in a particular initial state.
2. The keyword **When**, This keyword represents an action that changes the system from its "idle" state, forcing it to produce a result. It is typically the central action of the scenario and represents the event that triggers the system's response.
3. The keyword **Then**, represents an action that changes the system from its "idle" state, forcing it to produce a result. It is typically the central action of the scenario and represents the event that triggers the system's response.

In **Figure 11**, an example scenario written in Gherkin is presented, which concerns updating the details of a borrower in a hypothetical library system. This scenario belongs to a feature related to "Borrower Management". As seen in the example, each scenario also includes a brief title describing the use case addressed by the scenario that follows.

Figure 11. Example of a scenario in the form of Given-When-Then

```
Scenario: Updating the borrower's details when he is registered
This scenario describes the process of updating the details of a borrower who has already registered before
  Given George Red is registered as a borrower
  When George Red updates his borrowing details
  Then the system saves the changes
```

In Figure 12, another example of a hypothetical scenario in the same system as in Image 11 is presented, which concerns the case of a successful registration of a borrower in a library system.

Figure 12. Example of a scenario in the form of Given-When-Then

```
Scenario: Registering a new borrower
This scenario describes the process of registering a new borrower in the library system
  Given George Red is not registered as a borrower
  When George Red gets registered in the system with a unique borrower number and his details
  Then the system successfully stores the borrower's details
```

4.5.3 Creating Feature Files

Τα **Feature files**, as mentioned earlier, are files that contain Gherkin scenarios describing various use cases of a specific feature of the system. The structure of feature files is shown in Image 13 and starts with the keyword **"feature"** to initialize the file. After that, the feature name follows, which will also be the name of the file, along with a brief description of the feature and the scenarios that will follow. Additionally, a good practice is to add a user story to help better understand the set of scenarios that will follow.

Figure 13. Example of the structure of a feature file

```
Feature: Borrower handling by the system
  The system can register a new person, modify their credentials or delete their account

  User Story: As a librarian,
  I want to efficiently manage the loaning process of books to registered borrowers
  So that borrowers can easily borrow books from the library.

  Scenario: Registering a new borrower
  This scenario describes the process of registering a new borrower in the library system
    Given George Red is not registered as a borrower
    When George Red gets registered in the system with a unique borrower number and his details
    Then the system successfully stores the borrower's details

  Scenario: Borrower trying to register has registered before
  This scenario describes what happens when the system tries to register a new borrower who has already registered before
    Given George Red is registered as a borrower
    When the system attempts to register George Red with a unique borrower number and his details
    Then the system informs that the user already exists
```

4.5.4 Developing Step Definitions

Τα **Step Definitions** are code segments that link Gherkin-written scenarios to the actual code implementation so that the tests can pass and the system can be tested. Step Definitions can be written in various programming languages, such as Java, Ruby, and others, and define the behavior of each step in a Gherkin scenario.

Cucumber executes the scenarios found in feature files by running the corresponding Step Definitions code, ensuring that the steps are executed in the correct order, simulating the behavior described in the Gherkin scenarios.

The structure of Step Definitions, as shown in Image 14, includes helper data structures and objects used in the Step Definitions code. To link the steps of a scenario to a Step Definition, we use the format @Given, @When, or @Then, followed by the text of the step in the Gherkin scenario.

Figure 14. Example of the structure of Step Definitions

```
public class StepDefinitions {
    2 usages
    private Person sean;
    3 usages
    private Person lucy;
    2 usages
    private String messageFromSean;

    @Given("Lucy is {int} metres from Sean")
    public void lucy_is_located_metres_from_Sean(Integer distance){
        lucy = new Person();
        sean = new Person();
        lucy.moveTo(distance);
    }
    @When("Sean shouts {string}")
    public void sean_shouts(String message) {
        sean.shout(message);
        messageFromSean = message;
    }
    @Then("Lucy should hear Sean's message")
    public void lucy_should_hear_sean_s_message() {
        assertEquals(asList(messageFromSean), lucy.getMessagesHeard());
    }
}
```

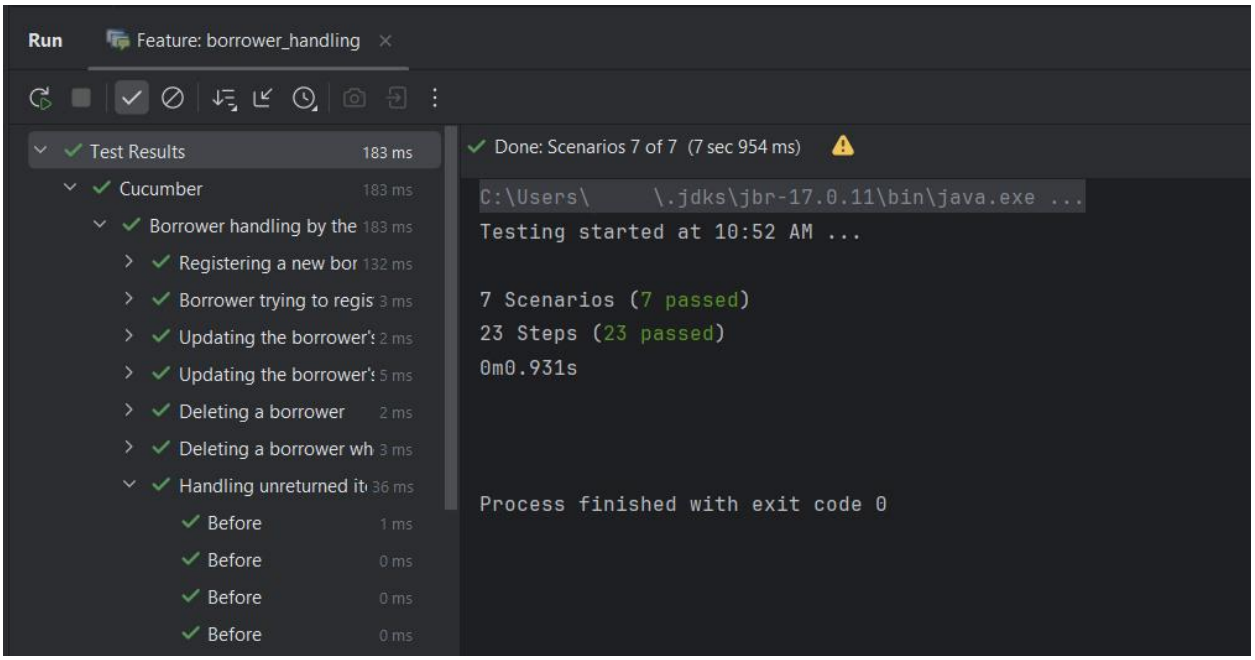
In the case of **Figure 14**, the first step of the Step Definition initializes the environment and the objects that will be used in the subsequent steps of the scenario, in a hypothetical messaging application system that delivers messages to people within a specific distance.

Step Definitions must be unique across feature files so that each step of a scenario can be matched to a single Step Definition. This helps to avoid confusion and ensures the correct execution of the tests.

4.6 Cucumber Reports

After executing the steps of the scenarios from the Step Definitions, Cucumber generates detailed reports and logs of the execution that followed. These reports include information about the steps that passed successfully, those that failed, and any that were skipped due to errors. These reports are a vital tool for developers and quality assurance specialists, as they help them understand the system's status and identify any potential issues. (Cucumber, n.d.)

Figure 15. Example of a report with Cucumber



4.7 Live Documentation

When using BDD as a development methodology for a system, examples or scenarios are written to guide development. These examples also serve as acceptance criteria and are written in a common language, making them understandable to all stakeholders. Therefore, these examples or scenarios function as "living" documentation of the system. Unlike traditional acceptance criteria documents, this documentation is continuously updated with every new change or addition of functionality to the system.

The importance of live documentation is especially significant in a software development environment, as it enables all team members to understand the system's requirements. This is particularly useful for new team members who are not already familiar with the system being developed. Live documentation ensures that knowledge of the system is maintained and effectively shared throughout the team (Cucumber, n.d.).

4.8 Best Practises for writing with Cucumber

Using Cucumber for test automation and adopting the BDD methodology can significantly contribute to improving the quality of the produced software. However, achieving this goal depends largely on how the scenarios are written and organized into feature files. Some best practices for writing scenarios with Cucumber, as mentioned by Pathak (2022), include the following:

1. **Small Scenarios** :Scenarios written for a feature, describing a specific use case of the system, should be small and focus on a single functionality of the system.
2. **Use Real-World Examples**: Scenarios should be based on real-world examples to ensure they are realistic and that the system will work correctly under real-world conditions.
3. **Reuse Steps** : Whenever possible, steps from existing scenarios should be reused to reduce repetition and make scenario maintenance easier.
4. **Organize feature files** : Feature files should be organized in a way that makes finding and understanding scenarios easier. Each feature file should contain scenarios related to a specific feature or functionality of the system, using clear titles and descriptions.

5. USE OF LARGE LANGUAGE MODELS FOR BDD

This chapter analyzes the methodology, evaluation criteria, and empirical assessment of the experiment conducted within the scope of this thesis.

5.1 Methodology

In this thesis, a systematic approach was followed to evaluate the performance of different large language models in a specific use context, based on the library management problem.

5.1.1 The library Problem

As an initial reference for the experiment discussed in the thesis, the library management problem was chosen. This problem involves an application composed of a series of typical use cases that cover the basic functions of a library, such as:

1. **Book Copy Search** : Finding book copies in the library's collection based on various criteria.
2. **Book Return** : Managing the process of book returns by borrowers and handling delays.
3. **Managing Borrowers**: Registering new borrowers, updating their information, deleting their records.

This problem was selected as the reference for the thesis because it is a well-documented, simple, and clear problem, created by the thesis advisor, Mr. Diamantidis. It provides a straightforward scenario in which testing scripts and automated code can easily be created using the BDD methodology and the Cucumber tool. This allows for a comparison with the results from the large language models (link to the library system on GitHub).

5.1.2 Converting Use Cases into Cucumber Scenarios

After the library management problem was chosen as the reference, the next step for deeply understanding the library system and the BDD methodology, along with the use of the Cucumber tool, was to convert the library's use cases into scenarios written in the Gherkin language. This process allowed for the automation of the scenarios with Cucumber and the creation of BDD tests, providing a benchmark for comparing the results of large language models.

For each use case of the library system, within the scope of the application, a corresponding feature was created consisting of Gherkin scenarios describing the system's behavior based on real-world examples. In **Figure 16,** an example of a use case for the library system is presented, describing the main flow of the borrowing process for a book copy and all the alternative flows it includes.

Figure 16. Example of the Borrowing Use Case in the Library System

βασική ροή «δανεισμός αντιτύπων»

1. Ο δανειζόμενος έρχεται στο βιβλιοθηκονόμο κρατώντας τα αντίτυπα των βιβλίων προς δανεισμό.
2. Ο βιβλιοθηκονόμος αναζητά τον δανειζόμενο.
3. Το Σύστημα παρουσιάζει τα στοιχεία του δανειζομένου.
4. Ο βιβλιοθηκονόμος αναζητά το αντίτυπο.
5. Το Σύστημα παρουσιάζει τα στοιχεία του αντιτύπου.
6. Ο βιβλιοθηκονόμος επιλέγει το αντίτυπο προς δανεισμό.
7. Το Σύστημα επιβεβαιώνει ότι ο δανειζόμενος μπορεί να δανειστεί το αντίτυπο.
8. Το Σύστημα καταχωρίζει το δανεισμό και εμφανίζει την προθεσμία επιστροφής.
9. Ο βιβλιοθηκονόμος ενημερώνει τον δανειζόμενο για την προθεσμία επιστροφής του αντιτύπου.

Ο βιβλιοθηκονόμος επαναλαμβάνει τα βήματα 4 έως 9 για όλα τα αντίτυπα.

εναλλακτικές ροές «δανεισμός αντιτύπων»

* Σε οποιοδήποτε σημείο το λογισμικό καταρρέει.

1. Ο βιβλιοθηκονόμος εκκινεί το Σύστημα.
2. Το Σύστημα ταυτοποιεί το βιβλιοθηκονόμο.
3. Ο βιβλιοθηκονόμος εκκινεί το δανεισμό για τα εναπομείναντα αντίτυπα.

2α. Ο δανειζόμενος έρχεται για πρώτη φορά για δανεισμό.

1. Ο βιβλιοθηκονόμος επιβεβαιώνει ότι ο δανειζόμενος μπορεί να δανειστεί βιβλία από τη Βιβλιοθήκη.
    - 1α. Ο δανειζόμενος δε δικαιούται να δανειστεί από τη Βιβλιοθήκη.
        1. Ο δανεισμός τερματίζει.
2. Ο βιβλιοθηκονόμος καταχωρίζει τον δανειζόμενο στο σύστημα με τη Διαχείριση Δανειζομένου.

εναλλακτικές ροές «δανεισμός αντιτύπων»

5α. Το Σύστημα δε βρίσκει το αντίτυπο του βιβλίου

1. Ο βιβλιοθηκονόμος κρατά το αντίτυπο για να διαπιστώσει το σφάλμα αργότερα.
2. Ο δανεισμός τερματίζει.

7α. Ο δανειζόμενος δεν μπορεί να δανειστεί βιβλία.

1. Ο βιβλιοθηκονόμος ενημερώνει το δανειζόμενο.
2. Κρατά τα εναπομείναντα αντίτυπα για να επιστρέψουν στα ράφια.
3. Ο δανεισμός τερματίζει.

Notes: Retrieved from «Τεχνολογία Λογισμικού Β'ΕΚΔΟΣΗ», των Ε.Α.Γιακουμάκη και Ν.Α. Διαμαντίδη, page. 127-128 (2021)

Subsequently, this specific use case was transformed into a feature with scenarios describing the usage cases. As shown in **Figure 17**, the feature "Loaning Items" consists of various scenarios that utilize characters or "personas," as they are referred to in many articles, for a realistic presentation of the example and better understanding by the readers.

Figure 17. Example of Cucumber Scenarios for the Loaning Items Use Case

```
Feature: Loaning items
  The library application allows for the librarian to loan an item of a book to a borrower based
  on some conditions of the system

  User story: As a library member
  I want to be able to borrow items
  So that I can study them at home


  Scenario: Successful loaning of an item
  This scenario describes the successful process of loaning an item to a borrower that is entitled to borrow
    Given the library has the item Harry Potter available
    And George Red is a registered borrower
    And George Red has 2 pending items to be returned
    And George Red has been assigned a maximum lending limit of 5
    When George Red borrows the item Harry Potter
    Then the system successfully loans the item Harry Potter to George Red with a due date set
    And George Red's pending items increase to 3

  Scenario: Borrower can borrow only one item due to his lending limit
  This scenario describes the successful process of loaning only one but not two items to a borrower that is entitled to borrow only one item
    Given the library has the items Harry Potter and Moby Dick available
    And George Red is a registered borrower
    And George Red has 2 pending items to be returned
    And George Red has been assigned a maximum lending limit of 3
    When George Red tries to borrow both items
    Then the system successfully loans the item Harry Potter to George Red with a due date set
    And the system does not loan Moby Dick to George Red due to the lending limit reached
    And George Red's pending items increase to 3

  Scenario:Item not found
  This scenario describes the edge case where the library system cannot find the item, so the loan isn't happening
  This scenario describes the edge case where the library system cannot find the item, so the loan isn't happening
    Given the item Harry Potter is in the library but not in the system
    And George Red is a registered borrower
    When George Red tries to borrow the item Harry Potter
    Then the system returns an error due to the item's status
    And the system withdraws the item Harry Potter

  Scenario:The borrower is not eligible to borrow
  This scenario describes the unsuccessful process of loaning an item to a borrower that has reached his max lending limit
    Given the library has the item Harry Potter available
    And George Red is a registered borrower
    And George Red has 3 pending items to be returned
    And George Red has been assigned a maximum lending limit of 3
    When George Red tries to borrow the item Harry Potter
    Then the system doesn't allow the loan
    And George Red's pending items remain 3
```

Specifically, scenarios were created for the successful loaning of items, cases where a borrowing limit applies to the borrower, instances where a specific item is unavailable, and situations where the borrower does not meet the borrowing criteria.

5.1.3 Development of BDD Tests

The next phase after drafting the Cucumber scenarios for the library management problem was the creation of automated tests, also known as Step Definitions, which the Cucumber tool used to validate the scenarios.

For each scenario, corresponding classes and methods were implemented in Java to test the functionality of each step against the actual library system.

During the implementation of the Step Definitions, **Data Access Objects (DAOs)** were utilized to simulate a real database, storing data and variables derived from the tests. Furthermore, the **Dependency Injection** technique was employed to reduce code complexity and enhance reusability. This technique essentially creates a shared class containing all utility methods used by various feature files.

However, this approach was not requested by the large language models, as it increased complexity without providing additional information. **Figure 18** illustrates the Step Definitions code written for the "Loaning Items" feature mentioned earlier.

Figure 18. Example Java Code for Scenario Testing

```java
package hellocucumber.StepDefinitions;

import ...

public class itemLoaningStepDefinitions {  Konstantinos Platias
    // Instance variables
    private LocalDate dueDate1; //Due date for the book Harry Potter  5 usages
    private LocalDate dueDate2; //Due date for the book Moby Dick  2 usages
    private Borrower georgeRed;  12 usages
    private Item hPotterItem;  10 usages
    private Item mDickItem;  2 usages
    private final LibraryWorld world;  18 usages
    public itemLoaningStepDefinitions(LibraryWorld world){  no usages  Konstantinos Platias
        this.world=world;
    }
    @Before  Konstantinos Platias
    public void setUp(){
        world.clearItems();
        world.clearLoans();
        world.clearBorrowers();
    }
    @Given("the library has the item Harry Potter available")  Konstantinos Platias
    public void givenItemHarryPotterAvailable() {
        // Create and save the item/book Harry Potter
        hPotterItem=world.createItem( title: "Harry Potter",  number: 1001112);
    }
    @Given("{borrower} is a registered borrower")  Konstantinos Platias
    public void givenBorrowerRegistered(Borrower borrower) {
        georgeRed = borrower;
    }
    @Given("George Red has {int} pending items to be returned")  Konstantinos Platias
    public void givenPendingItemsToBeReturned(Integer pendingItems) {
        // Adding dummy loans to create pending items (not reflected in DAO)
        for(int i=0;i<pendingItems;i++) {
            Loan dummy_loan = new Loan();
            dummy_loan.setBorrower(georgeRed);
        }
    }
    @Given("George Red has been assigned a maximum lending limit of {int}")  Konstantinos Platias
    public void givenBorrowerMaxLendingLimit(Integer maxLendingLimit) {
        BorrowerCategory category = new BorrowerCategory();
        category.setMaxLendingItems(maxLendingLimit);
        category.setMaxLendingDays(5); //dummy number that will be used later
        georgeRed.setCategory(category);
    }
    @When("George Red borrows the item Harry Potter")  Konstantinos Platias
    public void  whenBorrowerBorrowsItemHarryPotter() {
        // Attempt to borrow the item and get due date if successful
        if(world.loanService.findBorrower(georgeRed.getBorrowerNo())){
            dueDate1 = world.loanService.borrow(hPotterItem.getItemNumber());
        }
    }
    @Then("the system successfully loans the item Harry Potter to George Red with a due date set")  Konstantinos Platias
    public void thenSystemLoansItemHarryPotterToGeorgeRed() {
        Assertions.assertEquals(georgeRed,world.loanDao.findPending(hPotterItem.getItemNumber()).getBorrower());
        Assertions.assertNotNull(dueDate1);
    }
    @Then("George Red's pending items increase to {int}")  Konstantinos Platias
    public void thenPendingItemsIncrease(Integer pendingItems) {
        Assertions.assertEquals(pendingItems, georgeRed.countPendingItems());
    }
    @Given("the library has the items Harry Potter and Moby Dick available")  Konstantinos Platias
```

```
public void givenItemsHarryPotterAndMobyDickAvailable() {
    // Create and save the book and the item Harry Potter
    hPotterItem=world.createItem( title: "Harry Potter",  number: 1001112);
    // Create and save the book and the item Moby Dick
    mDickItem = world.createItem( title: "Moby Dick", number: 1001233);
}

@When("George Red tries to borrow both items")  Konstantinos Platias
public void whenBorrowerTriesToBorrowBothItems() {
    if(world.loanService.findBorrower(georgeRed.getBorrowerNo())){
        dueDate1 = world.loanService.borrow(hPotterItem.getItemNumber());
    }
    if(world.loanService.findBorrower(georgeRed.getBorrowerNo())){
        dueDate2 = world.loanService.borrow(mDickItem.getItemNumber());
    }
}

@Then("the system does not loan Moby Dick to George Red due to the lending limit reached")  Konstantinos Platias
public void thenSystemDoesNotLoanMobyDickToGeorgeRed() {
    Assertions.assertNull(dueDate2);
    Assertions.assertEquals(georgeRed.countPendingItems(), georgeRed.getCategory().getMaxLendingItems());
}

@Given("the item Harry Potter is in the library but not in the system")  Konstantinos Platias
public void givenItemHarryPotterInLibraryNotInSystem() {
    // Create Harry Potter item and mark it as withdrawn
    hPotterItem=world.createItem( title: "Harry Potter",  number: 1001112);
    hPotterItem.withdraw();
}


    @When("George Red tries to borrow the item Harry Potter")  Konstantinos Platias
    public void whenBorrowerTriesToBorrowItemHarryPotter() {
        if(world.loanService.findBorrower(georgeRed.getBorrowerNo())){
            dueDate1 = world.loanService.borrow(hPotterItem.getItemNumber());
        }
    }
    @Then("the system returns an error due to the item's status")  Konstantinos Platias
    public void thenSystemReturnsErrorDueToItemsStatus() { Assertions.assertNull(dueDate1); }
    @Then("the system withdraws the item Harry Potter")  Konstantinos Platias
    public void thenSystemWithdrawsItemHarryPotter() {
        // Assert that the item's state is withdrawn
        Assertions.assertEquals(ItemState.WITHDRAWN, hPotterItem.getState());
    }

    @Then("the system doesn't allow the loan")  Konstantinos Platias
    public void thenSystemDoesNotAllowLoan() {
        Assertions.assertNull(world.loanDao.findPending(hPotterItem.getItemNumber()));
    }
    @Then("George Red's pending items remain {int}")  Konstantinos Platias
    public void thenBorrowerPendingItemsRemain(Integer pendingItems) {
        Assertions.assertEquals(pendingItems, georgeRed.countPendingItems());
    }
}
```

5.1.4 Execution and Verification of BDD Tests

After developing the BDD tests, the scenarios were executed to validate their correctness in Gherkin. The execution of the BDD tests involved automated execution of the scenarios using the Cucumber tool, which streamlines this process and generates reports on the test outcomes, as illustrated in Figure 15 from the previous chapter.

The BDD tests (Step Definitions) verified whether the system responded appropriately under given conditions and if the outcomes aligned with the expected behaviors described in the scenarios.

In summary, using the library system, converting the use cases into Cucumber scenarios, and developing BDD tests were crucial steps in the methodology of this study. This process enabled a systematic evaluation of the large language models' ability to perform the predicted functions of the library system and laid the groundwork for comparing the models' performance using objective criteria.

5.1.5 Selection of Specific Large Language Models

For the experiment's implementation, specific large language models were selected, recognized for their speed and ability to produce high-quality outputs. The selection of these models was necessary to derive reliable conclusions regarding their performance in generating automated code.

The first model utilized was GitHub Copilot by Microsoft, based on the GPT-3.5 Turbo model. GitHub Copilot is specifically tailored for code generation and serves as a parallel assistant to aid and complement programmers. Its specialization in code production sets it apart from the other models used in the study, making it an extremely useful tool.

Subsequently, all major language models from OpenAI were examined, namely Chat GPT-3.5, Chat GPT-4, and Chat GPT-4o (omni). These models are ranked by ascending power and access level, from the least powerful to the most powerful. OpenAI, as a pioneer in developing large language models, has created some of the market's most popular models to date. The decision to use OpenAI's models was based on their broad recognition and the power of their technology, justifying their inclusion in this evaluation.

Overall, selecting the aforementioned large language models provided an objective assessment of their capabilities in supporting software development, particularly in generating automated code with limited information.

5.1.6 Designing Different Conversations Based on Prior Knowledge

A critical step in conducting the experiment and evaluating the automated test code generated by the large language models was designing and structuring different conversations based on the prior knowledge provided to the system each time.

The study created various interactions with each large language model to test their performance under different levels of prior knowledge. The evaluation was divided into four phases, each adding new information to the previous one, thus varying the amount and quality of knowledge available to the model at each stage:

1. **Phase 1**: In this phase, the large language model received limited prior knowledge, including a basic description of the library system's architecture, general information about expected responses, and all functional scenarios written in Gherkin using the Cucumber tool.
2. **Phase 2** : In this phase, the prior knowledge provided was the same as Phase 1, with the addition of the names of the library system's Domain classes needed for creating the automated tests.
3. **Phase 3** : Phase 3 included the knowledge from Phase 2, augmented with the properties used by each of the provided Domain classes, enabling the model to use these properties in the test generation process.
4. **Phase 4** : In this final phase, the knowledge from Phase 3 was expanded to include method signatures used by the Domain classes. This phase represented the maximum amount of prior knowledge given to the large language models, facilitating the most comprehensive understanding of the system and enabling the creation of the most complex and accurate automated tests.

For each of the aforementioned categories of prior knowledge, different techniques were employed to present this knowledge to the large language models to identify the most efficient communication method. The objective of this approach was to develop a methodology that ensures optimal interaction with the models for generating automated test code.

Specifically, during each interaction with the language models, four distinct techniques were applied to present the knowledge, as described below:

1. **Presenting All Cucumber Scenarios in Single or Multiple Messages:** In this technique, the prior knowledge of Gherkin scenarios related to the system was provided to the language model either in a single consolidated message (prompt) or across multiple individual messages. The model was then tasked with generating the Step Definitions code corresponding to these scenarios.
2. **Commanding the Language Model to Generate or Not Generate Domain Class Code First:** In this technique, after selecting one of the presentation formats mentioned above, an additional instruction was given: either to generate or not to generate the code for the Domain classes deemed necessary by the model. The aim was to assess whether producing the Domain class code first enhanced the quality of the automated test generation or whether skipping this step resulted in inferior outcomes.

The implementation of these techniques aimed to understand how different methods of presenting knowledge impacted the performance of the language models in automated code generation. This effort contributed to developing strategies for improving communication efficiency with the models.

5.2 Evaluation Criteria

The performance evaluation of large language models (LLMs) in generating automated BDD scenario code is based on a set of criteria designed to ensure a systematic and objective analysis of the results. These criteria focus on various aspects of the models' performance, ranging from the accuracy of their responses to the efficiency of message generation.

Specifically, the evaluation criteria include factors such as the number of messages required to obtain a complete response and the percentage of responses deemed acceptable based on the requirements of the BDD scenario.

Table 1 outlines the evaluation criteria, with each row corresponding to a distinct numbered criterion. The table columns provide essential details for each criterion, such as the name, description, unit of measurement, and the formula (where applicable) used to calculate the outcome described by the criterion.

Table 1. Evaluation criteria of large language models.

Criteria Number	Criteria Name	Description	Measurement	Formula
1	Providing features simultaneously	Whether the features are provided to the LLM all together or in multiple messages	Yes/No	-
2	Providing Domain/DAO/Services code from the start	Whether the LLM was instructed to generate the Domain/DAO/Services code first or not	Yes/No	-
3	Required messages for complete Step Definitions	Counting the number of messages required to create all Step Definitions with as much code as possible	Integer Value	-
4	Use and Accuracy of Data Access Objects	Evaluation of the accuracy of the usage and reminders for Data Access Objects by the AI	DAO Use: 1 if used from the start / 0.5 if not used from the start. Integer values	DAO Usage Score = DAO Usage × (Accurate DAO - Required Reminders)
5	Use and accuracy of Services	Evaluation of the acceptance and reminders for Services by the LLM	Service Use: 1 if used from the start / 0.5 if not used from the start. Integer values	Service Usage Score = Service Use × (Accepted Services - Required Reminders)
6	Accuracy of Domain classes guessed/used	Measuring the percentage of Domain classes that were guessed	Percentage value	Domain Class Accuracy = (Number of Correctly Guessed Domain

		or correctly used by the LLMή χρησιμοποιήθηκαν σωστά από το LLM.		Classes / Total Number of Domain Classes) × 100
7	Acceptable Step Definition solutions	Measuring the percentage of Step Definitions that are acceptable based on completion, accuracy, functionality, and integration	Percentage value	Accepted Step Definitions = (Number of Accepted Step Definitions / Total Number of Step Definitions) × 100
8	Better than acceptable Step Definition solutions	Measuring the percentage of Step Definitions that had better than accepted solutions	Percentage value	Better than Accepted Solutions = (Number of Better than Accepted Step Definitions / Total Number of Step Definitions) × 100
9	Replacing objects in natural language with code variables	Describes whether the LLM replaces instances of objects expressed in natural language with code variables	Alphanumeric	-
10	Accuracy of properties	Measuring the percentage of correct properties used by the LLM after being provided (phase 3,4)	Percentage value	Property Accuracy Score = (Correct Properties / (Correct Properties + Ignored Properties)) × 100
11	Accuracy of Methods (excluding Getters and Setters)	Measuring the percentage of correct methods used by the LLM after being provided (phase 4)	Percentage value	Method Accuracy Score = (Correct Methods) / (Correct Methods + Ignored Methods) × 100
12	Repetition/Additional explanation	Counting the instances where the LLM required additional explanation or repeated messages	Integer value	-
13	Empty Step Definitions	Counting the instances where the LLM provided empty Step Definitions with no code	Integer value	-

Here is the detailed explanation of each criterion from **Table 1:**

1. **Providing features simultaneously:** This criterion examines whether the features of the system were provided in a single message or gradually in multiple messages. If the features were presented in one message, the evaluation assigns the value "Yes". Otherwise, if the features were provided across multiple messages, the value "No" is assigned.
2. **Providing Domain/DAOs/Services code from the start:** This criterion evaluates whether the LLM was instructed to generate the code for Domain classes, Data Access Objects (DAOs), and Services right from the beginning. If such an instruction was given, the

evaluation assigns the value "Yes". Otherwise, if no such instruction was given, it assigns the value "No".

3. **Required messages for complete Step Definitions:** This criterion measures the number of messages exchanged to create the Step Definitions code from the LLM. A lower number of messages indicates better efficiency and understanding by the system.
4. **Use and accuracy of Data Access Objects:** This criterion evaluates the LLM's accuracy in using and guessing the correct Data Access Object classes. It is calculated by the formula: "DAO Usage Score = DAO Usage × (Accurate DAOs - Required Reminders)", where "DAO Usage" is 1 if the system used DAOs without additional help, and 0.5 if it required additional messages. "Accurate DAOs" refers to the number of correctly guessed DAOs, and "Required Reminders" counts the number of messages needed to remind the model of the DAOs. A high accuracy score reflects better understanding and use of DAOs by the model.
5. **Use and accuracy of Services:** This criterion evaluates the LLM's accuracy in using and guessing the correct Service classes. The formula is "Service Usage Score = Service Usage × (Accepted Services - Required Reminders)", where "Service Usage" is 1 if the system used Services from the start, and 0.5 if additional help was needed. "Accepted Services" measures the number of acceptable Service classes used by the model, even if they differ from the actual system services. "Required Reminders" counts the messages needed to remind the model to use Services. High accuracy indicates good understanding and use of Services by the model.
6. **Accuracy of Domain classes guessed/used:** This criterion measures the percentage of Domain classes that the LLM guessed or used correctly in generating the Step Definitions code. The formula is: "Domain Class Accuracy = (Number of Correctly Guessed Domain Classes / Total Number of Domain Classes) × 100". A high accuracy percentage indicates that the model successfully guessed and used Domain classes.
7. **Accepted Step Definition solutions:** This criterion measures the percentage of Step Definitions solutions that are acceptable based on their completeness, accuracy, functionality, and integration. The formula is: "Accepted Step Definitions = (Number of Accepted Step Definitions / Total Number of Step Definitions) × 100". A high percentage indicates that the model produces solutions that are acceptable and applicable to the system.

8. **Better than accepted Step Definition solutions:**This criterion measures the percentage of Step Definitions that are better than the accepted solutions or the existing code. The formula is: "Better than Accepted Solutions = (Number of Better than Accepted Step Definitions / Total Number of Step Definitions) × 100". A high percentage indicates that the model produces not only acceptable solutions but also enhanced and efficient ones.
9. **Replacing natural language objects with code variables:** This criterion evaluates the LLM's ability to understand and replace objects described in natural language within Cucumber scenarios with appropriate code variables in the Step Definitions. The model is assessed on how well it converts natural language descriptions into code variables. Correct replacement suggests good understanding of both language and code required for generating Step Definitions.
10. **Accuracy of properties:** This criterion evaluates the LLM's ability to guess the correct properties of all Domain classes after the information has been provided. This criterion is used in phase 3 and phase 4 when the information is provided to the model. It is measured as a percentage and calculated by the formula: "Property Accuracy Score = (Correct Properties / (Correct Properties + Ignored Properties)) × 100". A high score indicates that the model understood the given information and applied it effectively in its implementation.
11. **Accuracy of methods (excluding Getters and Setters):** This criterion measures the LLM's ability to guess the correct methods of Domain classes after the methods have been provided. It is used only in phase 4. The formula is: "Method Accuracy Score = (Correct Methods) / (Correct Methods + Ignored Methods) × 100". A high score indicates that the model correctly used the methods in its implementation.
12. **Repetition/Additional explanation:** This criterion measures the number of messages that needed to be provided as reminders, repetitions, or help for the LLM to continue producing the code. A lower number of repetitions indicates that the system understood the information better.
13. **Empty Step Definitions:** This criterion measures the number of times the LLM produced Step Definitions with little or no code. A low number of empty Step Definitions indicates that the system produced code more effectively and efficiently.

These evaluation criteria provide a comprehensive foundation for analyzing the performance of large language models in the context of generating automated code for BDD scenarios. They cover critical aspects of the model's utility, offering a systematic tool for comparing and evaluating the capabilities of LLMs.

5.3 Empirical Evaluation

As detailed in the methodology section, the evaluation of large language models was conducted in four phases, each representing a different amount of knowledge provided to the models.

5.3.1 Phase 1 Evaluation

Table 2, presented below, shows the results from the **first phase** of evaluating the performance of the large language models in generating automated code for BDD scenarios. The rows of the table represent the different numbered criteria analyzed earlier, while the columns represent the different conversations held with the various large language models.

Table 2. Phase 1 evaluation of Large Language Models

Phase 1															
	GPT -3.5				GitHub COPILOT 3.5 Turbo				GPT -4				GPT -4o		
Criteria Number	Chat 1	Chat 2	Chat 3	Chat 4	Chat 1	Chat 2	Chat 3	Chat 4	Chat 1	Chat 2	Chat 3	Chat 4	Chat 1	Chat 2	Chat 3
1	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No
2	No	Yes	Yes	Yes	No	No	No	Yes	No	Yes	No	Yes	No	Yes	No
3	13	10	11	11	8	7	9	8	9	8	8	8	5	7	5
4	3	1	2	1	0	1	3	3	0.5	3	1	2	2	2	2
5	3	3	2	3	3	3	3	4	3	3	3	4	3	3	4

6	28.57 %	42.85%	28.57%	42.85%	28.57%	42.85 %	42.85 %	42.85 %	28.57%	42.85 %	28.57%	28.57%	28.57%	28.57%	28.57%
7	22.91%	31.25%	20.83%	20.83%	20.83%	33.3%	18.75%	22.91%	18.75%	33.3%	8%	10%	35.4%	20,8%	18%
8	0%	0%	0%	0%	0%	0%	0%	0%	0%	2%	0%	0%	2%	0%	0%
9	The system did not fully utilize the objects expressed in natural language, only in a few Step Definitions.	The system did not fully utilize the objects expressed in natural language, only in a few Step Definitions.	The system did not use any of the objects expressed in natural language.	The system did not use any of the objects expressed in natural language.	The LLM partially understood and used some of the objects expressed in natural language, but not in all Step Definitions.	The LLM understood the objects expressed in natural language in almost all Step Definitions. .	The LLM understood the objects expressed in natural language in nearly all Step Definitions. .	The LLM understood the objects expressed in natural language in nearly all Step Definitions.	The LLM used only a few objects expressed in natural language. .	The system did not use any of the objects expressed in natural language	The system did not use any of the objects expressed in natural language	The system did not use any of the objects expressed in natural language	The LLM partially understood and used some of the objects expressed in natural language, but not in all Step Definitions.	The LLM partially understood and used some of the objects expressed in natural language, but not in all Step Definitions.	The system did not use any of the objects expressed in natural language.
12	4	4	4	4	4	2	3	3	5	2	2	2	0	1	0
13	28	7	15	11	23	11	19	8	10	4	2	2	0	10	0

In this initial evaluation phase, all large language models exhibited limited performance in terms of the amount of generated code, the detail of responses, and the acceptance of the produced Step Definitions. Specifically, during the initial discussions where no explicit instruction was given to create code for Domains, Data Access Objects (DAOs), or Services, the models faced significant challenges in producing code. This resulted in many empty Step Definitions and a need for frequent reminders or commands to generate the required code. The table shows that the number of empty Step Definitions reached up to 28.

This situation improved significantly when we started instructing the models to first generate the code for Domains, DAOs, and Services before proceeding to the Step Definitions. This approach helped the models focus better and utilize the classes they had already created more effectively.

As a result, there was a noticeable reduction in the number of empty Step Definitions and an improvement in the quality of the generated Step Definitions, with many models demonstrating higher acceptance rates or maintaining high approval rates. Furthermore, this technique appeared to enhance the use of DAOs by the models, which had been previously overlooked in many cases. Specifically, GitHub Copilot and GPT-4o stood out as they independently generated the code for Domains, DAOs, and Services without additional prompts, highlighting the beneficial impact of this technique on model performance. These two models also proved particularly adept at understanding and utilizing variables expressed in natural language, which contributed to producing more accurate code.

Next, we tested the method of incremental presentation of requirements, where each feature was given in a separate message to focus the model on each feature individually and improve outcomes. However, observations showed that incremental presentation did not enhance performance compared to other techniques, as the accepted Step Definitions were often fewer, and the models lost the broader system perspective that could lead to more accurate results.

From the first phase, it is evident that GitHub Copilot outperformed GPT-3.5 and GPT-4 in many cases, as it demonstrated better data understanding and faster code generation with fewer empty Step Definitions. GPT-4o, being the most recent and advanced model, proved superior to all others, exhibiting better comprehension of the provided information, a minimal number of messages required for generating Step Definitions, and code production speed that surpassed all other models.

5.3.2 Phase 2 Evaluation

In the **second phase** of the experiment, the same information presentation techniques as in the first phase were employed, as detailed in Table 3 below. However, the results were largely similar or even worse in many cases. Specifically, the acceptance rates for Step Definitions were significantly lower compared to the first phase, despite the fact that the language models had gained more knowledge. This suggests that the inclusion of class names did not substantially improve the outcomes.

In most cases, the models failed to comprehend the relationships between classes and were limited to using only the most basic ones, often overlooking many others. As in the previous phase, the command to generate the code for Domains, Data Access Objects (DAOs), and Services significantly improved the responses of the language models. This technique, in particular, helped the models better understand and produce more acceptable Step Definitions, while also facilitating the correct usage of DAOs for storing and retrieving objects, as evidenced by the improved usage rates of appropriate DAOs.

The technique of presenting requirements in natural language as separate messages resulted in similar issues to those observed in the first phase. Although there were instances of positive outcomes, the overall performance did not validate this approach as the best option. Conversely, presenting requirements in a single message and instructing the models to generate the code for Domains, DAOs, and Services first remained the most effective strategy.

In conclusion, GitHub Copilot continues to outperform GPT-3.5 and GPT-4 by demonstrating a better understanding of variables provided in natural language, such as *George Red*, *Moby Dick*, and *Harry Potter*, and using them correctly in many cases. However, GPT-4o once again confirmed its superiority over the other models, delivering far better results. GPT-4o effectively utilized the additional knowledge from Domain class names and produced a high quantity of high-quality Step Definitions, as reflected in the results for evaluation criterion 8.

Table 3. Phase 2 evaluation of Large Language Models

Phase 2															
	GPT -3.5			GitHub COPILOT 3.5 Turbo					GPT -4				GPT -4o		
Criteria Number	Chat 1	Chat 2	Chat 3	Chat 1	Chat 2	Chat 3	Chat 4	Chat 5	Chat 1	Chat 2	Chat 3	Chat 4	Chat 1	Chat 2	Chat 3
1	Yes	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes
2	Yes	No	No	No	No	Yes	Yes	Yes	No	Yes	No	Yes	No	Yes	Yes
3	10	10	7	9	11	10	8	7	9	9	9	8	6	3	5
4	3	0	0	0	3	0	3	1	3	2	1	1	3	3	3
5	3	2	3	3	3	3	3	2	3	4	3	3	3	1	3
7	6.25%	2.08%	10.4%	20.8%	12.5%	15%	33.3%	20.8%	25%	33.3%	12.5%	31.2%	50%	54.1%	37.5%
8	0%	0%	0%	0%	0%	0%	2%	2%	2%	2%	2%	8.33%	20.83%	20.83%	8.33%
9	The system did not utilize any objects expressed in natural language.	The system did not utilize any objects expressed in natural language.	The system did not utilize any objects expressed in natural language.	The LLM perfectly understood the objects provided in natural language and used them correctly.	The LLM perfectly understood the objects provided in natural language and used them correctly.	The LLM perfectly understood the objects provided in natural language and used them correctly..	The LLM perfectly understood the objects provided in natural language and used them correctly.	The LLM perfectly understood the objects provided in natural language and used them correctly.	The LLM did not understand the objects provided in natural language very well and used them only in rare cases.	The LLM did not understand the objects provided in natural language very well and used them only in rare cases.	The LLM did not understand the objects provided in natural language very well and used them only in rare cases.	The LLM did not understand the objects provided in natural language very well and used them only in rare cases.	The LLM only partially understood the objects provided in natural language, missing many of them.	The LLM only partially understood the objects provided in natural language, missing many of them.	The LLM only partially understood the objects provided in natural language, missing many of them.
12	4	3	1	1	5	2	1	1	2	2	3	2	0	0	0
13	31	26	26	3	15	0	9	5	1	1	1	3	6	4	0

5.3.3 Phase 3 Evaluation

In the **third phase** of the experiment, the knowledge provided to the language models was expanded, as outlined in the methodology section. This time, in addition to the system architecture, general system information, natural language requirements, and Domain class names, we also included the properties of each class. This allowed the models to discover and better understand the relationships between the classes and the methods that should be applied.

The inclusion of these properties, as shown in **Table 4** below, appears to have had a significant impact on the performance of the language models. The models significantly improved in analyzing and using the classes, as evidenced by the increased percentages of acceptable Step Definitions and Step Definitions that went beyond the acceptable solution. The instruction to generate the Domain/Data Access Objects/Services continued to prove useful, with the language models understanding and generating code more efficiently. This is reflected in the increase in successful functions and the reduction of empty Step Definitions.

In this phase, it is observed that the technique of presenting requirements in natural language in separate messages is used less and less. The results from the previous phases do not confirm that this technique provides the best outcomes, leading to conversations focusing more on the other mentioned techniques.

GitHub Copilot shows significant improvement compared to GPT-3.5 and GPT-4, particularly in terms of acceptable Step Definitions, better-than-acceptable Step Definitions, and the understanding of variables given in natural language. On the other hand, GPT-4o continues to stand out, providing complete code with minimal messages, without gaps, and with excellent understanding of the data and properties provided to it.

Furthermore, all four language models show significant progress in understanding the correct Data Access Objects and Services, compared to previous phases, as evidenced by the evaluation criteria 4 and 5.

Table 4. Phase 3 evaluation of Large Language Models

Phase 3														
	GPT -3.5				**GitHub COPILOT 3.5 Turbo**				**GPT -4**			**GPT -4o**		
Criteria Number	Chat 1	Chat 2	Chat 3	Chat 4	Chat 1	Chat 2	Chat 3	Chat 4	Chat 1	Chat 2	Chat 3	Chat 1	Chat 2	Chat 3
1	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
2	No	No	Yes	Yes	No	No(it did it by itself)	Yes	Yes	No	Yes	No	No	Yes	Yes
3	8	11	9	11	6	11	9	9	11	7	11	4	4	3
4	1	0	3	2	2	2	0	3	2	3	1	3	3	3
5	2	0.5	4	3	3	3	1	3	3	4	4	3	3	2
7	6.25%	18.75%	37.5%	35.41%%	54.1%	56.25%	22.91%	20.8%	22.91%	37.5%	20.8%	68.75%	60.41%	56.25%
8	0%	0%	4.1%	4.1%	4.1%	18.75%	14%	2%	4.1%	10.41%	4.1%	22.91%	20.83%	14.58%
9	The system did not use any of the objects expressed in natural language.	The system did not use any of the objects expressed in natural language.	The LLM partially understood the objects provided in natural language.	The system did not use any of the objects expressed in natural language.	The LLM understood the objects given in natural language to some extent but did not use them in all Step Definitions.	The LLM perfectly understood the objects given in natural language and used them correctly.	The LLM understood the objects given in natural language to some extent but did not use them in all Step Definitions.	The LLM perfectly understood the objects given in natural language and used them correctly.	The system did not use any of the objects expressed in natural language.	The LLM understood the objects provided in natural language to some extent, but did not use them in all step definitions.	The LLM understood the objects provided in natural language to some extent, but did not use them in all step definitions.	The LLM perfectly understood the objects given in natural language and used them correctly.	The LLM perfectly understood the objects given in natural language and used them correctly.	The LLM perfectly understood the objects given in natural language and used them correctly.
10	25%	15.6%	31.25%	43.75%	42.85%	59.37%	62,5%	31.25%				71.87%	78.1%	68.75%
12	1	3	2	4	0	4	1	2	2	2	2	0	0	0
13	24	4	7	38	6	12	0	3	1	1	2	0	2	9

5.3.4 Phase 4 Evaluation

In the final phase of the experiment, **phase 4**, as previously mentioned, we expand the knowledge provided to the language models by introducing additional information. In addition to the system architecture, general information, natural language requirements, class names, and the properties of each class, we also include the names of all the methods for each class, including their return types and parameters. This allows the models to have the best possible knowledge to produce efficient and acceptable results in automated tests.

The addition of this information, as shown in **Table 5** below, has a positive impact on the performance of the language models. It was observed that, in almost every discussion with each language model, the Data Access Objects and Services classes were fully understood, which had begun to be noticed in the previous phase, but to a lesser extent. The percentages of accepted Step Definitions and the best of the accepted Step Definitions increased significantly, regardless of how the information was presented, indicating that the addition of these elements was necessary for improving the system's performance.

The instruction to produce the code for the Domain/Data Access Objects/Services first continued to improve performance in some cases, although no significant difference was observed compared to conversations where this instruction was omitted, as the system often produced this code automatically. The system's tendency to generate the Domain/Data Access Objects/Services code first was confirmed again, contributing to the efficient production of automated code.

The technique of presenting requirements in separate messages was omitted in this phase as well, since it was shown to have more drawbacks than benefits.

GitHub Copilot stood out for its ability to better understand every criterion than GPT-3.5 and GPT-4, although it had a smaller percentage of "best" Step Definitions. It still didn't reach the nearly perfect results of GPT-4o. GPT-4o, with the new information, presented excellent results with minimal messages, demonstrating great speed and efficiency, and almost perfect percentages in the use of properties and functions provided to it (criteria 10 and 11).

Overall, the language models GitHub Copilot and GPT-4o were the only ones that largely understood the use of natural language variables and applied them correctly in the automated tests.

Ταβλε 5. Phase 4 evaluation of Large Language Models

Phase 4													
	GPT -3.5				**GitHub COPILOT 3.5 Turbo**			**GPT -4**			**GPT -4o**		
Criteria Number	Chat 1	Chat 2	Chat 3	Chat 4	Chat 1	Chat 2	Chat 3	Chat 1	Chat 2	Chat 3	Chat 1	Chat 2	Chat 3
1	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
2	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	Yes	No(it did it by itself)	No(it did it by itself)
3	14	10	14	14	11	8	8	10	12	8	4	4	3
4	3	3	3	1	3	3	1	3	2	2	3	3	3
5	3	2	3	2	2	3	3	3	3	3	3	4	2
7	10.41%	37.5%	37.5%	33.3%	33.3%	52.08%	43.75%	43.75%	27.07%	31.25%	58.33%	72.91%	64.58%
8	0%	16.6%	12.5%	10.41%	14.5%	12.51%	14.5%	14.5%	4.1%	6.2%	20.83%	31.25%	27.08%
9	The system did not use any of the objects expressed in natural language.	The LLM understood the objects provided in natural language to some extent, but it did not use them correctly in all the Step Definitions.	The system did not use any of the objects expressed in natural language.	The LLM used the objects provided in natural language only a few times, but not well enough to be noteworthy. .	Το LLM κατανόησε τα αντικείμενα που δόθηκαν σε φυσική γλώσσα σε πολλές περιπτώσεις, αλλά όχι σε όλες.	The system did not use any of the objects expressed in natural language.	The LLM understood the objects provided in natural language to some extent, but it did not use them correctly in all the Step Definitions.	The system did not use any of the objects expressed in natural language.	The system did not use any of the objects expressed in natural language.	The LLM understood the objects provided in natural language to some extent, but it did not use them correctly in all the Step Definitions. .	The LLM perfectly understood the objects provided in natural language and used them very effectively.	The LLM perfectly understood the objects provided in natural language and used them very effectively.	The LLM perfectly understood the objects provided in natural language and used them very effectively.
10	18.75%	78.12%	52.12%	46.87%	25%	46.8%	46.8%	46.8%	46.8%	43.75%	90.6%	90.6%	90.6%
11	11%	46.1%	34.6%	34.6%	30.7%	42.3%	30.7%	26.9%	34.6%	34.6%	50%	34.6%	50%
12	5	4	5	6	4	1	2	3	2	1	0	0	0
13	14	1	10	9	20	0	1	2	0	1	6	4	3

6. CONCLUSIONS

In conclusion, the analysis of the use of large language models within the context of BDD demonstrates that these tools can offer numerous advantages in the field of software development, particularly in the creation of automated code testing. Specifically, as explored in this study, large language models have proven to be highly useful in automating the creation of BDD scenarios, improving code quality, and reducing errors. However, the research has also highlighted limitations in the generation of automated code, such as the need for a better understanding of the data.

The results of the study and the experiment conducted clearly show that the application of large language models can significantly improve the process of creating automated code through BDD, while also presenting some limited results that require further improvement.

In the first phase, it was observed that although the language models were capable of generating code, they delivered limited results in terms of quantity, detail, and acceptance of the generated Step Definitions. **Specifically**, many empty Step Definitions were observed, especially when no initial command was given for generating the code for Domain/Data Access Objects/Services. The application of this command helped the models to focus better and improve the results, with GitHub Copilot and GPT-4o standing out for their ability to understand and use variables in natural language.

In the second phase, although the same techniques for presenting information were used, the results did not show significant improvement compared to the first phase. The acceptance rates for Step Definitions were lower, and techniques such as gradual presentation were not proven to be more effective. However, the command for generating the code for Domain/Data Access Objects/Services continued to **improve** the results.

In the third phase, the addition of class properties proved to have a positive impact on the models' performance. The large language models showed improved understanding and use of classes, leading to higher rates of accepted Step Definitions. However, the technique of presenting requirements in separate messages remained less effective.

In the final phase, the addition of method names, return types, and class parameters had a positive effect on the models' performance. GitHub Copilot and GPT-4o once again stood out for their ability to understand and use variables in natural language, with GPT-4o achieving almost perfect results with high speed and efficiency.

Overall, the language models GitHub Copilot and GPT-4o emerged as the most effective in generating code for BDD scenarios, with GPT-4o presenting the best results. The addition of extra information, such as class properties and method names, improved performance, while the technique of generating the domain/Data Access Objects/Services code first appears to be the most effective communication technique with large language models for code generation.

7. BIBLIOGRAPHY

Cucumber. (χ.χ.). *Gherkin Reference*. Retrieved from Cucumber documentation: https://cucumber.io/docs/gherkin/reference/

Fitzgibbons, L. (2021, October). *behavior-driven development (BDD).* Retrieved from TechTarget: https://www.techtarget.com/searchsoftwarequality/definition/Behavior-driven-development-BDD

Hurani, M., & Idris, H. (2024). *Investigating the use of LLMs for.* Karlskrona, Sweeden.

Lee, T. B. (2023, July 31). Retrieved from arstechnica.com: https://arstechnica.com/science/2023/07/a-jargon-free-explanation-of-how-ai-large-language-models-work/6/

Lutkevich, B. (2024, June 21). *TechTarget.* Retrieved from https://www.techtarget.com/whatis/feature/12-of-the-best-large-language-models

Ozkaya, I. C. (2023, October 2). *Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change?.* Ανάκτηση από https://doi.org/10.58012/6n1p-pw64.

Pathak, K. (2022, September 12). *Cucumber Best Practices to follow for efficient BDD Testing*. Retrieved from Medium: https://kailash-pathak.medium.com/cucumber-best-practices-to-follow-for-efficient-bdd-testing-b3eb1c7e9757

Pi, W. (2024, May 7). *Research Graph.* Retrieved from Medium: https://medium.com/@researchgraph/brief-introduction-to-the-history-of-large-language-models-llms-3c2efa517112

Sanderson, G. (2017, October 16). *3blue1brown.* Retrieved from 3blue1brown: https://www.3blue1brown.com/lessons/gradient-descent#another-way-to-think-about-the-gradient

Sumrak, J. (2024, March 11). *7 LLM use cases and applications in 2024.* Retrieved from AssemblyAI: https://www.assemblyai.com/blog/llm-use-cases/

Xinyi Hou, Y. Z. (2023). *Large Language Models for Software Engineering: A Systematic Literature Review.* Retrieved from https://ar5iv.labs.arxiv.org/html/2308.10620

Zharovskikh, A. (2023, June 22). *Best applications of large language models*.

Ανδρουτσόπουλος, Ί. (2024, February). Τεχνητή Νοημοσύνη και Μεγάλα Γλωσσικά Μοντέλα. *ΟΠΑ News Εφημερίδα Οικονομικού Πανεπιστημίου Αθηνών Τεύχος 51*, σσ. 8-9.

8. APPENDICES

8.1 Phase 1

8.1.1 GPT -3.5 Chat 1

Description:

In this initial conversation with GPT-3.5, I provided the LLM with the system description/architecture and all system features in a single message. I then instructed it to generate the code. Initially, the LLM did not fully understand the task and only created Step Definitions with empty bodies. After some guidance through messages, it began generating slightly more code, but in many cases, it mentioned that I would need to implement the system logic as a comment. Later, when I instructed it to provide the system logic (such as DAOs and domain), it did a fairly good job, especially for the Data Access Objects. As the conversation progressed, the LLM understood the instructions better and was more willing to generate code for the step definitions.(link for full chat).

8.1.2 GPT -3.5 Chat 2

Description:

In this conversation with GPT-3.5, I provided the LLM with the knowledge of the system description/architecture and all system features in a single command. I then asked the LLM to generate the code, initially for the domain classes and later for the DAOs and Services. With this slight variation, the system immediately began generating code without providing empty step definitions, in contrast to the first conversation where it didn't generate any code at the beginning. The LLM provided more information after each response and did an excellent job assessing the Domain and DAO classes based on the scenarios. Overall, the instruction to first generate the code for the domain classes and then proceed with the creation of the step definitions had a positive impact on the LLM's understanding of the system. (link for full chat).

8.1.3 GPT -3.5 Chat 3

Description:

In this conversation, I attempted to provide the same information as always, presenting the features one by one, which seems to help the LLM stay focused on each specific feature and try to infer the code for the remaining characteristics. However, I slightly altered the initial instruction to see if there would be significant changes, providing a more detailed explanation of the process I wanted the LLM to follow. The LLM produced good results, correctly inferring some domain classes from the start, along with the DAOs, and immediately began implementing the code. This is consistent with all the conversations where I first asked for the code for the domain classes. Unlike other conversations, where the LLM struggled to generate the code for the step definitions without further explanations, this approach had a positive impact. Overall, the results were good for all the features. The LLM correctly guessed some classes, and very little explanation of the instruction was required for it to understand what needed to be done.(link for full chat).

8.1.4 GPT -3.5 Chat 4

Description:

In this conversation, I attempted to provide the LLM one feature at a time, then requested the creation of the code for the domain classes and, subsequently, for the step definitions, for each feature separately. The goal was to explore whether providing the information in additional steps (by adding the domain code) would help the system generate better code with fewer instructions and less guidance. The LLM understood the task well and created the code for the domain classes with the DAOs, as requested, and then developed the code for the step definitions without further assistance. This meant that fewer instructions were needed to complete the task. Additionally, the system seems to better understand the persona I created ("George Red"), and for the first time, it used this persona directly, rather than as the variable {string}, as it had done in previous conversations. Overall, the results were moderate and similar to those from other conversations. I found that providing the features one by one doesn't seem to be particularly helpful, as one feature in the last step may contain valuable information for a scenario in the first feature, which the LLM isn't always aware of. Nevertheless, the instructions are shorter, and the system seems to remember the last feature given more easily.(link for full chat).

8.1.5 GitHub Copilot Chat 1

Description:

In my first interaction with GitHub Copilot, I didn't need to provide the system's features, as it could automatically read the files from the editor. Instead, I provided the LLM with the architecture and general knowledge of the system, as well as the system's features. Initially, the LLM only produced the Step Definitions with empty bodies, but after a few messages, it understood the task and began to generate code with ease. For the first time, with the limited knowledge provided to it, the LLM was able to understand the significance of "personas" (e.g., George Red) and used the variables Harry Potter and Moby Dick correctly. However, like ChatGPT, the LLM often used assertions in the GIVEN steps, which is not the best practice for writing Step Definitions, as the GIVEN steps should create the environment and variables for scenario verification. Overall, the LLM grasped the instructions immediately and required only a few messages to generate the code, unlike ChatGPT. The generated code wasn't perfect, but it was quite good in many areas. The LLM made good use of the Service classes, but not the Data Access Objects, and missed some important Step Definitions due to their complexity and the lack of prior knowledge given to it.(link for full chat).

8.1.6 GitHub Copilot Chat 2

Description:

In this conversation, I provided the same information as in previous interactions, with slight variations in the initial instruction. However, instead of using GitHub Copilot's reference system, I included the system's features as a message. Initially, the LLM began generating the necessary domain classes, DAOs, and Services without being explicitly asked, something it had never done with ChatGPT. This is a positive sign, as it indicates the LLM proactively considered domain classes first before proceeding with the requested code generation later.

The domain classes were satisfactory. Subsequently, the LLM began generating the step definition code from the start without requiring extensive instructions. The code was similar to other cases, but the system utilized more Services, which is highly beneficial, and understood the need to use DAOs (e.g., creating BorrowerService classes that leverage DAOs).

Overall, the LLM demonstrated a strong understanding of what needed to be done. The generation of domain code first appears to facilitate more effective code creation. Further testing will determine whether the reference system or command-based input is the better approach. However, based on this conversation, commands seem to be the better choice. (link for full chat).

8.1.7 GitHub Copilot Chat 3

Description:

This conversation was an experiment to determine if the LLM would once again provide the Domain classes as a starting point without explicit prompting, which it ultimately did. I then continued the discussion to obtain the complete code for the step definitions. The results were overall good and quite similar to those of previous conversations, based on the knowledge I provided to the LLM. The system began generating code immediately, requiring very few commands. (link for full chat).

8.1.8 GitHub Copilot Chat 4

Description:

In this conversation with GitHub Copilot, I provided the LLM with the system's architecture, general knowledge, and features in a single command, similar to previous interactions. The goal was to see if the LLM would generate the same domain classes as it did initially when I provided the features through GitHub's reference system. The results were quite similar, making this conversation less interesting overall. (link for full chat).

8.1.9 GPT -4 Chat 1

Description:

In my first interaction with GPT-4, I provided the LLM with all the system's features and architecture simultaneously. The LLM required several commands to fully understand my expectations, and I had to repeatedly emphasize the need to generate complete code and step definitions for the features. Overall, the results were not very satisfactory. The generated code was often overly simplistic, and the LLM needed many explicit instructions to understand what it was supposed to create. Additionally, it did not autonomously utilize DAOs and required clear guidance to incorporate them.(link for full chat).

8.1.10 GPT -4 Chat 2

Description:

In this interaction, I provided the LLM with all the system's features and its architecture, and I also requested it to initially generate the code for Domains, DAOs, and Services before proceeding to the step definitions. The LLM understood the task and created the requested code. However, it required repeated commands to produce all step definitions for the features, leveraging the architecture I had provided at the start. Overall, the results were acceptable for some step definitions, but many were overly simplistic and did not match the complexity or quality expected for a real-world system.(link for full chat).

8.1.11 GPT -4 Chat 3

Description:

In this conversation, I provided the LLM with the system's features one by one, instead of all at once, as well as the architecture. This approach resulted in the LLM retaining the instructions better, as each feature was presented individually. However, the LLM lacked awareness of later features when implementing the initial step definitions. Overall, the LLM understood the task reasonably well but still required repeated commands to generate the complete step definitions for the features, similar to all other cases.(lnik for full chat).

8.1.12 GPT -4 Chat 4

Description:

In this conversation, I provided the LLM with all the system's features one by one, along with the architecture. Before each feature, I asked the LLM to first generate the code for the domain classes and then proceed to create the code for the step definitions. As in all other conversations, the LLM required repeated instructions to fully generate the step definitions. The results were good in some cases but not exceptional. The LLM seemed to understand the task better, although it lacked critical knowledge of subsequent features, which could have assisted in implementing the earlier ones. (link for full chat).

8.1.13 GPT -4o Chat 1

Desciption:

In the first conversation with GPT-4, I provided the same information I had shared in other Phase 1 interactions with various LLMs (features and a brief system description) but without additional instructions for creating the Domain code or other components, as I had done in other cases. The results were quite satisfactory given the limited amount of information provided.

The LLM immediately understood the task and easily created the first feature. Notably, the LLM did not omit any scenario and demonstrated a unique characteristic compared to previous conversations: for each response related to a different feature, it remembered the initial instruction outlining the architecture (Domain, DAOs, Services) and adapted accordingly. It generated code for DAOs, Services, and the Domain, making necessary adjustments based on the new information provided.

Moreover, for the first time, the LLM also provided the dependencies needed to integrate Cucumber into the system, which had not occurred in previous conversations. Overall, the results were quite good for an initial interaction. While the LLM used assertions in the GIVEN steps—which might not be ideal—it quickly grasped the requirements and produced the necessary code with minimal guidance, effectively utilizing DAOs, Domain classes, and Services(link for full chat).

8.1.14 GPT -4o Chat 2

Description:

In this conversation, I provided exactly the same information as in Conversation 1, but added the instruction to first generate the code for the Domain, Services, and DAOs classes. The goal was to see if it would affect the responses of the LLM. GPT-4 understood the task extremely well from the start and, as in the first conversation, provided the Domain, DAOs, and Services classes in detail. While some features in the Domain classes were correct and others were not, the LLM described all the classes and methods in detail. After the creation of the classes, the LLM easily created the features without many additional instructions. Overall, the results were similar to those in Conversation 1, with the LLM also providing the architecture that should be followed in your file structure. The additional instruction to create the Domain/Services/DAOs classes first did not seem to have a significant impact on the results. GPT-4 continued to produce high-quality results, effectively using the DAOs, Domain classes, and Services as required. The answers were once again useful and correct, showing that the LLM handled the architecture and system requirements well. (link for full chat).

8.1.15 GPT -4o Chat 3

Description:

In this conversation, the approach of providing the features one by one was used, as with other LLMs. The goal of this technique was to see if the LLM would focus more on each individual feature and improve the quality of the code. Overall, the results were almost the same as those obtained previously or, in some cases, even worse. The LLM seemed to quickly understand the features and generate the answers immediately, but the approach of providing the features one by one did not have a positive effect on the quality of the generated code. Providing all the information from the start could have improved the code quality, as the LLM would have had a complete view of the system from the beginning. The technique of providing features one by one did not bring the desired improvement and will therefore not be used in future conversations, as it did not have a positive impact on producing better code. (link for full chat).

8.2 Phase 2

8.2.1 GPT -3.5 Chat 1

Description:

In this phase 2 conversation, I provided the LLM with the system's features in Gherkin format and the names of the domain classes, asking it to use Data Access Objects (DAOs) and Services (without providing the classes for these). The instruction included all the features in a single message, and I asked the LLM to first generate the code for the domain classes, DAOs, and Services before creating the code for the step definitions. The results were impressive. The LLM was able to guess the code for the classes with considerable accuracy. Specifically, its guess for the `ItemState` class as an Enumeration was accurate, as confirmed by the actual code. For the DAOs, the LLM created more classes than necessary but showed a good understanding of the required functions. It also did a good job with certain services, such as `NotificationService` and `LoanService`, although it also included some extra services. Overall, the generated code was simple and executable, though it didn't always match the actual back-end of the system. The instruction to generate the code for the domain classes, DAOs, and Services together seemed to help the LLM integrate these elements more effectively into its solutions. However, the LLM left many step definitions empty or provided comments for me to implement the logic, which suggests it did not have a full understanding of the requirements from the start.(link for full chat).

8.2.2 GPT -3.5 Chat 2

Description:

In this conversation, I provided the same information as in Phase 2, Conversation 1 (features and domain classes), but I did not ask the LLM to first generate or guess the code for the domain classes. Instead, I asked it to directly create the step definitions. Initially, the LLM did not generate actual code and only provided the names of the step definition classes. With some additional help and instructions, the LLM eventually provided code for the step definitions, but it also included comments explaining what needed to be done. One significant issue was that the LLM consolidated all the system logic into a class named libraryService, which was not the best coding practice. This shows that the LLM preferred to use a single class to avoid system complexity, which does not follow best practices for separation of concerns.

Compared to Phase 2, Conversation 1, where I asked for the domain class code first and then the step definitions, this approach yielded better results. In the previous conversation, the LLM successfully guessed many domain classes and generated more consistent code, such as the Enum class for ItemState. Ultimately, when I asked the LLM to provide the domain class code after completing the code for all the step definitions, the results were less satisfactory compared to the first phase, where the LLM had a better understanding of the requirements and created more accurate domain classes. This suggests that integrating the domain classes at the beginning of the process can help the LLM better understand and generate more accurate and organized code for the step definitions. (link for full chat).

8.2.3 GPT -3.5 Chat 3

Description:

In this conversation, I provided the LLM with the features and domain classes of the system, but I presented the features one by one instead of giving them all at once in one command. I asked the LLM to first generate the code for the domain classes and then the code for the step definitions for each feature separately. The LLM seemed to understand the task very well and started generating code from the first message. The results were satisfactory, but the main issue was that, because it didn’t have all the information for the features from the start, it didn't create some classes that were clearly needed if it had read the last feature. These classes would have been very useful for the first feature, resulting in imperfect code. Overall, however, the approach of providing small, guiding messages to the LLM, rather than asking it to do everything in one command, seems to be somewhat better at times.(link for full chat).

8.2.4 GitHub Copilot Chat 1

Description:

In this conversation, I provided the system features as a command along with the names of the domain classes that it should use. This time, the LLM did not generate the code for the domain classes, as was done in Phase 1, possibly because it already had the names of the domain classes. Instead, it immediately started generating code (without asking for the domain classes to be created first). The code produced was quite simple in many cases (e.g., assertions in the GIVEN steps, etc.). The LLM used the service classes well, adding some additional ones, but the result was similar to GPT-3.5. However, the LLM completely omitted the use of DAOs and only used the domain classes I provided, neglecting many classes such as `Book` and not integrating them at all. Overall, the LLM generated code very easily and required almost no instructions, but the code was simple. This shows that providing only the names of the domain classes doesn't really help the Copilot generate more complex code (the same issue occurred with GPT-3.5). However, the LLM understood the task very well from the start.(link full chat).

8.2.5 GitHub Copilot Chat 2

Description:

In this conversation, I provided the Copilot with exactly the same information as in Conversation 1, but I tried to provide the features as "references" (using GitHub Copilot's reference system, where you provide the feature files). The results were almost the same, although the LLM did not generate code for some scenarios, even when I pressed it to do so, and gave me comments to implement the logic myself. Going forward, I will use the command-based technique for providing information, as it seems to be more effective.(link full chat).

8.2.6 GitHub Copilot Chat 3

Description:

In this conversation, I provided Copilot with the same information as in the previous conversations, but asked the LLM to first create the domain classes, as I did in phase 2 of GPT-3.5. The results for the domain classes were the best across all phase 2 conversations (for both LLMs). The LLM understood very well the properties that the classes should have. For example, it added the Book as a feature to the Item class and correctly guessed the Enum for ItemState, along with some other great features. However, the code for the step definitions was quite similar to previous conversations, and the LLM again forgot to use DAOs. Despite these points, the LLM understood the persona of George Red and item names like Harry Potter, Animal Kingdom, etc., just like in all the other conversations.(link for full chat).

8.2.7 GitHub Copilot Chat 4

Description:

In this conversation, I provided the LLM with the same information as in Phase 2, Conversation 3, but also requested the creation of code for the domain classes, as well as the DAOs and Services, so the system would actually use DAOs. As expected, the LLM did an excellent job in creating the domain classes, as well as the DAOs and Services. It was the first time the LLM created an interface for the DAOs. Then, when I asked the LLM to generate the code, I found that it was almost a copy-paste from previous conversations 1, 2, and 3. Overall, instructing the LLM to also generate the code for the DAOs seems to help in their effective use in later steps. (link for full chat).

8.2.8 GitHub Copilot Chat 5

Description:

In this conversation, I provided the LLM with the same information as in all previous conversations, but this time I delivered the features one by one, rather than all at once. For each feature, I requested that the system provide code for the domain classes, DAOs, and Services. The LLM understood the task quite well, did exactly what I asked, and for each new feature, it recreated the domain classes, DAOs, and Services, enhanced with the new properties that resulted from the feature. It also provided fairly good code for the Services, and for the latest features, where the LLM had the most information, it generated very good code. The only issue was that if I had provided all the features upfront, the LLM could have used some of the new properties in its earlier answers for the step definitions. However, the strategy of providing the features one by one wasn't bad, and this approach might have its own advantages.(link for full chat).

8.2.9 GPT -4 Chat 1

Description:

In this conversation, I provided the LLM with all the system's features, domain class names, as well as the architecture and general information about the system. The LLM understood the task well with very few commands, but it required some repetition to provide the full implementation code for the step definitions. Overall, the LLM followed the architecture I provided and created good responses for some scenarios. However, the code was also overly simplified in many other cases. (link for full chat).

8.2.10 GPT -4 Chat 2

Description:

In this conversation, I provided the LLM with all the system's features, domain class names to use, as well as the architecture and general information about the system. The difference in this case was that I first asked the LLM to think and create the code for the domain classes. The LLM estimated the relationships between the classes and some features quite well. It started creating code from the beginning, which shows that it understood very well what I had provided. However, some repetition of commands was still required to create all the step definitions, as it missed a few. Overall, it created fairly complex code, which was good in some cases but poor in others because it was too complex to easily understand the step definition.(link for full chat).

8.2.11 GPT -4 Chat 3

Description:

In this conversation, I provided the LLM with all the system's features one by one, the domain class names to use, as well as the architecture and general information about the system. The LLM understood the task quite well, although some repetitions were still needed to generate the step definitions for all the features, and it produced useful code in many cases. However, the problem with this technique is that it would have been helpful if the LLM had known some of the features provided later earlier, as this could have helped improve its responses and the implementation of the features. Overall, it wasn't a bad conversation. (link for full chat).

8.2.12 GPT -4 Chat 4

Description:

In this conversation, I provided the LLM with all the system features one by one, along with the domain class names to use, the architecture, and general information about the system. The difference this time was that, after each feature was provided, I asked the LLM to generate the code for the domain classes, DAOs, and services. The LLM understood this step very well and didn't forget it, creating each feature using the same process. However, some repetition and reminders were needed to complete all the step definitions for each feature. Overall, it used the architecture quite well and guessed many properties and relationships between the classes. Despite the good use of the architecture, the step definitions were not all excellent. Some were too simplistic, while others were acceptable. (link for full chat).

8.2.13 GPT -4o Chat 1

Description:

In this conversation, I provided the LLM with the system's features, its description, and the names of the required domain classes. The LLM understood the task perfectly, just like in the previous phase, and began generating code for the domain classes (without prior instruction to create the domain code), with attributes that it guessed very well, almost exactly like the actual domain class characteristics. It also immediately generated the code for the features with very few instructions, as it had a clear understanding of what needed to be done. Overall, the code was much better compared to the previous phase, with the LLM producing code with many details and a good understanding of the system and the step definitions. In conclusion, the additional knowledge of the domain class names had a significant positive impact on the result.(link for full chat).

8.2.14 GPT -4o Chat 2

Description:

In this conversation, I provided the LLM with exactly the same information as in Conversation 1, but I intended to ask it to first create the domain code (something it had done automatically in previous cases). However, the LLM created the domain code without needing the features, using only the domain class names. I then provided the features and asked it to generate the updated domain code.

Overall, the domain class code was close to the system's code, but it included many additional attributes. The LLM used only one service, merging all the service classes into it (instead of creating multiple services, which would have been more appropriate). Impressively, for the first time, the LLM created code for all the step definitions of the features simultaneously (in one response) without missing anything, which was excellent and very time-efficient.

Overall, the code provided wasn't better than the code from Conversation 1, but it included many more details in its implementation and showed a deep understanding of the code and the requirements. With more information, even better results might be produced.(link for full chat).

8.2.15 GPT -4o Chat 3

Description:

In this conversation, I provided the same information as in Conversation 2 and asked the LLM to first create the domain code, which it did very easily. The LLM understood the task extremely well and generated code for the Domain, Services, and DAOs, then proceeded to create the step definitions for ALL the features with only TWO messages. This demonstrates that the LLM has strong memory to recall the step definitions for each feature.

The results of the code were generally the same as in previous conversations, with many details, although in many cases, it wasn't as close to the actual code. (link for full chat).

8.3 Phase 3

8.3.1 GPT -3.5 Chat 1

Description:

In this first conversation of phase 3, I provided the LLM with the system's features and the domain classes along with their properties. I asked the LLM to provide the implementation code for the step definitions. Initially, the LLM did not immediately provide the code but only gave the function names for the step definitions, with a comment stating that I needed to implement the code. After a few commands, it began creating the code.

Overall, the system successfully used DAOs for storing, deleting, and accessing objects (whereas, in other conversations, it used Services for implementing this logic). The code was quite simplified in many cases, but it wasn't incorrect. The main issue was that the LLM continuously used assertions in the GIVEN steps, while it should have set up the necessary preconditions to run the scenarios. By using assertions, it assumes the conditions were already completed somewhere else.

In the final step, where I asked for the system's logic to be created, the LLM generated a library class that stores entities. This contrasts with the use of DAOs that it had adopted earlier. Additionally, it did not use the Item class and mostly relied on the Book class because it didn't understand the relationship between them.(link for full chat).

8.3.2 GPT -3.5 Chat 2

Description:

In this conversation, I provided the LLM with exactly the same information as in Phase 3, Conversation 1, but with a few changes. Instead of using the term "fields," I used "attributes," which did not affect the outcome. The key difference, however, was that I asked the LLM to create the domain classes first. It is notable that the LLM understood the task very well. It provided all the classes with the attributes exactly as I had given them and created some "helper" classes such as "TelephoneNumber," "Address," "Money," "Currency," and "SystemDate" based on the attributes I had provided. Although I had not specifically mentioned them, the LLM understood the need for these classes and created them correctly. Specifically, it generated the Enum for the ItemState and correctly inferred the "currency" variable that was needed in the "Money" class without explicit instructions.

However, the LLM did not use any Services or DAOs (I will provide instructions for these in the next conversation). Instead, it used a class named `LibrarySystem` for storing, deleting, and accessing entities, as well as for using methods similar to those in Services. Although the results were correct based on the system the LLM considered suitable, it should have used DAOs and Services as I had requested. Additionally, it did not use the Item class at all, as in Phase 3, Conversation 1, and only used the Book class. (link for full chat).

8.3.3 GPT -3.5 Chat 3

Description:

Σ' αυτήν τη In this conversation, I provided the LLM with the same information as in the other Phase 3 conversations. However, I asked it to create both the Services and DAOs code along with the domain classes (in previous conversations, I only requested the domain classes code). This was something I had also done in Phase 2, Conversation 1, and Phase 2, Conversation 4.

The results were very good for both the domain classes and DAOs. However, for the Services, the LLM created a few more classes than were necessary, although these extra classes were still good. Then, the LLM immediately began generating the code (as it usually does when I ask it to create the domain classes code first), and the results were quite good compared to the previous conversations.

Specifically, the LLM understood that "George Red" is a persona and did not treat it as a regular expression; instead, it created a variable with that name. It also used the Item class as needed (this time, it forgot the Book class, whereas in other conversations it had forgotten the Item class), and it utilized the EmailService class for the notification delay functionality.

Overall, the results of this conversation were very good and useful based on the current code, with only a few minor changes. It seems that reminding the LLM to create the code for DAOs and Services helps it use them more effectively, compared to when nothing is mentioned (in other conversations, I simply asked the LLM to include them in the architecture in the first instruction).) (link for full chat).

8.3.4 GPT -3.5 Chat 4

Description:

In this conversation, I provided the same exact information as in all the previous ones, but I tried to give the features one by one. For each feature, I asked the LLM to provide code for the domain classes as well as for the DAOs/Services. The approach of providing the features one at a time seems to help the LLM focus more on each specific feature and generate the Services classes much better than in previous conversations. The domain and DAO classes are exactly the same, which is a positive outcome.

However, the LLM didn't generate as much code as in the other conversations, primarily because it is waiting for the remaining features to give it further cues. As a result, it mentions in the comments that the logic for the feature needs to be implemented, whereas in the other conversations, it provided code for everything. Overall, the code in many cases was the same as in other conversations and sometimes even better, but I am not sure if this method always leads to better code.

While there are advantages, such as better focus on each feature, there are also drawbacks, such as the lack of a complete implementation for each feature. Additionally, this method might not always be the best, as a scenario described in the last feature could influence the correct implementation of a scenario in the first feature.(link for full chat).

8.3.5 GitHub Copilot Chat 1

Description:

In this conversation, I provided the LLM with the system features and the domain classes with their properties. The Copilot understood the task very well and immediately started generating code, without me explicitly asking it to create the domain class code. Then, with a few additional small commands, it quickly created all the step definitions.

Overall, the LLM made good use of many of the properties of the classes I provided and understood much better how to utilize them compared to GPT-3.5. It also used the Services and DAOs, although the results weren't significantly better than the conversations in Phase 2, mainly due to the use of too many assertions in the GIVEN steps. In the next conversation, I will ask the LLM not to use assertions in the GIVEN steps and we'll see the results. That said, many of the step definitions were very useful.(link for full chat).

8.3.6 GitHub Copilot Chat 2

Description:

In this conversation of phase 3, I provided exactly the same information and initial instructions as in Conversation 1, but I asked the LLM not to use assertions in the GIVEN steps and explained the reason. The Copilot understood this very well, automatically created the domain classes (which seems to help its memory later on), and produced excellent results for the step definitions—possibly the best we've seen.

The LLM made excellent use of the properties I provided, especially with the Item/Book class. In previous conversations, it hadn't used the Book class at all, but now it remembered its connection to the Item class and created very good relationships. Moreover, it understood the persona of George Red and the creation of dummy items very well, just as it had done in many previous phases.

Overall, the generated code was excellent: simple but effective in many cases, and quite close to the real-world implementation. The LLM used some different Services, but this was acceptable and did not negatively affect the quality of the generated code. (link for full chat).

8.3.7 GitHub Copilot Chat 3

Description:

In this conversation, I provided Copilot with the same information as in conversations 2 and 3, but I asked it to create the code for the domain classes first (while in previous conversations I hadn't given such an instruction). The LLM understood quite well which domain classes to create. However, it did not use DAOs or Services and instead used a class called `LibrarySystem` for everything, which did not utilize DAOs.

Overall, the results were good, but the use of this `LibrarySystem` class was not ideal. In the next conversation, I will ask the LLM to create the code for the DAOs and Services from the start to see if this helps the system use them more, similar to the positive outcomes seen with GPT-3.5.(link for full chat).

8.3.8 GitHub Copilot Chat 4

Description:

In this conversation, I provided the LLM with the same information as in all previous conversations, but asked it to give me the code for the DAOs and Services along with the code for the domain classes. As expected, the LLM created fairly good code for the domain classes, as well as for the Services and DAOs. However, the code it produced was not of the highest quality and had some issues, leading to the creation of less code than desired. Overall, this conversation was used to test whether the LLM would use more Services and DAOs, which it did, but we did not focus as much on the quality of the code itself. (link for full chat).

8.3.9 GPT -4 Chat 1

Description:

In this conversation, I provided the LLM with all the features of the system, the names of the domain classes it would need to use, as well as the system's architecture and general knowledge. The LLM immediately understood the task and created the features with minimal guidance. However, it initially forgot to use the DAOs, despite the explicit instructions I had given, and only started using them after I mentioned it again. Overall, the LLM used the provided attributes correctly but presented some unrealistic or overly complicated answers for certain step-by-step definitions. This could affect the readability and ease of modification of the steps. (link for full chat).

8.3.10 GPT -4 Chat 2

Description:

In this conversation, I provided the LLM with all the features of the system, the names of the domain classes it would need to use, as well as the system's architecture and general knowledge. The LLM immediately understood the task and created the features with minimal guidance. However, it initially forgot to use the DAOs, despite the explicit instructions I had given, and only started using them after I mentioned it again. Overall, the LLM used the provided attributes correctly but presented some unrealistic or overly complicated answers for certain step-by-step definitions. This could affect the readability and ease of modification of the steps.(link for full chat).

8.3.11 GPT -4 Chat 3

Description:

In this conversation, I provided the LLM with the system's features, the names of the domain classes and their properties, as well as the architecture and general knowledge of the system. The LLM understood the task quite well, created code with very little guidance, and provided simple but good answers. However, as in other conversations where I used this technique, the LLM would have benefited more if it had all the information at once instead of receiving it gradually. Many features are interconnected and provide valuable insights that could improve the accuracy and consistency of the generated code. (link for full chat).

8.3.12 GPT -4o Chat 1

Description:

In this conversation, I provided the LLM with the system's features, the domain classes of the system along with their properties, and the system's description. As with every other phase with this LLM, it understood incredibly quickly what needed to be done and how to link the classes. It created all the code for the domain classes, services, and DAOs without needing guidance, and then, with just two messages, it generated all the step definitions for all the features. Overall, the code was quite good, often very close to the actual system's code, and included many details in each step, without resorting to simple statements or prints in the step definitions.(link for full chat).

8.3.13 GPT -4o Chat 2

Description:

In this conversation, I provided the LLM with the same information as in Conversation 1, with the slight difference that I instructed it to first generate the code for the Domain, Services, and DAOs, and then create the code for the Step Definitions. Once again, the LLM understood the task well, created good code for the Domain, Services, and DAOs with many details, and in just 2-3 messages, it generated all the Step Definitions without missing any. This is very important for the user in terms of time efficiency. The results were quite good with these specific details and very useful in many cases. (link for full chat).

8.3.14 GPT -4o Chat 3

Description:

In this conversation, I provided the LLM with exactly the same information as in Conversations 1 and 2 of this phase, but I delivered all the information in just 2 messages instead of splitting them. The LLM managed to handle all the information with ease. The results were similar to those of Conversations 1 and 2, with minor differences in some areas. (link for full chat).

8.4 Phase 4

8.4.1 GPT -3.5 Chat 1

Description:

In this conversation of phase 4, I provided the LLM with all the domain classes, their properties, the function names, and the system features. Initially, I asked the LLM to provide the code for the domain classes, DAOs, and services. The LLM responded well to my request, providing each class along with the properties and functions as I had provided. It did a fairly good job with the DAOs and somewhat good with the services. However, overall, the LLM used very few of the functions I had provided, and, as in previous conversations, it followed its own way of writing the code. While it wasn't incorrect, it was quite simplified in many cases, which, in a real system, could lead to errors. Therefore, it seems that the LLM may struggle to remember a lot of information about all the features and functions. While the properties certainly helped the LLM, the functions did not seem to significantly affect the quality of the generated code. (link for full chat).

8.4.2 GPT -3.5 Chat 2

Description:

In this conversation, I provided the LLM with exactly the same information as in Phase 4, Conversation 1 (features, domain classes with properties and methods, and the architecture), but I replaced the word "functions" with "methods." Initially, I asked the LLM to provide code for the domain classes, DAOs, and services, which it did very well, as in all the other conversations. Then, I specifically asked the LLM to remember and use the methods/properties I had provided for each class. The results were much better this time. The LLM truly understood the task, didn't miss any of the step definitions (although in some cases, it gave very simple answers), and used many of the methods I had provided, such as for counting pending items. It also almost correctly guessed and provided useful/executable results for many of the step definitions, like "@Given('^George Red has (\\d+) pending items to be returned')". Overall, the results of this conversation were excellent.(link for full chat).

8.4.3 GPT -3.5 Chat 3

Description:

This conversation was not as important. I provided exactly the same information at every point as in Phase 4, Conversation 2, just to check if with the same or similar instructions, the LLM would produce results as good as those in Conversation 2. In many cases, the LLM provided results that were quite good, although not exactly as good as in Conversation 2. This suggests that the level of results may have been affected by random factors in the previous conversation.(link for full chat).

8.4.4 GPT -3.5 Chat 4

Description:

In this conversation, I provided the LLM with the same information as in all the previous ones, but I gave the features one by one (as I do in each phase). The code for the domain classes, etc., was almost the same as in all the other conversations, but the code for the step definitions was not as good compared to previous conversations (it used a library class instead of the DAOs). My conclusion is that, since there are a few more instructions with the methods, the LLM seems to forget the DAOs when they were mentioned many messages earlier. When I give the last features, it seems to have forgotten them and uses different classes. This does not happen if I provide all the features at once from the beginning, as the LLM takes the information upfront and then implements only the code without needing additional information. Additionally, some DAOs were not created because it did not have the characteristics that provide the solution for certain DAO implementations.(link for full chat).

8.4.5 GitHub Copilot Chat 1

Description:

In this conversation, I provided Copilot with all the features of the system, as well as the code for the domain classes with their properties and methods. I asked the LLM to first provide the code for the domain classes, as well as the code for the DAOs and Services, with the aim of having it use the DAOs more. The LLM did a fairly good job with this (very similar to GPT's responses for the `friendLoanDomain` methods). However, the results were not particularly impressive and were very similar to those from phase 3. This means that the LLM didn't really use the methods, or it used them in limited cases (although it did use some). Additionally, the LLM didn't provide any Step Definitions until I requested them multiple times, and it used assertions in the GIVEN steps again, despite my request not to do so. Overall, the results were mediocre based on the information provided. (link for full chat).

8.4.6 GitHub Copilot Chat 2

Description:

In this conversation, I provided Copilot with the exact same information as in Conversation 1, but I asked it to use as many of the methods and properties as possible to create the step definitions. This instruction had worked well with GPT-3.5, so I thought it would be a good idea to try it with this LLM as well. Overall, the code for the domain classes was good, as always. However, the step definitions were not remarkable, and the results were mediocre. The LLM used assertions in the GIVEN steps again, which is not ideal and oversimplifies things more than necessary. Despite the fact that the LLM used many of the methods I provided, the overall results remained mediocre. (link for full chat).

8.4.7 GitHub Copilot Chat 3

Description:

In this conversation, I provided the LLM with the exact same information as in previous conversations, but I didn't give it the instruction to first create the code for the domain classes/DAOs/Services, in order to see if that would have any impact on the code generation. This change made no significant difference, as the LLM generated almost the same code for the first feature. So, I continued the conversation until all the code was produced. feature.(link for full chat).

8.4.8 GPT -4 Chat 1

Description:

In this conversation, I provided the LLM with all the system features, the names, properties, and methods of all domain classes, as well as the system architecture. I asked the LLM to first generate the code for the Domain classes, DAOs, and Services, and it did so excellently. The LLM understood the system very well, created most of the Step Definitions with ease, and provided good code for many of them. Overall, all this information helped the LLM to produce useful results for many of the Step Definitions, demonstrating the effectiveness of providing comprehensive information from the beginning (link for full chat).

8.4.9 GPT -4 Chat 2

Description:

In this conversation, I provided the LLM with all the system features, the names, properties, and methods of the Domain classes, as well as the architecture and general knowledge of the system.

The LLM produced fairly good results, although it required some repeated guidance to remember to use the DAOs and create all the Step Definitions. Despite these small difficulties, the results were similar to those from the previous conversation in this phase and were satisfactory. (link for full chat).

8.4.10 GPT -4 Chat 3

Description:

In this conversation, I provided the LLM with all the system features one by one, along with the names, properties, and methods of the Domain classes, as well as the architecture and general knowledge of the system. The LLM understood the task easily and required very few messages to produce the results. Despite the good understanding, the results were weaker compared to other conversations. (link for full chat).

8.4.11 GPT -4o Chat 1

Description:

In this conversation, I provided the LLM with all the system features, the Domain classes with their corresponding properties and method names, as well as the system description. I requested the LLM to first provide the code for the Domain, which is not necessary since the LLM does this automatically even without specific guidance. As always, the LLM understood the task very well, provided the code for the Domain, DAOs, and Services with many details, and then gave me the code for the Step Definitions. In many cases, the code was exactly the same as the actual solution, while in some other cases, it produced even better solutions. Overall, the results were quite good, requiring very few messages, and the LLM had a very good understanding of what needed to be done (link to full conversation). (link for full chat).

8.4.12 GPT -4o Chat 2

Description:

In this conversation, I provided the LLM with the exact same information as in Conversation 1 but did not guide it to first generate the code for the Domain, Services, and DAOs. However, as in all previous conversations, the LLM created them on its own. The code results were quite good and similar to those of the other conversations. The LLM utilized the architecture I provided at the beginning of the conversation, gave detailed answers, and needed very few messages to create all the Step Definitions for all the features I provided.(link for full chat).

8.4.13 GPT -4 Chat 3

Description:

In this conversation, I provided the LLM with the exact same information as in Conversation 2, but I decided to give all the Domain classes, Services, DAOs, properties, methods, and features in one message to see if the LLM could understand all the knowledge without it being split into multiple messages. Overall, the LLM reacted in the same way as in previous conversations. The results were quite good, but in some cases, they were less effective compared to Conversations 1 and 2. However, it took very few messages to generate all the Step Definitions for all the features. (link for full chat).