# Exploring How Well Llama3 can Generate State Machines Represented in Umple

**Parva Pathak**

Thesis submitted to the University of Ottawa
in partial Fulfillment of the requirements for the
Masters of Science in Computer Science

uOttawa

Ottawa-Carleton Institute for Computer Science
School of Electrical Engineering and Computer Science
University of Ottawa

# Abstract

Modelling a system is an important part of design which can be time consuming and difficult. A common type of model is a state machine, describing a system or component's behaviour. Multiple languages have been created to make this process smoother, one of them being Umple, which enables describing state machines both textually and graphically, as well as embedding them in multiple programming languages and generating code from them. Although tools such as Umple have made the process easier, developers or business analysts still have to translate requirements into state machines. In this thesis, we investigate how well this step can be automated with the application of artificial intelligence. We show that using modern large language models, Llama 3 in our case, we can allow a user to generate a state machine by only providing a short description of the system requirements. These state machines generated by large language models (LLMs) can be used as a model for the system as is if they meet the requirements or a base that can be improved on. We found that for simple systems using a large language model along with techniques such as retrieval augmented generation and multi-shot learning, can save users large amount of time compared to coding state machines from scratch.

# Table of Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| FM | Foundation Model |
| GPT | Generative Pre-Trained Transformer |
| LLM | Large Language Model |
| RAG | Retrieval-Augmented Generation |
| NLP | Natural Language Processing |
| SE | Software Engineering |
| UML | Unified Modeling Language |
| ICP | Invalid Code Percentage |
| EUCP | Extraneous Code Percentage |
| AF | Additional Features |

# Chapter 1 Introduction

In this thesis we explore how well the large language model (LLM) Llama 3, a modern generative AI tool, can be used to generate state machines represented in the Umple language, which can be used to describe the behaviour of software.

## 1.1 Motivation

State machines are a useful formalism for software specification and implementation, succinctly describing the behaviour of an object, subsystem or system (Merseguer et al., 2002). Describing systems using state machines can result in more precise control over behaviour and greater understandability of that specification than coding the behaviour in a programming language. The process of specifying behaviour using state machines can be simple if given sufficient requirements, though the actual process of creating a state machine from the requirements can still be time consuming due to the many details to consider. The core goal in this thesis is to speed up generation of state machines using generative AI.

We will use the term foundation model (FM) for generative AI models that can be used to create many different kinds of artifacts, including code; FMs are a broader category than the commonly used term ‘large language models’ (LLMs) (Awais et al., 2025), where the latter are specialized at producing language. FMs are trained on large sets of data, the majority of which being unlabelled (Shen, 2024). The term GPT (generative pre-trained transformer) is often used to describe both FMs and the systems that use FMs to create output, although OpenAI has tried and failed to obtain trademark registration over that acronym (David, 2024). We will use the term ‘GPT’ generically, or simply ‘transformer’ interchangeably for the tools that use FMs.

Generative AI can speed up the process of creating many kinds of documents by providing first drafts or boilerplate information. There have been much research attempts to use generative AI to generate code (Iyer et al., 2018) and models (Shehata et al., 2024) from requirements. However, there has been no published attempt to apply generative AI to state machines. The purpose of this thesis is to explore the effectiveness of generating state machines that are expressed in the Umple modeling language; we chose Umple because it has a textual representation in

addition to the classic diagrammatic representation, and because it has capabilities to validate and execute state machines, thus enabling us to validate output.

Even if a transformer can generate modeling code, there is no AI model that can generate code that adheres to the given requirements 100% of the time. Usually, the generated code requires editing to meet the given requirements, which costs the user time. This may lead to FMs not being worth the time to use (at least with current methods and technologies). This difference or lack thereof needs to be measured.

Modern methods of solving similar problems include single/multi-shot learning (Section 2.5) where a transformer is provided single or multiple examples of user inputs and ideal responses. Another method used with foundational models is retrieval-augmented generation (Section 2.10) also known as RAG. In RAG, a transformer is provided a document set and selects relevant documents that may help the system answer a user prompt.

## 1.2 Research Questions

Our research questions are the following:

RQ1. To what extent can LLMs use FMs to generate Umple state machine modeling code without additional information?

RQ2. To what extent can LLMs use FMs to generate Umple state machine modeling code using a single example?

RQ3. To what extent can LLMs use FMs to generate Umple state machine modeling code using a RAG based solution?

## 1.3 Method Overview

To answer our research questions, we first need to select an FM, and a corresponding transformer to use. We chose Llama 3 (described in Section 3.2) as it is a modern open-source model. With this model we do the following experiments:

For RQ1: Provide Llama 3 with sets of requirements and ask it to generate a corresponding state machine in Umple modeling language, then analyze the results (Chapter 4).

For RQ2: Provide Llama 3 with sets of requirements along with an example containing a different set of requirements and a corresponding state machine in the Umple modeling language.

Then we ask it to generate a corresponding state machine in Umple. Finally, we analyze the results (Chapter 5).

For RQ3: Create a Llama 3 based RAG, provide it with sets of requirements and ask it to generate a corresponding state machine in Umple modeling code, then analyze the results (Chapter 6).

## 1.4 Thesis Contributions

This thesis deals with a unique problem in generative AI that has not been analyzed thoroughly. This problem is using AI to generate state machines. This is a unique problem because the generated code needs to follow a specified format for the modeling language (in this case Umple), but the code cannot be analysed using traditional code analysis methods involving test cases since state machines themselves need additional information such as code to read input and generate output, as well as suitable input and output sequences as well, in order to for executable test cases to be used for their evaluation. Common metrics used in the AI code generation space focus exclusively on the quality of code generated while we wanted to see how easy it is to convert the code into a successful state machine.

This problem forced us to create unique metrics that measured the information we wanted. This being how much "effort" is required to fix generated code (Section 3.3). AI code generation in the domain of state machine has not been investigated thoroughly, and we wanted to contribute to the filling of that gap.

The thesis also opens the door for lots of future work (Section 7.4) as this research can be easily expanded on.

## 1.5 Thesis Outline

The rest of the thesis is organised according to the following outline:

- Chapter 2: Background
    - This chapter explains the background knowledge and general concepts required to understand the research in this thesis
- Chapter 3: Research Methods

    - This chapter uses the concepts explained in Chapter 2 to show what is done in this thesis
- Chapter 4: Experiment 1 Zero shot learning
    - The experiment to answer research question 1, described in Section 1.2. We generate state machines with no example in the prompt,
- Chapter 5: Experiment 2: One-shot learning
    - The experiment to answer research question 2, described in Section 1.2. We generate state machines with a single example provided in the prompt.
- Chapter 6: Experiment 3: RAG
    - The experiment to answer research question 3, described in Section 1.2. We generate state machines by providing selected examples in the prompt.
- Chapter 7: Conclusion and future Work
    - In this chapter we discuss the results of our experiments, provide answers to our research questions and suggest further research that can be done.

# Chapter 2 Background

## 2.1 State Machines

State machines in the domain of modeling are used to describe the behaviour of a system through states and transitions (Harel, 1987).

A state machine can be applied to an object, a subsystem or the system as a whole, which we will call stateful elements. State machines are made up of two main parts. States define a unique and meaningful set of characteristics a system has at a point in time. Transitions are the actions required to make the system move from one state to another. Each stateful element can normally be in a single state at a time, although in more complex models, there can be concurrent states, or multiple separate state machines. States are connected by transitions; each transition determines the reaction of the state machine when an event happens. Adjusting the transitions allows the creator to change how a state machine reacts to events. For example, in a state machine for a microwave, if it is in the “Running” state then pressing the stop button should do something, for example switch the microwave to the “Stopped” state, but if the stop button is pressed when the microwave is in the “Stopped” state then the state machine should not react. This can be modelled by adding a transition from “Running” to “Stopped” when the “stopButton” event occurs but not having a transition going out from the “Stopped” state when the “stopButton” event occurs.

## 2.2 Umple

Umple is an open-source modeling technology which allows a user to create software models in UML, including state machines. The models created with Umple can also be used to generate software in traditional coding languages such as C++ and Java (Lethbridge et al., 2021).

Umple addresses two main problems. The first, allowing software developers to develop a system using high-level textual and visual representation along with high levels of abstraction (Lethbridge et al., 2021). The second problem addressed by Umple is reducing the large amount of code that needs to be written to describe a complex system (Lethbridge et al., 2021).

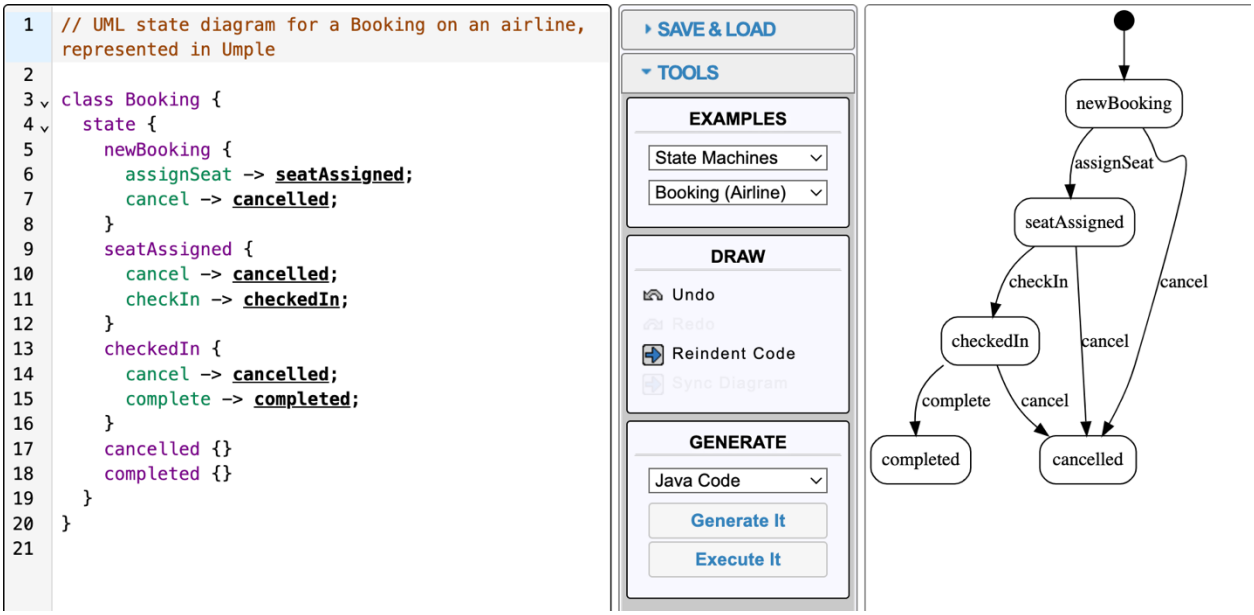


**Figure 1 Airline booking state machine example**

In Figure 1, we see how textual Umple modeling code is mapped to a basic state machine containing exclusively states and transitions. The user can edit the diagram or the text, however in this thesis we will generate the textual form. Extensive details of Umple state machines can be found in the user manual (Umple, 2025) and in several papers (Lethbridge et al., 2021) that focus on that aspect of Umple.

Umple syntax for a basic state machine can be illustrated by the following:

```
class ClassName {
    StateMachineName {
        State1 {
            event -> State2;
        }
        State2 {
            event -> State1;

        }
    }
}
```

One additional feature of Umple that we will use in this thesis, is nested state machines. In nested state machines there can be substates that group similar behaviours within in a state. These can be illustrated by the following:

```
class ClassName {
  StateMachineName {
    State1 {
      event1 -> Substate2b;
    }
    State2 {
      event2 -> State1;
      Substate2a {
        event2 -> Substate2b;
      }
      Substate2b {
        event3] -> Substate2a;
      }
    }
  }
}
```

Umple has many additional features for the creation of state machines such as state machines with actions that allow software methods to be called on a transition or when reaching/exiting a state. A user can also create timed transitions which allow for a transition after a certain period instead of a condition being met. Additional features include concurrent do-activities, guards, state-dependent methods, composition of state machines using traits, and history states (Umple User Manual, 2025). These features are not part of this thesis therefore we do not describe them.

Like other programming languages, invalid syntax causes Umple state machine code to not compile. But Umple has a feature which allows non-Umple code (code that Umple cannot recognize as a state machine, association, attribute, method or other valid Umple feature) inside a class. Umple will assume that this code might be some valid feature of the target programming language to be generated; instead of parsing this code, Umple simply emits it in the generated code. Umple labels this code, “extra code”; it warns the user when it encounters such code in case it was not intended. Sometimes an incorrectly written Umple feature, such as a slightly invalid state machine, will be assumed by Umple to be such extra code (Umple, 2025).

## 2.3 Embeddings

Embeddings are multi-dimensional vectors containing numbers that embed the “meaning” of the original tokens in a piece of text (Wang et al., 2024). These vectors are used to compare

different texts. Embeddings are useful because comparing vectors is easier than comparing raw text, due to mathematical algorithms such as cosine similarity (Section 2.6) and CodeBLEU (Section 2.8). In an effective LLM, embeddings representing tokens with more-similar “meanings” are closer together in vector space compared to embeddings representing more-dissimilar tokens.

The first step to generate embeddings is to tokenize text, this involves breaking up text into “tokens” usually based on words but also on punctuation. The specific behaviour changes based on the tokenizer used.

```
BlackJack Requirements:
- The player and dealer will be dealt cards
- The player can choose to hit or stand
- The dealer can choose to hit or stand
- If dealer gets a blackjack the dealer wins
- If the player gets a blackjack and the dealer does not the player wins
- If the player stands and the dealer has a lower score the player wins
- If the player stands and the dealer has a higher score the player wins
- If the dealer and player have the same score the game will push


BlackJack Umple Modelling Code:
class BlackJack
{
  status
  {
    Start
    {
      dealInitialCards -> PlayerTurn;
      dealInitialCards -> DealerTurn;
    }
    PlayerTurn
    {
      hit -> PlayerTurn;
      stand -> DealerTurn;
      playerBust -> DealerWin;
      playerBlackjack -> DealerTurn;
    }
    DealerTurn
    {
      hit -> DealerTurn;
      stand-> CompareScores;
      dealerBust -> PlayerWin;
      dealerBlackjack -> CompareScores;
    }
    CompareScores
    {
      sameScore -> Push;
      playerHigherScore -> PlayerWin;
      dealerHigherScore -> DealerWin;
    }
  }
}
```

**Figure 2 Tokenized text example (OpenAI, 2025)**

Figure 2 shows an example of how text is tokenized when provided to an OpenAI GPT model. Once the text is tokenized, the GPT converts these tokens into embeddings. In the general case, the same token will result in the same embedding for that token. There are multiple ways to do this, for example a popular method of generating embeddings is “Word2Vec” which was an efficient algorithm for generating embeddings developed in 2013 (Mikolov et al., 2013).

The most modern and popular method used for generating embeddings is to use a LLM specifically created to generate embeddings. For our thesis the model used was Nomic. Nomic is an open-source machine learning model that outperforms similar models on multiple benchmarks (Nussbaum et al., 2024). These benchmarks include MTBE: Massive Text Embedding Benchmark (Muennighoff et al., 2023), and the LoCo benchmark (Saad-Falcon et al., 2024).

## 2.4 Transformer Based Models and Llama

Llama 3 is a large language model and generative pre-trained transformer (GPT) tool developed by Meta in 2024. Predecessors of Llama 3 such as the language model BERT use a transformer architecture, where there are encoders and decoders (Hey et al., 2020). The transformer architecture, along with the attention mechanism introduced by Vaswani et al. (2017), allows the embeddings to change based on its surrounding tokens or the “context”.

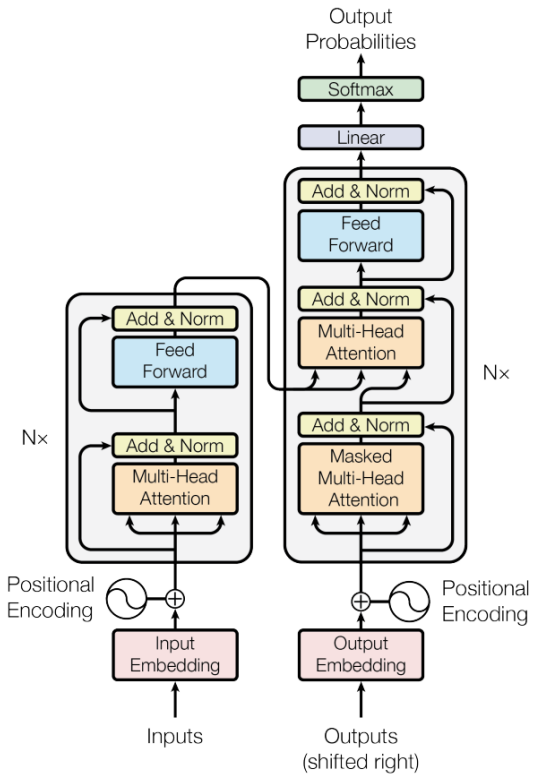


**Figure 3 The Transformer - model architecture. (Vaswani et al., 2017)**

Some transformers are made up of two components, the encoder and the decoder. In Figure 3, the encoder is seen on the left and the decoder is seen on the right. The encoder's job is to define the meaning and context of the input information. It does this by defining relationships between the sequence of inputs (Vaswani et al., 2017). This information gets sent to the decoder where it takes the output of the encoder and generates those tokens are statistically most probable to come next (Vaswani et al., 2017).

Llama 3 and many modern transformer tools such as those created by OpenAI (GPT3, GPT4 etc.), differ from this traditional transformer architecture by having decoder-only architecture. This means that they do not use an encoder and exclusively use a decoder or multiple decoders to decide what the token output will be.

Llama 3 was trained on data from publicly available sources for a total of 15 trillion tokens (Grattafiori et al., 2024). Llama 3 comes in varied sizes based on the number of parameters (measured in billions), these include the 8B, 70B, and 405B models (Grattafiori et al., 2024).

These parameters refer to numerical values that define the behaviour of the model. The parameter count for a model is a function of the number of distinct tokens, the size of the vectors used to represent tokens when creating embeddings, the maximum input sequence of tokens that can be handled, as well as the number and size of layers in the neural net used (Grattafiori et al., 2024).

The more parameters the model has, the larger and more accurate it will be. For this thesis we used the 8B param model as it was being run on a local machine. Without using a cloud-based solution it is not practical to use larger versions of these models.

## 2.5 Prompt Engineering

Prompts are the input provided to a transformer model. These prompts, when written by a general user are in the form of natural language. Prompt engineering is the process of writing prompts using specific methods and formats to get better results from a FM due to the input being structured.

Most transformers break prompts up into two sections, these are the system message and the prompt itself. The system message describes the high-level goal the user is trying to achieve. This message remains the same throughout the conversation a user has with the system. The prompt is the information a user gives to the transformer. The transformer then responds to this information. The user prompt and the response steps can be repeated, and the transformer will remember the previous information provided.

Using prompt engineering, a simulated conversation can be provided to the transformer giving it information on what the “ideal” response should be to a prompt.

Most languages will have a recommended prompt format which gives additional context to the transformer. These formats differ for each model and allow the user to encode information that they would not easily be able to convey such as an example conversation between themselves and the transformer, commonly referred to as “assistant”.

The prompts used in this paper follow Llama 3’s recommended format (Meta, 2024). The format of Llama 3’s prompts involve writing a header. Part of the header lets the transformer know who this string belongs to; this can be the “system” which lets us provide a system message. The other headers are commonly “user” and “assistant”. The “user” header lets the FM know the text

that follows is a prompt from the user and the “assistant” header lets it know what the output should be. The other tag that is used is “eot_id” which “specifies the end of the input message” (Meta, 2024).

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful AI assistant for travel tips and recommendations<|eot_id|>
<|start_header_id|>user<|end_header_id|>

What is France's capital?<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Bonjour! The capital of France is Paris!<|eot_id|>
<|start_header_id|>user<|end_header_id|>

What can I do there?<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Paris, the City of Light, offers a romantic getaway with must-see attractions like the
Eiffel Tower and Louvre Museum, romantic experiences like river cruises and charming
neighborhoods, and delicious food and drink options, with helpful tips for making the
most of your trip.<|eot_id|><|start_header_id|>user<|end_header_id|>

Give me a detailed list of the attractions I should visit, and time it takes in each
one, to plan my trip accordingly.<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
```

**Figure 4 Example Llama3 Prompt (Meta, 2024)**

A common method of prompt engineering is providing examples for a transformer to use to generate a response. These methods are referred to as zero-shot, one-shot and multi-shot learning.

Zero-shot, one-shot and multi-shot learning methods refer to the number of conversational examples given to an FM. Zero-shot is simply giving a FM a prompt and allowing it to come up with a response based on its pre-trained model alone. This may work if the training data contained relevant examples and analogous queries.

As the names would suggest, one-shot learning involves giving a FM a single example while multi-shot learning involves giving a FM multiple examples. These examples are provided using the method of a simulated conversation with the assistant.

## 2.6 Cosine Similarity

Cosine similarity is a way of determining the "distance" between two vectors. Unlike in common life where distance is in 2 or 3 dimensions, in generative AI distance is calculated in n-dimension space where n is the length of each vector representing the embedding of a token. "Cosine similarity measures the similarity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction." (Han et al., 2012).

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \|\vec{b}\|}$$

**Figure 5 Cosine Similarity Formula**

This metric has multiple applications. In this paper it is used to determine the closest examples to use for RAG (Section 2.10).

## 2.7 Pass@K

Pass@k describes chance of generating a solution to a problem in k attempts. This metric is commonly used in functional programming problems because these problems have test cases, allowing for easy validation of the solutions allowing for an easy way to measure a "pass" (Chen et al., 2021).

A transformer can generate a potential solution, which is automatically ran against a set of test cases to see if it is a valid solution, then this process can be repeated as many times as needed with little effort.

A method of calculating pass@k is generating solutions to a problem until a valid solution is found. This method will result in high variance making the metric unreliable in measuring the ability of a transformer to generate a solution.

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

**Figure 6 pass@k formula (Chen et al., 2021)**

A method commonly used to compute pass@k is shown in Figure 6. To calculate pass@k using this method we first generate n samples. Then we count how many of these samples are successful (c). With these 2 numbers we can calculate the probability of a generating a correct sample in k attempts using the formula seen in Figure 6. This assumes $k \leq n$ and preferably much lower than n to reduce variance. A part of the formula not seen in the figure but later mentioned in the paper is if $n - c > k$ then the combination becomes 1 as the first part of a combination not be higher than the second part.

## 2.8 CodeBLEU

CodeBLEU is an evaluation metric for code. It is a weighted average of four metrics (Ren et al., 2020). These include BLEU score, weighted n-gram match, syntactic abstract syntax tree match (AST) and semantic dataflow match.

BLEU score is a method of evaluating machine translation originally developed by Papineni et al. (2002). This is a metric commonly used to assess the effectiveness of solutions to NLP problems (Hou et al., 2024). The metric compares a generated response to the "ground-truth" and provides a score between 0 and 1 (Odu et al., 2024). It does this by matching n-grams between the translations and "ground truth".

Weighted n-gram match is similar but certain keywords are given a higher weight. These keywords depend on the programming language being analyzed. For Java some of these include datatypes such as char, double, int as well as other keywords such as class, enum, interface, return etc. This metric is included because compared to other words, these keywords are more important to the functionality of the code.

Syntactic AST uses a language-specific parser to create an abstract syntax tree match for the predicted code as well as the ground truth. These trees are then compared and scored. This is used to calculate structural correctness.

Semantic dataflow match analyzes the predicted and ground truth code to see if the flow of data is preserved. This is used because it weighs the transfer of data more than what words are used to label it. For example, if the reference code and predicted code have the same logic but use different variable names, they will behave the same but look very different.

All parts of CodeBLEU, except BLEU score are based on the language the code is written in. Since BLEU score is based on n-gram matching, it does not change based on the language. The remaining three parts of CodeBLEU (Weighted n-gram match, Syntactic AST, and Semantic data-flow match) depend on the language the code is written in.

For weighted n-gram the keywords that need to be weighed higher change depending on the language. For example, “interface” is a keyword in Java, but it is not a keyword in Python.

The remaining two parts Syntactic AST and Semantic dataflow match require the code to be parsed, either to create an abstract syntax tree or determine what variables are being used and how they are being used.

## 2.9 Levenshtein Distance

Levenshtein distance is a measure of how different two strings are. It does this by calculating the number of transformations required to get from one string to another string. These transformations are:

- Insertions: adding a character
- Deletion: removing a character
- Substitution: replacing a character with another character

## 2.10 Retrieval-Augmented Generation (RAG) Applied to Code

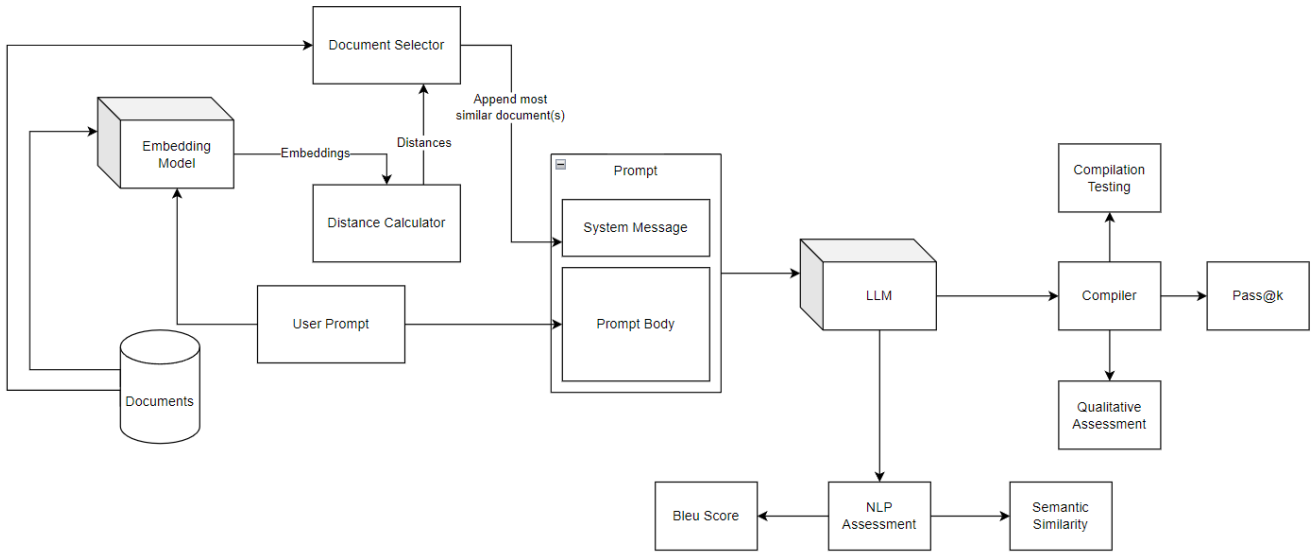


**Figure 7 RAG applied to code**

RAG involves a transformer using relevant documents to generate a response to a prompt. To do this we first generate a database of documents (Gao et al., 2024). These documents can be anything that will help the transformer complete its purpose. For example, if the goal is to output information about a country, the database could be a collection of encyclopedia pages about different countries that the transformer can use.

With the database of documents, each document is sent to an embedding model. This is a model used to generate embeddings, there are multiple options for models such as "Word2Vec" or an LLM created specifically to generate embeddings. The user's prompt is taken, and the same model is used to generate embeddings for it as well (Gao et al., 2024). We now have embeddings for the user's prompt and each of the documents. All these embeddings are then sent to a distance calculator.

The next step is to calculate distance measurements. In the distance calculator we compare the embeddings of the user's prompt to the embeddings of each of the documents in the documents database and generate distances. These distances will tell us how similar a document's "meaning" is to the user prompt's "meaning" (Wang et al., 2024). These distances are then sent to the document selector.

The document selector uses the distances provided to choose documents to give to the transformer. Its common practice to use the document(s) that are most similar to the user's prompt. Therefore, the distances for each document are sorted from low to high and the plain text version of the document(s) are appended to the system message.

This system message is then combined with the original prompt body to create the final prompt to send to the transformer. The transformer will then generate a response (Gao et al., 2024). In this example we are using code so we will assume that is what the transformers generating. This code is sent to a compiler to be tested.

This compiler will compile the code allowing for compilation testing to see if the generated code compiles. The code is also run against test cases. This lets us to see how successful the LLM is in generating code to complete the task it was asked to complete. Metrics can be calculated using this information such as pass@k, described earlier. The code can also be assessed qualitatively, this is more open-ended, but an example is how much time the code took to run, if there was useless code in the solution etc.

The code can also be analyzed using NLP methods, these do not require the code to be compiled. Methods of doing this include BLEU Score (Section 2.8) as well as semantic similarity such as using cosine similarity (Section 2.6).

## 2.11 Code Generation by Transformers

Liu et al. (2024), studied ChatGPT's proficiency in generating code. They did this by giving it publicly available programming problems that it was asked to solve using multiple programming languages (C, C++, Java, Python, and JavaScript).

Since the authors used publicly available problems with defined tests and errors, they were able to do a lot of quantitative analysis. Since their code returned specific errors, their results can be fed back into the prompt they used.

In the context of this thesis though, since there is no way to automatically check the output of a state machine (besides checking if the results are compilable) a transformer cannot improve on its output without human intervention.

Liu used well known programming languages (C, C++, Java, Python, and JavaScript), of which ChatGPT has learned countless examples. This contrasts with the work of this thesis where we are attempting to generate state machine code written in Umple. Umple is not only a substantially less popular language, but we are using a subset of it, that being the state machine part as opposed to class diagrams, where there are more examples.

The style of question used by Liu (2024) is one that ChatGPT has seen in its training data. The authors do separate problems that were created before and after ChatGPT was trained but ChatGPT has seen many examples of the "style" of question asked in the paper. This contrasts with our thesis's style of questions, which are in a format created for this thesis.

ChatGPT had a poor success rate in completing the questions asked by Liu's team. The most successful language (Python3) had a total success rate of 23.93% on problems it had not seen before. This is broken up into 57.30% of the easy problems solved, 18.13% of the medium problems solved and 2.20% of the hard problems solved.

The authors used ChatGPT as their foundational model of choice. They are using ChatGPT with GPT-3.5 which has 175 billion parameters.

## 2.12 Evaluation Metrics

In the next two subsections we will discuss two approaches for using metrics to evaluate generated software from LLMs: Evaluation based on code quality and evaluation derived from natural language approaches.

### *2.12.1 Evaluation Based on Code Quality*

The classifications used by Liu et al. (2024) for determining the functionality of the generated code are:

- "Accepted: The submitted code snippet passes all test cases."
- "Wrong Answer: The submitted code snippet has no compile errors but cannot pass all test cases"
- "Compile Error: The submitted code snippet cannot be compiled."
- "Time Limit Exceeded: The runtime of the submitted code snippet exceeds the permitted execution time."
- "Runtime Error: The execution of the submitted code snippet triggers a runtime error for at least one test case."

These functionality metrics are related to the results of functional tests. In our work in this thesis, we do not have a comprehensive set of test cases, so we are not able to use the above approach without adaptation.

Other classifications used by Liu et al. (2024) are:

- Wrong Detail
    - "These detail errors stem from a little misunderstanding (e.g., a word) of the given problem or the generated code that is not consistent with the understanding of the problem."
- Misunderstanding Certain Content
    - "The code generated by ChatGPT does not hold the main condition of the given problem. However, the algorithm used by the generated code is suitable."
- Misunderstanding Problems

    - "ChatGPT misunderstands or does not understand the problem description given."

Additional metrics have been used in the domain of code validation, specifically code generated by LLMs (Liu et al., 2023). Metrics used to reduce the size of the test suite include:

- Code coverage
    - "The amount of code elements (e.g., statements or branches) executed by each test" (Liu et al., 2023)
- Mutant killings
    - Generating large numbers of programs that slightly vary from the ground-truth, these "mutants" are then ran against a test case to see its effectiveness (Liu et al., 2023).

These metrics are specifically for reducing the size of the test suite which is not part of the current problem being researched. The problem is validating whether a program fulfils the given requirements, rather than a set of tests.

Recent research analyze transformer effectiveness writing functional code (Chen et al., 2021; Liu et al., 2022), commonly use the pass@k metric. This metric measures the chance of the AI generating a correct solution in k number of attempts.

An early paper from 2018 where transformers are used to generate Java code, simple severity descriptors are used to qualify the severity of errors (Iyer et al., 2018). These qualifiers are:

- Totally Wrong
- Marginally Correct
- Mostly Correct
- Exact Match
- Semantically Equivalent

In another paper about generating Java code, developers generated Java methods with different AI assistants (Corso et al., 2024). The four qualifiers this paper used were:

- Correct: method passes all test cases and passes manual review
- Plausible: method passes all test cases but fails manual review
- Incorrect: method does not pass all test cases

- Invalid: code did not compile

Unlike similar papers that have unique qualifiers, this paper has concrete definitions of each of them.

Another paper that measured the usefulness of AI code assistants (Millam et al., 2024). This paper used the following metrics:

- Extraneous code: number of lines and characters of code that is useless and can be removed without "negatively affecting the code's readability or efficiency" (Millam et al., 2024)
- Major modifications: Code would not compile without significant effort to fix
- Minor modifications: Code would not compile without minimal effort

These metrics were then given threshold values to determine if the respective AI assistant "passed". These metrics (besides extraneous code) are vaguely defined and may vary based on the person examining the generated code.

### *2.12.2 Evaluation Based on Natural Language Approaches*

Common natural language processing (NLP) metrics may be used to assess generated code. The issue with this in our case is the output of the FM will be Umple modeling code which is different from natural language.

A paper dealing with a similar issue to the one we faced discussed generating *assurance cases* with LLMs (Odu et al., 2024). The assurance cases in the paper were written in natural language, they were evaluated by three metrics (Odu et al., 2024):

- Exact Match
    - Compares the outputs character by character and determines the similarity between the expected and actual result.
- BLEU Score
    - Measures the similarity between a reference and a generated sentence (Section 2.8).
- Semantic Similarity
    - Cosine similarity to measure the similarity between the ground-truth and the generated cases (Section 2.6).

These metrics are commonly used for NLP problems where the input and the output are both written in natural language. Our FM is generating Umple modeling code. These metrics may not give insight on if the output code was correct but are still worth considering. As stated in the evaluation of LLMs for code generation paper, “classic NLP metrics like BLEU score are no longer reliable in the context of program synthesis” (Liu et al., 2023).

We will discuss the final approach we use for evaluation metrics in Section 3.3.

# Chapter 3 Research Methods

## 3.1 Experiments

To answer our research questions, we perform a set of experiments leveraging zero-shot, one-shot and RAG with multiple examples to measure FMs' abilities to generate state machines in the Umple modeling language.

Our initial reaction may be to assume providing multiple examples is always better, but a risk is over-fitting as the information in our prompt has a disproportionately high effect on our results compared to another knowledge the FM has. This can cause the generated code to be too similar to the example code.

## 3.2 Foundation Models

For our FM we used Llama 3 because it outperformed similar models in the most common use cases including general, common sense, knowledge, math and reasoning, reading comprehension, and code (Grattafiori et al., 2024). All these benchmarks are important in our use case, but the ones of interest are math and reasoning, as well as code. We decided to use the 8 billion parameter model because it will be running on a local machine, this is a smaller version than the commonly used ones.

## 3.3 Evaluation Metrics

Evaluation metrics are difficult to create because of the unique problem of generating modeling code. The generated code can be compiled, therefore common metrics for analyzing code may be used but though Umple code does compile, it cannot be tested against any test cases as Umple code does not "run" in the traditional sense unless a main program is present in the Umple source code, and we are not asking the AI model to generate such a main program.

The metrics must be able to analyze the Umple modeling code's ability to be compiled, as well as its adherence to the provided requirements. The code's ability to be compiled can be measured trivially, but adherence to the requirements is not simple. Since the test cases cannot be run against Umple code, the generated code must be compared against the requirements.

To determine evaluation metrics, we returned to our original goal, which is providing a tool to help users generate state machines given a set of requirements. Ideally the LLM should generate a state machine that fulfills all the given requirements which it may be able to do sometimes. If the LLM was to generate a state machine that only fulfilled some of the requirements, the user could use it as a baseline and build upon it and transform it into their desired state machine saving them time over creating a state machine from scratch. Not all generated state machines are equally useful, therefore a method of quantifying how "useful" the generated state machine is must be used. To do this we used Levenshtein distance (Section 2.9).

We could have calculated the Levenshtein distance from the generated state machine to the "ground truth" state machine but this has its issues. Like most coding languages, variable names can be different but behave the same. Another issue with comparing the generated code to the ground truth is the order of the states (except for the first state which Umple considers to be the start state), does not matter, the machine will behave the same. This is also true for the transitions within a state, their order does not matter, the machine will behave the same. The order of the sates and the transitions in the ground truth examples are arbitrary and the state and transition names were created by the person who wrote the state machine. Therefore, even if the LLM generated a state machine that behaved the same as the ground truth, there may be a large amount of distance added.

Calculating the metric this way would not fit for our original goal because a user would not be converting a state machine to fit an exact ground truth, in practice they would build on the generated code. To account for the we created a "reference" state machine for each of the generated outputs which was a corrected version that fixes errors as well as fulfilling all the requirements. This is how the tool will be used in practice and avoids the pitfalls mentioned in the previous paragraph.

An issue that appears when using raw distance is the complexity of the system, we want to generate a state machine for. For example, a Levenshtein distance value of 200 for a state machine is meaningless in a vacuum because if the final reference state machine was only 100 characters and the generated state machine was also 100 characters then this distance value would mean all characters in the original have been deleted and a completely new state machine had been created. On the other hand, if the generated state machine is 1300 characters and the reference state machine is 1500 characters then a distance value of 200 would be good, because this means all the generated

code is useful and the user only needs to add some of their own code, this may be to fulfill requirements or fix errors.

To remedy these issues, we divide the distance by the total length of the reference state machine. The length of the reference state machine will give an idea of how "complex" the system is. We will call this the "normalized Levenschtein distance" and use it as a measure of how much "effort" is required to transform the generated state machine into a version that meets the requirements.

For the normalized Levenschtein distance we will use multiple thresholds for "passing". A pass for our use case is a user deciding the generated state machine is worth using and it is better than starting from scratch or generating a new state machine. These will be set at 0, 0.2, 0.3 and 0.5. Zero would be for a perfect state machine, 0.5 would be the edge of acceptable. The remaining two thresholds of 0.2 and 0.3 were decided on after calculating the normalized Levenschtein distance for most of the generated state machines as many of our values fell in this range. We note that many of the cases fell between 0.2 and 0.3, hence these were also set as thresholds.

Along with this metric we will also be using CodeBLEU (Section 2.8) to compare the generated code to the ground truth code. As stated in the CodeBLEU section, three out of four parts of CodeBLEU depend on the language used. Unfortunately, there is no CodeBLEU metric for Umple, so we needed to use the closest possible language, which was Java. This means all parts of CodeBLEU, except BLEU score, also known as "n-gram match score", will be inaccurate. Therefore, in our data we will include BLEU score on its own, along with CodeBLEU.

Another metric that needs to be used is "Invalid Code Percentage" (ICP), which is the number of lines of code that are invalid and cause the Umple code to not compile. This metric is unique to state machines as in similar papers that measure generated code, code that does not compile is classified separately and there is no measure of what is needed to correct it.

An Umple-specific metric being used is "Extraneous Unparsed Code Percentage" (EUCP). This measures the percentage of lines of code that are labelled "extra code" by Umple (Section 2.2). It is not likely this extra code would be useful for the user and would likely simply be removed, but in some cases, it might be very similar to a valid result, just not quite parsable.

An additional metric we use records the percent of lines that represent where the LLM has attempted to add an additional feature or features not found in the requirements. This metric, which we abbreviate as AF, is neutral as it depends on what the user wants. Some users may appreciate

the additional feature even if it was not implemented correctly while another user would like the LLM to generate a state machine strictly based on their requirements. Additionally, a broken feature may need to be removed causing the normalized Levenschtein distance to be higher.

## 3.4 State Machine Examples

For multi-shot learning and RAG, we will be using the five state machine examples explained below. These will also be the same state machines we ask Llama 3 to create when we provide its corresponding requirements. When we ask Llama 3 to create a state machine, we will leave that answer out of the reference set.

### *3.4.1 Blackjack*

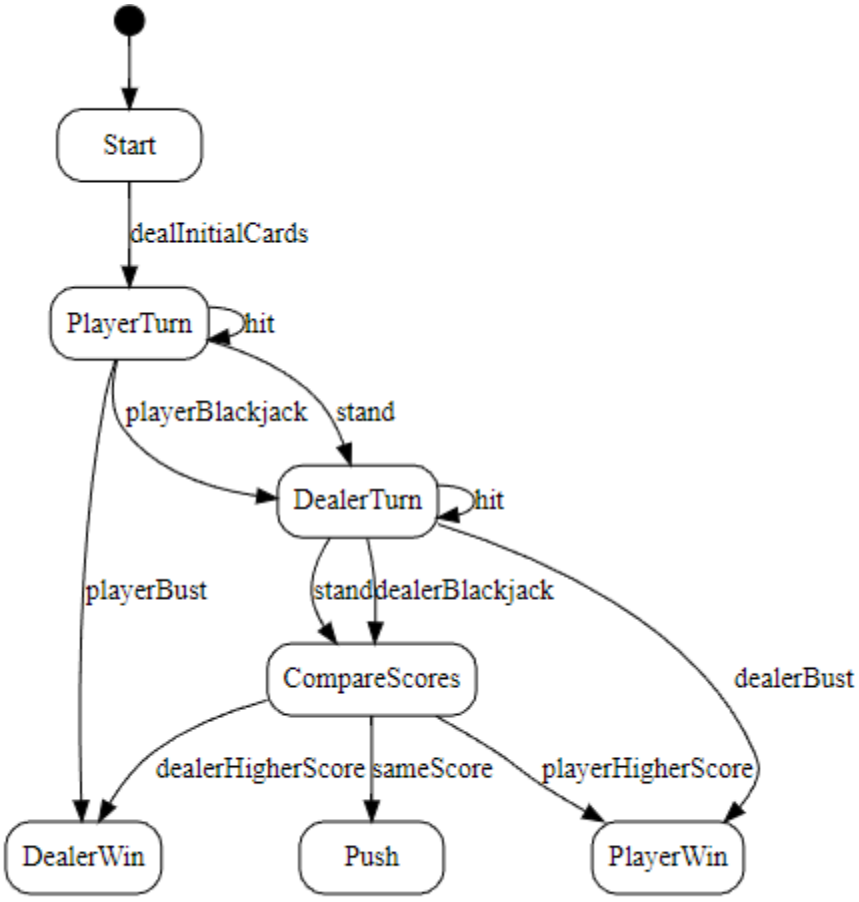


**Figure 8 Blackjack example state machine**

Requirements:

- The player and dealer will be dealt cards
- The player can choose to hit or stand
- The dealer can choose to hit or stand
- If dealer gets a blackjack the dealer wins
- If the player stands and the dealer has a lower score the player wins
- If the player stands and the dealer has a higher score the dealer wins
- If the dealer and player have the same score the game will push

This state machine is a simplified version of blackjack which has one dealer and one player. Normally in blackjack the dealer starts with two cards (one being facedown), this has been removed to further simplify the state machine. In casinos, the dealer has rules for when they should hit and when they should stand, these have been removed to simplify the state machine. One casino rule is implicitly expressed with these requirements. That rule being if both the player and the dealer get a blackjack the player wins. This is expressed with the requirement stating if the dealer gets blackjack, they win but there is no rule like that for the player. It is not common knowledge but there is nothing special about a player getting blackjack, that is simply the highest point total they can get to.

### 3.4.2 Course Section

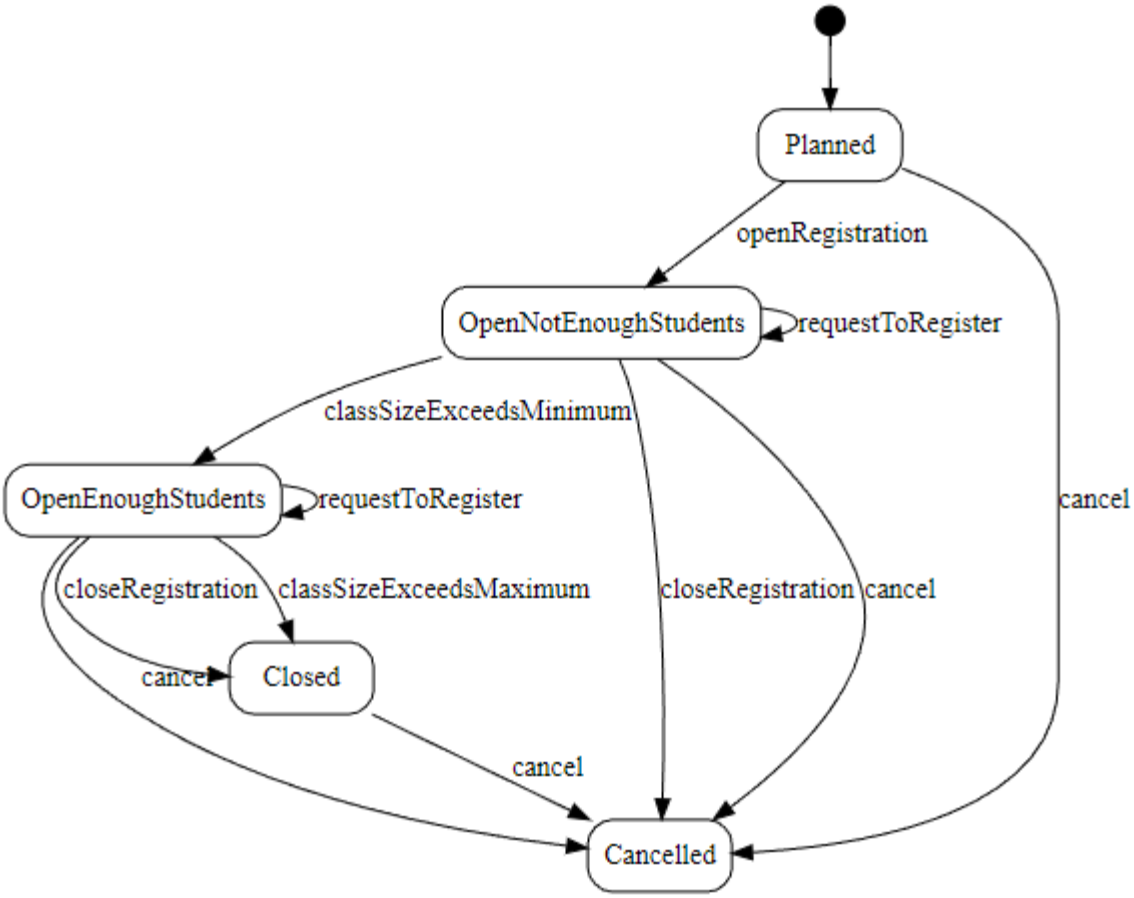


**Figure 9 Course section example state machine**

Requirements:

- A course will initially be in the planning stage
- Once a course is open students can request to register
- A course needs to reach a minimum number of registered students
- When a course exceeds its maximum number of students, the course is closed
- If the course did not reach the minimum number of registered students, when the deadline for registration is reached the class is cancelled
- If the course did reach the minimum number of registered students, when deadline for registration is reached the class is closed
- The class may be cancelled at any step of this process

This is a much "vaguer" example compared to blackjack as this is not a standardized system so the FM will have to heavily rely on the provided requirements. This is an example provided in the official Umple documentation so the FM may have seen this example before. This may bias the results; this will be further explored in the experiments.

### *3.4.3 Credit Card Transaction*

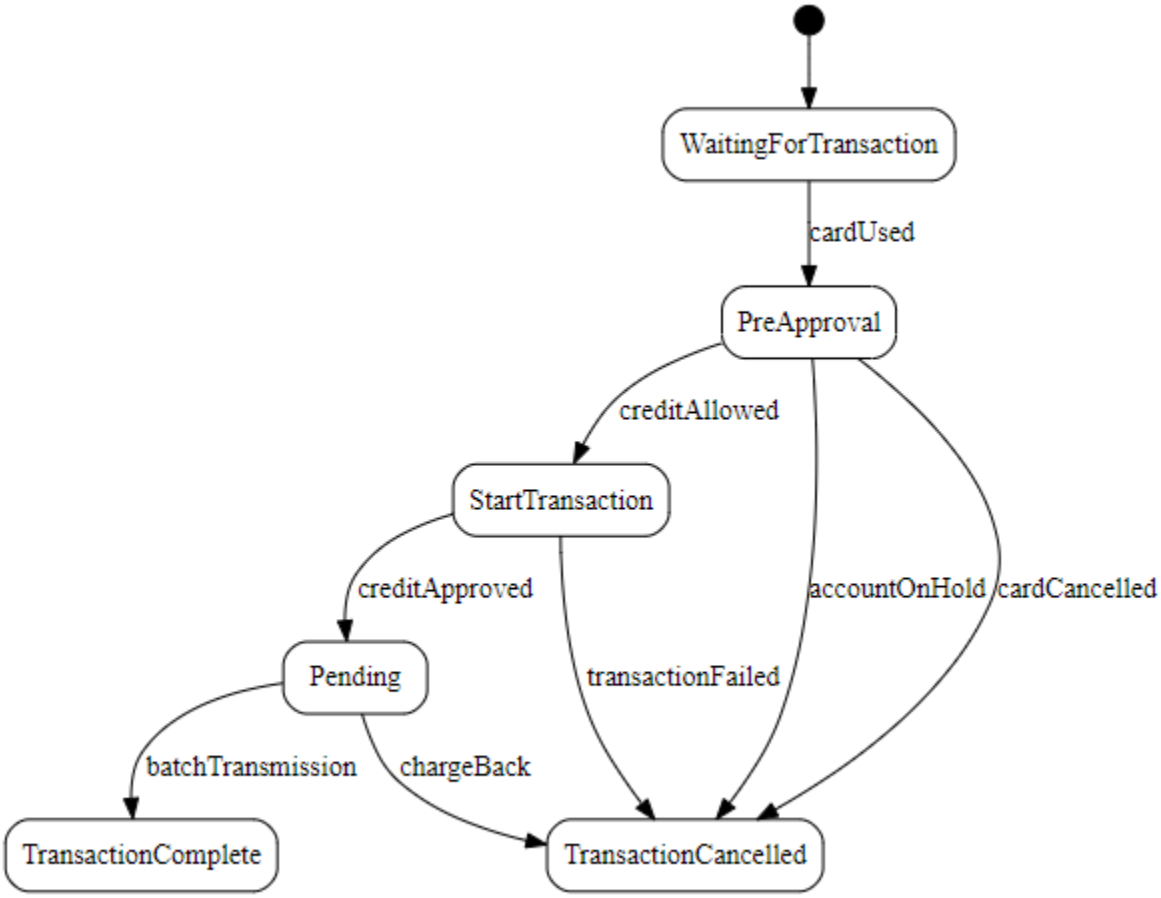


**Figure 10 Credit card transaction example state machine**

Requirements:

- The different stages of the approval process must be included (Pre Approval, Cancelled, Complete, Failed)
- The status of the account must be included (On Hold, Not On Hold)

- The transaction can be successful or fail

This state machine has only 3 requirements provided and like the previous state machine (course section), there is no standardized system for a credit card transaction so the FM will have to rely on the few requirements provided. These requirements explicitly say some of the states should be created. Overall, the state machine is simple.

### *3.4.4 Driver License*

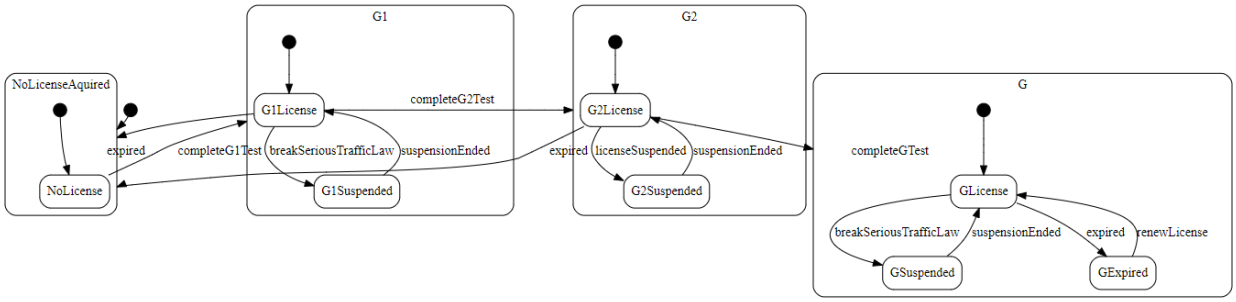


**Figure 11 Driver license example state machine**

Requirements:

- The user can have no license, a G1 license, a G2 license or a G license
- To get a G1, a test must be complete
- To get a G2, the user must have a G1 and a test must be completed
- To get a G, the user must have a G2 and a test must be completed
- Each type of license can expire.
- If a G1 or G2 expires then the license is lost
- If a G expires it can be renewed
- Each type of license can be suspended

The driver license system is complex. FMs may have some information about this as the requirements provided are for the Ontario Canada driver license process (this is mentioned in the prompt), but this information may be difficult for the FM to extract as it likely also knows about the driver license process in other countries.

When an FM is provided this state machine as an example it will give the FM the knowledge of nested state machines. This is something Umple provides access to but is not needed for any of the state machines in this paper. This state machine can be created without the use of nested state machines.

### *3.4.5 Hotel Stay*

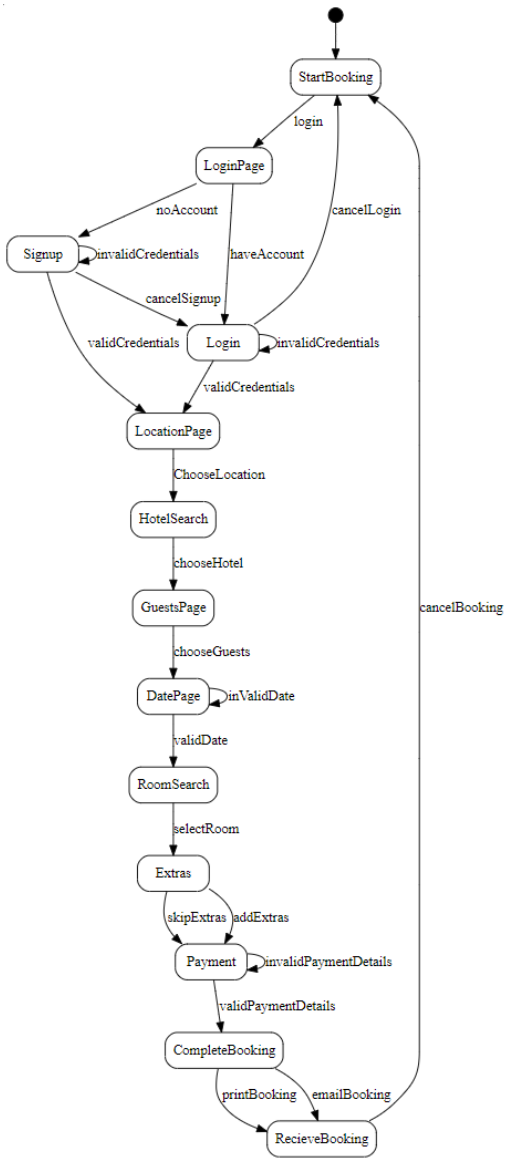


**Figure 12 Hotel stay example state machine**

Requirements:

- The user can optionally login to the hotel website
- The user can choose a location
- The user can choose a hotel
- The user can choose the number of guests
- The user can choose the date range of the visit
- The user can choose a room type
- They user can pay for the hotel
- The user can cancel the booking

This is a very simple system that has many requirements describing it. It is the step-by-step process of booking a hotel online. This process is slightly different for each website, but they are still similar. The provided requirements are all the steps required. This system is very simple to implement as a state machine and very easy to expand upon. The most difficult part is the optional login step as it is the only step that is not mandatory.

## 3.5 Multi-Shot Learning

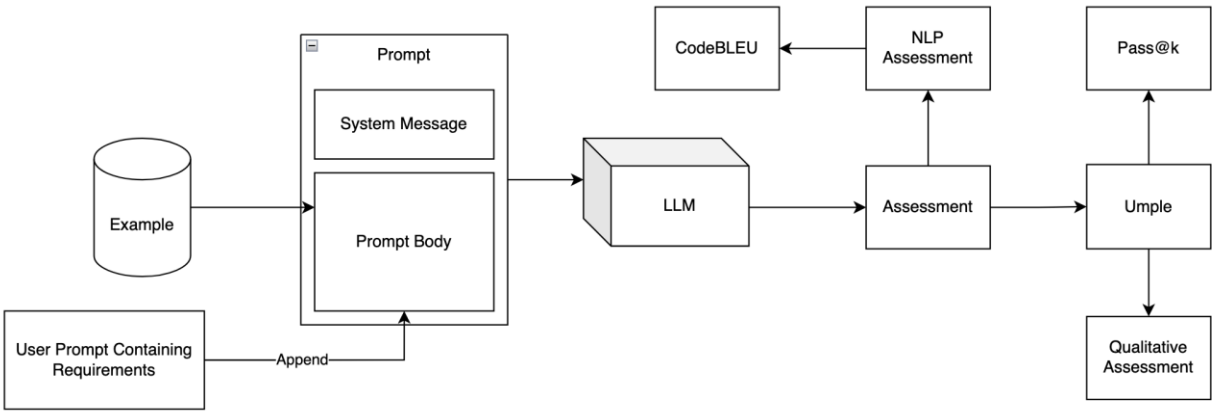


**Figure 13 Diagram for Multi-shot learning**

For our first two experiments we will be using zero-shot and one-shot learning. Our method of multi-shot learning is shown in Figure 13. The prompt is created with a combination of an

example conversation between a user and the system along with a user's prompt containing a description of the desired state machine along with its requirements.

Here is the prompt format used for this specific use case:

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful AI assistant that designs state machines in the 'Umple' modeling language based on a set of requirements. You will not describe or explain the code you have written.<|eot_id|>

<|start_header_id|>user<|end_header_id|>

Write a state machine for [DESCRIPTION OF EXAMPLE STATE MACHINE] with the following requirements -

- [EXAMPLE REQUIREMENTS]

<|eot_id|>

<|start_header_id|>assistant<|end_header_id|>

[UMPLE CODE DESCRIBING EXAMPLE STATE MACHINE]

<|eot_id|>

<|start_header_id|>user<|end_header_id|>

Write a state machine for [DESCRIPTION OF STATE MACHINE] with the following requirements -

- [REQUIREMENTS]

<|eot_id|>

<|start_header_id|>assistant<|end_header_id|>
```

The code above shows an abstracted conversation between the user and system. There is a simulated conversation where the user provides a set of requirements, and the system generates a state machine. This state machine represents an ideal state machine for the provided set of requirements that the system should be emulating in its response. After this the real user provides a description of a different state machine along with a set of requirements for it. The system attempts to generate a state machine based on the provided requirements.

To do zero-shot learning, the initial simulated conversation is removed. Therefore, the prompt only consists of the system message along with the real user input (a description of the state machine and a set of requirements).

This prompt is then sent to our LLM Llama 3 which generates Umple code to be assessed. This involves testing what percentage of the code is invalid or extraneous. Along with these the normalized Levenschtein distance will be calculated and given different passing thresholds the pass@k will be calculated. Along with these Some NLP assessment will be done by calucating the CodeBLEU.

## 3.6 RAG

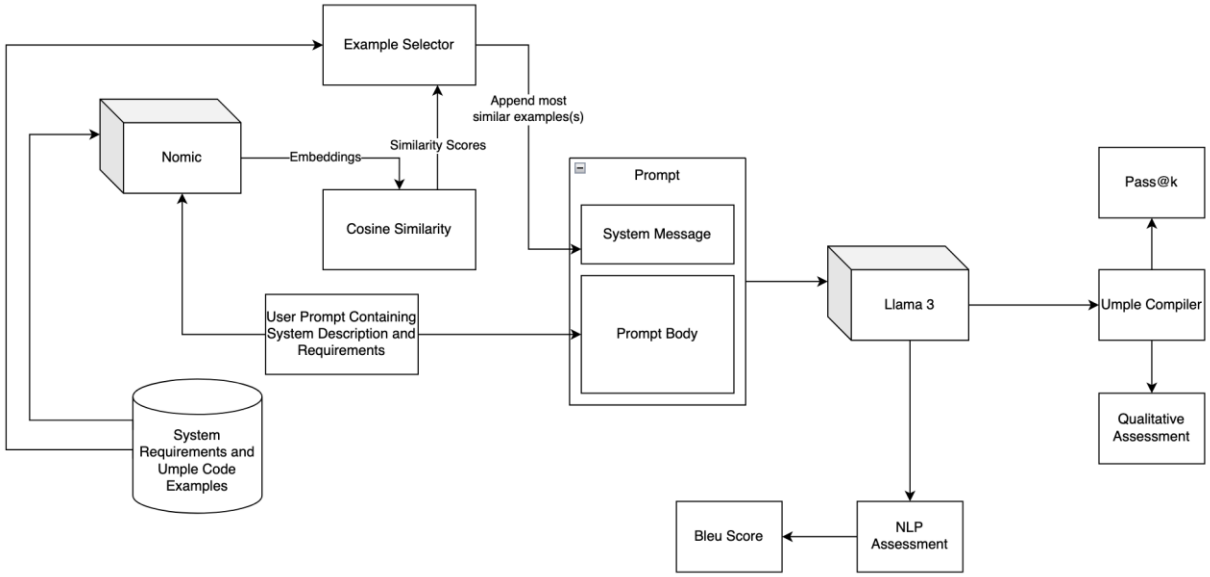


**Figure 14 Diagram for RAG**

To use RAG, we first generate a database of documents. Initially we attempted using the state machine pages in the Umple documentation (Umple Documentation). The results were unimpressive and in many cases were worse than not having any examples because the FM kept trying to explain the state machine instead of creating it (because the Umple user manual explains all the examples).

The dataset we settled on is a version of the examples containing a short description of the system along with the requirements and the Umple code to generate it. This dataset had much

better results. With our database of documents, we can send each document to our Nomic embedding model. Once we have the embeddings for the document, we take the user's prompt and use the same model to generate embeddings for it as well. We now have embeddings for the user's prompt and each of the documents. All these embeddings are sent to a distance calculator.

To calculate our distance measurements, we use cosine similarity (2.6). Cosine similarity is different from other distance measurements because it measures similarity, so it is a metric we want to maximize. In our distance calculator we compare the embeddings of our user's prompt to the embeddings of each of our documents in the documents database to generate similarity scores. These similarity scores will tell us how similar a document's "meaning" is to the user prompt's "meaning". These similarity scores are then sent to the document selector.

The document selector uses the similarity scores provided to choose documents to give to Llama 3 which generates our state machine in Umple code. Therefore, we sort the similarity scores for each document from high to low and append the plain text version of our documents to the system message.

Our newly created system message is combined with the user's prompt to generate a final prompt to send to Llama 3. Llama 3 then generates state machine code in Umple which is then tested.

The generated Umple code first gets sent to the Umple compiler. First, we test the code's ability to compile, this means checking if the generated code follows Umple state machine syntax. If it does not compile, we calculate the percentage of invalid code, if it compiles but has extra code we calculate the percentage of extraneous code. Then the generated Umple code is analyzed based on its ability to fulfil the requirements stated in the prompt. Finally, the code is evaluated qualitatively. All these processers will use the metrics discussed in Section 3.3.

The generated Umple code is also analyzed using NLP methods. These include CodeBLEU, explained in Section 2.8.

# Chapter 4 Experiment 1 Zero shot learning

For the initial experiment we tested if Llama 3 can generate a state machine in Umple using zero-shot learning, by exclusively giving requirements.

**Table 1: Zero-shot learning compilation**

| Requirements | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| BlackJack | Error | Error | NSF | NSF |
| Course Section | Error | NSF | NSF | NSF |
| Credit Card Transaction | NSF | Error | NSF | Error |
| Driver License | Compiled | NSF | Error | NSF |
| Hotel Stay | Error | Error | NSF | Error |

None of the runs resulted in passable state machines. The generated code was largely Java code embedded into some basic Umple syntax. NSF means “No State Machine Found”. NSF can occur if the code is, for example all methods, or if Umple classifies all the code as “extra” and there is nothing left to compile.

Below is a one of the outputs for “Blackjack”. Although it technically compiled, most of the lines are simply classified by the compiler as extra code.

```
class BlackJackGame {
  state = Init
  variables = {playerHand=Array, dealerHand=Array, playerScore=Int,
dealerScore=Int}

  Init:
    playerHand = []
    dealerHand = []
    playerScore = 0
    dealerScore = 0

  Deal:
    while card in (1..10) {
      playerHand.add(randomCard)
      dealerHand.add(randomCard)
      playerScore += valueOf(card)
      dealerScore += valueOf(card)
    }
    if playerScore > 21 { goto Bust }
    if dealerScore > 21 { goto DealerBust }

  PlayerHit:
    card = randomCard
```

```
    playerHand.add(card)
    playerScore += valueOf(card)
    if playerScore > 21 { goto Bust }

  PlayerStand:
    return DealDealer

  DealDealer:
    while (dealerScore < 17) {
      card = randomCard
      dealerHand.add(card)
      dealerScore += valueOf(card)
    }
    if dealerScore > 21 { goto DealerBust }
    compareScores

  Bust:
    dealerScore += valueOf(randomCard)
    compareScores

  DealerBust:
    player wins
    return PlayerWins

  compareScores:
    if (playerScore == dealerScore) {
      game pushes
      return Push
    } else if (playerScore > dealerScore) {
      player wins
      return PlayerWins
    } else {
      dealer wins
      return DealerWins
    }
}
```

The GPT tries to use Java style code and logic as seen in the “onEntry” function in the “stand” state. Though the states have aspects that resemble Umple syntax, the logic inside the functions are written in Java.

Such state machines would take a lot of work to refactor into compilable code, let alone code that meets the requirements. It would therefore be useless.

The single run that resulted in a compiled state machine only had a single transition and 10 states. With heavy refactoring there is some logic coming through, but the state machine is filled with unreachable states and states that are dead ends. This state machine has no purpose to a user.

These state machines had so many errors it would have not been worth it to calculate any metrics as each state machine would need to be completely rewritten. This is not something that would be useful to a user trying to save time, therefore we can classify this experiment as a failure.

# Chapter 5 Experiment 2: One-shot learning

For one shot learning, we tested Llama 3's ability to generate state machine code using a single example. For each of the 5 state machines we provided a single example from the pool of remaining examples. We generated each state machine with each example for 5 iterations.

## 5.1 Syntactic Analysis

**Table 2: Average ICP (invalid code percentage), EUCP (extraneous code percentage), Normalized Levenschtein Distance and AF (additional features) for each system using one-shot learning**

| Metrics | Blackjack | Course Section | Credit Card Transaction | Driver License | Hotel Stay |
|---|---|---|---|---|---|
| Average ICP | 2% σ=0.11 | 0% σ=0.00 | 0% σ=0.00 | 0% σ=0.00 | 0% σ=0.01 |
| Average EUCP | 5% σ=0.12 | 1% σ=0.01 | 1% σ=0.01 | 0% σ=0.00 | 1% σ=0.01 |
| Normalized Levenschtein Distance | 0.46 σ=0.46 | 0.18 σ=0.18 | 0.13 σ=0.17 | 0.29 σ=0.16 | 0.17 σ=0.28 |
| Average AF | 0.85 σ=0.36 | 0.50 σ=0.50 | 0.80 σ=0.40 | 0.35 σ=0.48 | 0.55 σ=0.50 |

Overall, the ICP and EUCP was very good and was much higher than in zero shot learning. It was rare for a machine to have any code causing it to fail compilation. There was also a low amount of extraneous code, this is positive because it was nearly impossible for the LLM to understand Umple well enough to use code from other programming languages and have it be useful.

Llama 3 with one-shot learning is best at generating state machines for "Credit Card Transaction" with an average normalized Levenschtein distance of 0.13. This state machine is for a system that is very common, and the requirements are open ended giving the LLM a lot of leeway. This let the LLM easily generate a machine that fulfills the requirements while also giving it room to add features. The most common features added are additional security checks not found in the requirements. Also, because these additional features are simple to add, they were rarely broken and did not need to be fixed or removed, lowering the distance value.

The system with the second lowest normalized Levenschtein distance value is "Hotel Stay" with an average normalized Levenschtein distance of 0.17. The LLM performed well with this system was because it was a simple set of instructions that lead from page to page. We expected

the average AF to be higher for hotel stay because it was very simple to add a way to go back to a previous page and that would be an additional feature. Some of the generated state machines did this, but fewer than expected.

The next best state machine is “Course Section” with an average normalized Levenschtein distance of 0.18. The LLM only did slightly worse with this system, there is only an increase of 0.01 in the average normalized Levenschtein distance compared to the “Hotel Stay” system. This was the only system that had a state machine previously on the internet and was not made exclusively for this thesis. It is unknown if Llama 3 had seen the state machine before but that may have biased the results for it. It is a simple system that is used to teach undergraduate students the basics of state machines, so that may also be a reason the LLM consistently generated a state machine for it that required few changes.

The second worst state machine is driver license. This is a unique system because it is for the driver license process specifically for Ontario Canada. There are some unique aspects to Ontario licenses that is not seen in other countries. For example, in Ontario there are three levels of licenses, a G1, G2 and G. In countries such as the United States there are only two levels. Another unique aspect about Canadian licenses is if a G2 expires, the owner loses the license and must start from the beginning. Overall, the LLM struggled with this state machine.

Finally, the worst state machine was blackjack. This was unexpected because Blackjack is a game with lots of information online and is not very complex. We expected this to be one of the better state machines. The main issues the LLM ran into when trying to generate blackjack was it not understanding the player and the dealer had to take turns and the dealer only plays after the player. Blackjack did have a high average AF; this was because the LLM added additional rules found on the internet and rules commonly used in casinos such as adding a threshold value, this is where a dealer must hit if they are below this value or stand if they are at or above it. This is commonly 17, which is what the LLM used most of the time.

## 5.2 Pass@K Analysis

**Table 3: Pass@1 One-Shot Learning**

| | 0 | 0.1 | 0.2 | 0.3 | 0.5 |
|---|---|---|---|---|---|
| Blackjack | 0% | 0% | 20% | 65% | 85% |
| Course Section | 5% | 15% | 60% | 95% | 95% |
| Credit Card Transaction | 25% | 65% | 75% | 85% | 90% |
| Driver License | 0% | 10% | 40% | 60% | 80% |
| Hotel Stay | 25% | 65% | 85% | 85% | 85% |

If the user only generates a single state machine, based on how much they are willing to change the state machine, the previous pass rates are expected.

**Table 4: Pass@5 One-Shot Learning**

| | 0 | 0.1 | 0.2 | 0.3 | 0.5 |
|---|---|---|---|---|---|
| Blackjack | 0% | 0% | 72% | 99% | 100% |
| Course Section | 25% | 60% | 100% | 100% | 100% |
| Credit Card Transaction | 81% | 100% | 100% | 100% | 100% |
| Driver License | 0% | 45% | 95% | 100% | 100% |
| Hotel Stay | 81% | 100% | 100% | 100% | 100% |

If the user only generates five state machines, based on how much they are willing to change the state machine, the previous pass rates are expected.

**Table 5: Pass@10 One-Shot Learning**

| | 0 | 0.1 | 0.2 | 0.3 | 0.5 |
|---|---|---|---|---|---|
| Blackjack | 0% | 0% | 96% | 100% | 100% |
| Course Section | 50% | 89% | 100% | 100% | 100% |
| Credit Card Transaction | 98% | 100% | 100% | 100% | 100% |
| Driver License | 0% | 76% | 100% | 100% | 100% |
| Hotel Stay | 98% | 100% | 100% | 100% | 100% |

If the user generates ten state machines, based on how much they are willing to change the state machine, the previous pass rates are expected.

Based on the pass@k values, a different type of user may say the LLM is better at generating different systems. For example, if a user does not want to edit a state machine and wants the LLM to generate state machines until it gives a machine the fulfills all the requirements then they would say the LLM is the best at generating state machines for the systems "Hotel Stay" and "Credit Card Transaction" and the state machine is just as good at generating either while doing a much worse job generating code for the "Course Section" system.

If a user who is willing to edit and fix whatever Umple code is generated, then their experience would reflect the average normalized Levenschtein distances. They would still say the LLM is best at generating state machines for the systems "Hotel Stay" and "Credit Card Transaction" but the LLM is noticeably better at generating code for the "Credit Card Transaction" system compared to the "Hotel Stay" system. They would also say there is no noticeable difference between the LLM's ability to generate Umple code for systems "Hotel Stay" and "Course Section".

## 5.3 Semantic Analysis Using CodeBLEU

**Table 6: Semantic analysis for One-Shot Learning showing means with the highest values in bold, and standard deviations**

| | Blackjack | Course Section | Credit Card Transaction | Driver License | Hotel Stay |
|---|---|---|---|---|---|
| Average N-Gram Match Score (BLEU Score) | **0.75** σ=0.04 | **0.65** σ=0.09 | **0.86** σ=0.09 | **0.71** σ=0.22 | **0.59** σ=0.02 |
| Average Weighted N-Gram Match Score | 0.14 σ=0.09 | 0.14 σ=0.05 | 0.10 σ=0.07 | 0.03 σ=0.01 | 0.03 σ=0.00 |
| Average AST Match Score | 0.76 σ=0.03 | 0.74 σ=0.05 | 0.79 σ=0.02 | 0.63 σ=0.02 | 0.81 σ=0.08 |
| Average Dataflow Match Score | 0.18 σ=0.11 | 0.15 σ=0.05 | 0.16 σ=0.07 | 0.04 σ=0.01 | 0.03 σ=0.00 |
| Average CodeBLEU | **0.47** σ=0.06 | **0.42** σ=0.05 | **0.48** σ=0.05 | **0.35** σ=0.05 | **0.37** σ=0.04 |

We used CodeBLEU as discussed in Section 2.8. The results are presented in Table 6, using "Java" as the language. Use of Java affects all metrics except "Average N-Gram Match Score (BLEU Score)", the first row in the table; so we will primarily focus on that to determine the

effectiveness of the LLM semantically. These results are calculated by comparing the generated state machine code to the "ground truth" code created by the authors.

The semantic analysis did not match the results of the normalized Levenschtein distances. According to the BLEU Score the LLM does the best when generating Umple code for the "Credit Card Transaction" system, then the "Blackjack" system, then the "Driver License" system, then the "Course Section" system and finally the "Hotel Stay" system. Excluding the "Credit Card Transaction" system, which was considered the best for both metrics, the order of the remaining four metrics are reversed versions of each other.

This may be a coincidence but there may be an inverse relationship between the normalized Levenschtein distance and the BLEU score. This will be further explored in experiment 3 (Chapter 6).

# Chapter 6 Experiment 3: RAG

For experiment 3 we explored the effectiveness of using the RAG method. To do this we provided our LLM with different numbers of examples. These examples are the same ones used in Chapter 5.

## 6.1 Choosing Examples

To determine the examples used we compared the examples to the requirements of the system to be generated. The example contained the requirements for a system along with the "ground truth" Umple modeling code to generate its state machine. To compare them we used cosine-similarity (Section 2.6).

**Table 7: Similarity scores with requirements and code**

| Requirements | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | Credit Card Transaction | Hotel Stay | Driver License | Course Section |
| BlackJack | 0.59 | 0.57 | 0.54 | 0.47 |
| | Driver License | Hotel Stay | Credit Card Transaction | BlackJack |
| Course Section | 0.66 | 0.64 | 0.61 | 0.51 |
| | Driver License | Hotel Stay | BlackJack | Course Section |
| Credit Card Transaction | 0.63 | 0.60 | 0.57 | 0.55 |
| | Credit Card Transaction | Course Section | Hotel Stay | BlackJack |
| Driver License | 0.63 | 0.59 | 0.55 | 0.53 |
| | Credit Card Transaction | BlackJack | Course Section | Driver License |
| Hotel Stay | 0.57 | 0.54 | 0.53 | 0.50 |

When providing "X" number of examples, we will go in order of similarity scores in descending order. This means if we want RAG to generate a state machine for the "Blackjack" system and give it two examples, the examples we will use are "Credit Card Transaction" and "Hotel Stay" as these are the two most similar examples.

The average similarity scores of the examples from highest to lowest are:

- Credit Card Transaction – 0.60
- Hotel Stay – 0.59
- Driver License – 0.58

- Blackjack – 0.54
- Course Section – 0.53

These imply the "Credit Card Transaction" example is the most relevant example on average.

**Table 8: Similarity scores with only requirements, ranked left-to-right from most to least similar) Highlighted cells changed order as compared to when code was also provided**

| Requirements | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| BlackJack | Credit Card Transaction | Hotel Stay | Driver License | Course Section |
| | 0.59 | 0.56 | 0.54 | 0.51 |
| Course Section | Driver License | Hotel Stay | Credit Card Transaction | BlackJack |
| | 0.62 | 0.59 | 0.58 | 0.49 |
| Credit Card Transaction | Driver License | Hotel Stay | BlackJack | Course Section |
| | 0.68 | 0.61 | 0.60 | 0.60 |
| Driver License | Credit Card Transaction | Course Section | BlackJack | Hotel Stay |
| | 0.68 | 0.64 | 0.55 | 0.54 |
| Hotel Stay | Credit Card Transaction | Course Section | BlackJack | Driver License |
| | 0.58 | 0.56 | 0.55 | 0.51 |

Though not commonly done in with RAGs, we experimented with comparing the requirements of the system to be generated with only the requirements of the example system. The results of this can be seen in Table 8. This way we would only be comparing requirements to requirements instead of comparing requirements to requirements and code as we did previously. We would then still provide requirements as well as the code to the RAG as input.

The average similarity scores from highest to lowest are:

- Credit Card Transaction – 0.61
- Driver License – 0.59
- Course Section – 0.58
- Hotel Stay – 0.57
- Blackjack – 0.55

Though this had a large influence on the order of the average similarity score, they stayed in the same general range of ~0.5 – ~0.6. The only use of the similarity scores was to determine which examples would be provided to the RAG. As seen in Table 8 the order of the systems did not change very much. The only changes were the two sets of systems highlighted, which switched positions.

This change did not have a large effect on what we were going to do, along with being something not commonly done in a RAG. This led us to the decision of not moving forward with this change and sticking to the original idea of using an example's requirements and code when comparing it using cosine similarity.

## 6.2 Syntactic Analysis

We asked the RAG to generate each of the systems using a single example, two examples, three examples, and four examples. We conducted five iterations of each system with each number of examples. Table 9 shows the average results for all 100 iterations.

**Table 9: Average ICP, EUCP, Normalized Levenschtein Distance and AF for each system using RAG**

| Metrics | Blackjack | Course Section | Credit Card Transaction | Driver License | Hotel Stay |
|---|---|---|---|---|---|
| ICP | 0% σ=0.03 | 0% σ=0.00 | 0% σ=0.00 | 0% σ=0.01 | 0% σ=0.00 |
| EUCP | 2% σ=0.02 | 0% σ=0.01 | 0% σ=0.01 | 0% σ=0.00 | 0% σ=0.00 |
| Normalized Levenschtein Distance | 0.31 σ=0.15 | 0.31 σ=0.11 | 0.25 σ=0.26 | 0.32 σ=0.13 | 0.07 σ=0.09 |
| AF | 0.70 σ=0.50 | 0.35 σ=0.48 | 0.50 σ=0.50 | 0.25 σ=0.43 | 0.35 σ=0.48 |

The ICP and EUCP was very good and much higher than in zero shot learning. The ICP was nearly the same as one-shot learning. Like one-shot learning, it was rare for a machine to have any code causing it to fail compilation. There is also a significant decrease in extraneous code compared to both zero-shot and one-shot learning. These reductions are from the LLM having a larger pool of examples which gave the LLM a better understand of Umple syntax, decreasing the amount of invalid and extraneous code created by these syntax errors.

Along with a reduction in ICP and EUCP, there is a reduction in AF. This shows that as the LLM is provided more examples it is less likely to attempt to add new features to the state machine it is generating.

Compared to one-shot learning the best state machines and the worst state machines significantly changed. The RAG is best at generating state machines for the "Hotel Stay" with an average normalized Levenschtein distance of 0.07. This was expected to be one of the lowest distances as it was simply a set of instructions of what the user can do, along with an optional login feature. This normalized Levenschtein distance is a large improvement over the 0.17 seen when using one-shot learning. We did not expect AF to decrease for the "Hotel Stay" state machine because even though we understand additional examples make the state machine more "rigid" we expected AF for hotel stay to still increase as the additional features that could be added to the "Hotel Stay" system are simple.

The system with the second lowest normalized Levenschtein distance is "Credit Card Transaction". This is the best system when using one-shot learning. Its normalized Levenschtein distance is much higher compared to one-shot learning where it is 0.13 while with RAG it is 0.25. This time the LLM generated multiple machines that required a high number of changes some with normalized Levenschtein distances going up to 0.89.

The third lowest state machine is "Course Section" with an average normalized Levenschtein distance of 0.31. This is another large increase from the 0.18 seen when using one-shot learning. This state machine's average normalized Levenschtein distance is very close to the second worst state machine's normalized Levenschtein distance when doing one-shot learning (the "Driver License" system).

The system with the second highest normalized Levenschtein distance is "Blackjack". It has a normalized Levenschtein distance of 0.31, a significant decrease from its 0.46 normalized Levenschtein distance when using one-shot learning. This is surprising now because it is the worst performing system when using one-shot learning, but this was what we originally expected before conducting the experiments as we assumed the LLM would have a lot of knowledge of the rules of blackjack due to how popular the game is. The LLM still struggled at fulfilling all the requirements, but correcting the state machines required fewer changes compared to the state machines created by using one-shot learning. Though this is the same value as the "Course

Section" system when using 2 significant figures, the "Blackjack" system's normalized Levenschtein distance is 0.315 while the "Course Section" system's is 0.307.

The worst state machine is "Driver License". It has an average normalized Levenschtein distance of 0.32 while is a slight increase compared to the normalized Levenschtein distance seen in one-shot learning of 0.29. This is a unique system as the rules are unique to Ontario. The LLM always did a poor job with this system using any method.

## 6.3 Pass@K Analysis

**Table 10: Pass@1 RAG Learning**

| | 0 | 0.1 | 0.2 | 0.3 | 0.5 |
|---|---|---|---|---|---|
| Blackjack | 0% | 15% | 40% | 50% | 75% |
| Course Section | 0% | 0% | 25% | 50% | 95% |
| Credit Card Transaction | 25% | 40% | 55% | 70% | 80% |
| Driver License | 0% | 10% | 10% | 45% | 90% |
| Hotel Stay | 30% | 75% | 85% | 95% | 95% |

If the user only generates a single state machine, based on how much they are willing to change the state machine, the previous pass rates are expected.

**Table 11: Pass@5 RAG Learning**

| | 0 | 0.1 | 0.2 | 0.3 | 0.5 |
|---|---|---|---|---|---|
| Blackjack | 0% | 60% | 95% | 98% | 100% |
| Course Section | 0% | 0% | 81% | 98% | 100% |
| Credit Card Transaction | 81% | 95% | 99% | 100% | 100% |
| Driver License | 0% | 45% | 45% | 97% | 100% |
| Hotel Stay | 87% | 100% | 100% | 100% | 100% |

If the user only generates five state machines, based on how much they are willing to change the state machine, the previous pass rates are expected.

**Table 12: Pass@10 RAG Learning**

| | 0 | 0.1 | 0.2 | 0.3 | 0.5 |
|---|---|---|---|---|---|
| Blackjack | 0% | 89% | 100% | 100% | 100% |
| Course Section | 0% | 0% | 98% | 100% | 100% |
| Credit Card Transaction | 98% | 100% | 100% | 100% | 100% |
| Driver License | 0% | 76% | 76% | 100% | 100% |
| Hotel Stay | 99% | 100% | 100% | 100% | 100% |

If the user generates ten state machines, based on how much they are willing to change the state machine, the previous pass rates are expected.

Like the one-shot example, user's judgements about the RAG's ability to generate different systems will differ. For example, a user who does not want to edit any of the state machines will judge the RAG's ability to generate the "Hotel Stay" and "Credit Card Transaction" as equally good and will judge the RAG's ability to generate the remaining systems as equally bad. While a user who will edit and fix any system generated will judge the RAG's abilities according to the normalized Levenschtein distances in Table 9.

## 6.4 Semantic Analysis

**Table 13: Semantic analysis for RAG showing means with the highest values in bold, and standard deviations**

| | Blackjack | Course Section | Credit Card Transaction | Driver License | Hotel Stay |
|---|---|---|---|---|---|
| Average N-Gram Match Score (BLEU Score) | **0.74** σ=0.17 | **0.60** σ=0.15 | **0.84** σ=0.12 | **0.65** σ=0.12 | **0.54** σ=0.18 |
| Average Weighted N-Gram Match Score | 0.19 σ=0.09 | 0.08 σ=0.04 | 0.05 σ=0.05 | 0.03 σ=0.01 | 0.02 σ=0.01 |
| Average Syntax Match Score | 0.76 σ=0.04 | 0.76 σ=0.07 | 0.80 σ=0.04 | 0.64 σ=0.02 | 0.82 σ=0.05 |
| Average Dataflow Match Score | 0.23 σ=0.11 | 0.09 σ=0.4 | 0.07 σ=0.05 | 0.04 σ=0.03 | 0.03 σ=0.02 |
| Average CodeBLEU | **0.47** σ=0.07 | **0.38** σ=0.05 | **0.44** σ=0.05 | **0.34** σ=0.03 | **0.35** σ=0.05 |

Like one-shot learning, CodeBLEU is being calculated using “Java” as the language. This affects all metrics except “Average N-Gram Match Score” or BLEU Score.

The semantic analysis did not match the results of the normalized Levenschtein distances. According to the BLEU Score the LLM does the best when generating Umple code for the “Credit Card Transaction” system, then the “Blackjack” system, then the “Driver License” system, then the “Course Section” system and finally the “Hotel Stay” system. This is the exact same order seen in one-shot learning. Therefore, this shows the BLEU score has no correlation with the normalized Levenschtein distance.

**Table 14: BLEU Score for RAG by number of examples**

| | Blackjack | Course Section | Credit Card Transaction | Driver License | Hotel Stay |
|---|---|---|---|---|---|
| 1+ Examples | 0.74 | 0.60 | 0.84 | 0.65 | 0.54 |
| 2+ Examples | 0.76 | 0.57 | 0.82 | 0.67 | 0.67 |
| 3+ Examples | 0.77 | 0.57 | 0.81 | 0.77 | 0.58 |
| All Examples | 0.74 | 0.57 | 0.86 | 0.81 | 0.57 |

When separating the BLEU scores by the number of examples there was slight variation in the order of the systems’ BLEU scores, but this is likely a coincidence as the alternatives are not closer the order of the distance values.

# Chapter 7 Conclusion and future Work

## 7.1 Answers to Research Questions

Research question 1 asks, “to what extent can LLMs use FMs to generate Umple state machine modeling code without additional information”? This is tested in Experiment 1, it was shown the LLM can not generate Umple state machine modeling code to any useful level. All state machines generated had little to no similarity to Umple modelling code and it was clear the LLM did not have a grasp on the language.

Research question 2 asks, “To what extent can LLMs use FMs to generate Umple state machine modeling code using a single example”? This is tested in Experiment 2. For normalized Levenschtein distance the average values obtained for each system ranged from 0.13 to 0.46. Four out of five of the systems had a normalized Levenschtein distance value below 0.30. The LLM is also only able to generate state machines that meet all requirements for three out of five of the systems. These results mean different types of users will see this as a success or failure. Overall, we believe the answer to this research question is this will save users’ time.

Research question 3 asks, “To what extent can LLMs use FMs to generate Umple state machine modeling code using a RAG based solution?”? This is tested in Experiment 3. For normalized Levenschtein distance the average values obtained for each system ranged from 0.07 to 0.32. The LLM is also only able to generate state machines that meet all requirements for two out of five of the systems. Like one-shot learning different users will weigh these results differently but based on our goals, an LLM can save a user time when generating Umple state machines using a RAG based solution.

Overall, the average normalized Levenschtein distances of the one-shot method and RAG are very close to each other. They are 0.246 and 0.244 respectively.On average the RAG based solution had lower variance in with average normalized Levenschtein distances for each of the systems. This leads us to believe that using a RAG with multiple examples is more consistent. Though the RAG is more consistent compared to one-shot learning it is not consistent with its performance based on the number of examples. For example, the “Blackjack” system performs best when using one example and four examples but performs worse when using two or three

examples. While for the “Driver License system”, there is a noticeable decrease in the normalized Levenschtein distances as the number of examples provided increases.

## 7.2 Summary of Contributions

Using AI to generate state machine has not been thoroughly analyzed. We have devised a method of comparing a generated state machine to its corrected version using the normalized Levenschtein distance metric. This allows for the measure of the relative amount of effort needed for a user to correct a generated state machine.

Along with this we have shown that Umple state machine code can be generated using even a small LLM model when combined with common techniques such as one-shot learning and RAG.

We have shown that there is no connection between the BLEU score and the quality of a state machine.

## 7.3 Threats to Validity and How We Addressed the Threats

These reference state machines are not a perfect solution because the person who created the example that the state machine uses is also the person who created a corrected version of the machine, and this will not be the case for users. Ideally, multiple users should reference state machines and average the results. To address this, we added state machines not created by the primary author such as the course section machine. Another thing we do to address this is when calculating normalized Levenschtein distance we compare the generated state machine to a corrected version of it, instead of the ground truth state machine.

The version of Llama 3.1 is the smallest available version of the LLM (8 billion parameter version). This version of the model performs worse compared to larger version of Llama 3.1. To address this, we did not use Llama 3.1 alone, we combined it with other prompting techniques such as one-shot learning and RAG. These methods have a large impact on the model’s performance.

Foundation models are constantly developing. Llama 3 was one of the best FMs when this thesis was started, but there are likely to be newer models that give different results, including models better at generating state machines.

As stated in Section 3.4.2, the “Course Section” state machine is from the UmpleOnline website, therefore Llama 3 has likely seen its code before. This may bias its results when the LLM tries to generate a state machine for it, as well as when it is used as an example. However, its requirements, along with all other requirements and state machines were created for this thesis and have not been sourced from elsewhere; therefore Llama 3 will not have at least seen the requirements for the Course Section state machine before.

## 7.4 Future Work and Recommendations

The findings in this thesis can be expanded on through multiple avenues.

As mentioned in section 7.3 the model used is the smallest version of Llama 3.1. This model is the 8B model and performs worse than the other sizes such as the 70B and 405B. The experiments done in this thesis should be repeated with a larger model to see if results improve.

In this thesis we are not able to use CodeBLEU to its full potential because it does not have compatibility with Umple, as an Umple-specific version would need to be created that understands Umple keywords as well as implementing an Umple parser specific to CodeBLEU. For future work it would be interesting to see how CodeBLEU compares to the results of normalized Levenschtein distance as Bleu score did not correlate to it.

The modeling language “Mermaid” is a popular language for generating multiple types of diagrams including state machines. The official website “mermaidchat.com” has an AI tool that lets a user generate mermaid code based on a prompt (Sveidqvist, 2024). The experiments in this thesis can be repeated using their tool with the Mermaid modeling language and compare results.

This thesis used simple examples with few requirements. It would be interesting to see how well a LLM performs with more complex examples or requirements. Perhaps an LLM is better at generating state machines for simple systems when provided examples of simple systems, or maybe the LLM learns more from complex systems therefore improves when provided examples of state machines of complex systems, even when generating state machines for simple systems.

# References


Adler-Nissen, R. and Drieschova, A. (2019) 'Track-Change Diplomacy: Technology, Affordances, and the Practice of International Negotiations', *International Studies Quarterly*, 63(3), pp. 531–545. doi:10.1093/ISQ/SQZ030.

Awais, M., Naseer, M., Khan, S., Anwar, R., Cholakkal, H., Shah, M., Yang, M., Khan, F. (2025) 'Foundation Models Defining a New Era in Vision: A Survey and Outlook', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 47(4), pp. 2245-2264. doi:10.1109/TPAMI.2024.3506283.

Brown, T., Mann, B., and Ryder, N. et al. (2020) 'Language Models are Few-Shot Learners', *Advances in Neural Information Processing Systems*, 33. arXiv:2005.14165v4.

Chen, M., Tworek, J., Jun, H. et al. (2021) 'Evaluating Large Language Models Trained on Code', *arXiv*. arXiv:2107.03374v2.

Corso, V., Mariani, L., Micucci, D., Riganelli, O. (2024) 'Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants', *ICPC '24: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 32, pp. 13-23. doi:10.1145/3643916.3644402.

David, E. (2024) 'OpenAI can't register 'GPT' as a trademark — yet', *The Verge*. Available at: https://www.theverge.com/2024/2/16/24075304/trademark-pto-openai-gpt-deny.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., Wang, H. et al. (2024) 'Retrieval-Augmented Generation for Large Language Models: A Survey', *arXiv*. arXiv:2312.10997v5.

Grattafiori, A., Dubey, A., Jauhri, A. et al. (2024) 'The Llama 3 Herd of Models', *arXiv*. arXiv:2407.21783v3.

Han, J., Kamber, M., Pei, J. (2012) 'Data Mining: Concepts and Techniques', *Data Mining: Concepts and Techniques*, 3, pp. 39-82. doi:10.1016/B978-0-12-381479-1.00002-2.

Harel, D (1987) 'Statecharts: a visual formalism for complex systems', *Science of Computer Programming*, 8(3), pp. 231-274. doi: 10.1016/0167-6423(87)90035-9.

Hou, X., Zhao, Y., Liu, Y. et al. (2024) 'Large Language Models for Software Engineering: A Systematic Literature Review', *ACM Transactions on Software Engineering and Methodology*, 33(8), pp. 1-79. doi:10.1145/3695988.

Hey, T., Keim, J., Koziolek, A., Tichy, W. (2020) 'NoRBERT: Transfer learning for requirements classification', *Karlsruher Institut für Technologie (KIT)*, 28, pp. 169-179. doi:10.1109/RE48521.2020.00028.

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L. et al. (2018) 'Mapping Language to Code in Programmatic Context', *Association for Computational Linguistics*, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 1643–1652. doi:10.18653/v1/D18-1192.

Lethbridge, T., Forward, A., Badreddin, O. et al. (2021) 'Umple: Model-driven development for open source and education', Science of Computer Programming, 208. doi:10.1016/j.scico.2021.102665.

Lewis, P., Perez, E., Piktus, A. et al. (2020) 'Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks', *Advances in Neural Information Processing Systems*, 33. arXiv:2005.11401v4.

Li, Y., Choi, D., Chung, J. et al. (2022) 'Competition-Level Code Generation with AlphaCode', *Science*, 378(6624), pp. 1092-1097. doi:10.1126/science.abq1158.

Liu, J., Xia, C., Wang, Y., Zhang, L. (2023) 'Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation', *Advances in Neural Information Processing Systems*, 36. arXiv:2305.01210v3.

Liu, Z., Tang, Y., Luo, X., Zhou, Y., Zhang, L. (2024) 'No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT', *IEEE Transactions on Software Engineering*, 50(6), pp 1548-1584. doi:10.1109/TSE.2024.3392499.

Merseguer, J., Campos, J., Bernardi, S., Donatelli, S. (2002) 'A compositional semantics for UML state machines aimed at performance evaluation', *Sixth International Workshop on Discrete Event Systems*, 6, pp. 295-302. doi:10.1109/WODES.2002.1167702.

Meta (2024) 'Llama 3 | Model Cards & Prompt formats', *Llama*. Available at: https://www.llama.com/docs/model-cards-and-prompt-formats/meta-llama-3/.

Mikolov, T., Chen, K., Corrado, G., Dean, J. (2013) 'Efficient Estimation of Word Representations in Vector Space', *1st International Conference on Learning Representations*, 1. arXiv:1301.3781v3.

Millam, A. and Bakke, C. (2024) 'Coding with AI as an Assistant: Can AI Generate Concise Computer Code?', *Journal of Information Technology Education: Innovations in Practice*, 23, pp. 9. doi:10.28945/5362.

Muennighoff, N., Tazi, N., Magne, L., Reimers, N. (2023) 'MTBE: Massive Text Embedding Benchmark', *Association for Computational Linguistics*, 17, pp. 2014–2037. doi: 10.18653/v1/2023.eacl-main.148.

Nussbaum, Z., Morris, J., Duderstadt, B., Mulyar, A. (2024) 'Nomic Embed: Training a Reproducible Long Context Text Embedder', *arXiv*. arXiv:2402.01613v2.

Odu, O., Belle, A., Wang, S., Kpodjedo, S., Lethbridge, T., Hemmati, H. (2024) 'Automatic Instantiation of Assurance Cases from Patterns Using Large Language Models', *Journal of Systems and Software*, 222. doi: 10.1016/j.jss.2025.112353.

OpenAI (2025) *Tokenizer*. Available at: https://platform.openai.com/tokenizer (Accessed: 13 May 2025).

Papineni, K., Roukos, S., Ward, T., Zhu, W. (2002) 'BLEU: a method for automatic evaluation of machine translation', ACL '02: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistic*, 40, pp. 311-318. doi:10.3115/1073083.1073135.

Rabbi, F., Champa, A., Zibran, M., Islam, R. (2024) 'AI Writes, We Analyze: The ChatGPT Python Code Saga', *IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, 21, pp. 177-181. doi:10.1145/3643991.3645076.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S. (2020) 'CodeBLEU: a Method for Automatic Evaluation of Code Synthesis', *arXiv*. arXiv:2009.10297v2.

Saad-Falcon, J., Fu, D., Arora, S., Guha, N., Ré, C. (2024) 'Benchmarking and Building Long-Context Retrieval Models with LoCo and M2-BERT', *ICML'24: Proceedings of the 41st International Conference on Machine Learning*, 41(1749), pp. 42918-42946. doi:10.5555/3692070.3693819.

Shehata, M., Lepore, B., Cummings, H., Parra, E. (2024) 'Creating UML Class Diagrams with General-Purpose LLMs', *2024 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 157-158. doi:10.1109/VISSOFT64034.2024.00031.

Shen, Z., Tao, T., Neiswanger, W. et al. (2024) 'SlimPajama-DC: Understanding Data Combinations for LLM Training', *arXiv*, arXiv:2309.10818v3.

Sveidqvist, K. (2024) 'Mermaid AI Is Here to Change the Game For Diagram Creation', *Mermaid Chart*. Available at: https://docs.mermaidchart.com/blog/posts/mermaid-ai-is-here-to-change-the-game-for-diagram-creation.

Tang, N., Chen, M., Ning, Z., Bansal, A., Huang, Y., McMillan, C., Li, T. (2023) 'An Empirical Study of Developer Behaviours for Validating and Repairing AI-Generated Code', *13th Workshop on the Intersection of HCI and PL*. arXiv:2405.16081.

Touvron, H., Martin, L., Stone, K. et al. (2023) 'Llama 2: Open Foundation and Fine-Tuned Chat Models', *arXiv*. arXiv:2307.09288v2.

Umple (2025) *User manual*. Available at: https://cruise.umple.org/umple/GettingStarted.html (Accessed: 13 May 2025).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I. (2017) 'Attention is All you Need', *Advances in Neural Information Processing Systems*, 31, pp. 6000 – 6010. arXiv:1706.03762v7.

Yenduri, G., Murugan, R., Govardanan, C. et al. (2024) 'Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies', *Potential Applications, Emerging Challenges, and Future Directions*. IEEE, 12, pp. 54608-54649. doi:10.1109/ACCESS.2024.3389497.

Wang, L., Yang, N., Huang, X., Yang, L., Yang, L., Majumder, R., Wei, F. (2024) 'Improving Text Embeddings with Large Language Models', Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 1, pp. 11897–11916. doi:10.18653/v1/2024.acl-long.642