CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Automated Statechart Generation from Natural Language Requirements Using AI Techniques in Automotive Software Engineering

Master's Thesis in Software Engineering and Technology

Lakshmi Sri Rupa Kurukuri

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Automated Statechart Generation from Natural Language Requirements Using AI Techniques in Automotive Software Engineering

Lakshmi Sri Rupa Kurukuri

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Automated Statechart Generation from Natural Language Requirements Using AI Techniques in Automotive Software Engineering
Lakshmi Sri Rupa Kurukuri

Supervisor: Farnaz Fotrousi, Department of Computer Science and Engineering
Examiner: Jennifer Horkoff, Department of Computer Science and Engineering
Examiner-in-Practice: Khan Mohammad Habibhullah, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Automated Statechart Generation from Natural Language Requirements Using AI Techniques in Automotive Software Engineering
Lakshmi Sri Rupa Kurukuri
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Context: Statecharts are widely recognized as an effective modeling technique for describing system behaviors, allowing engineers to clearly represent states and transitions. However, the process of creating statecharts from textual requirements remains largely manual, requiring engineers' expertise to interpret textual descriptions, define states and transitions, and identify potential errors. As systems grow increasingly complex, this approach faces challenges in scaling, leading to inefficiencies, inconsistencies, and delays in the design and validation phases.

Problem: Existing tools such as Stateflow, Yakindu, and Enterprise Architect, although useful for statechart modeling, still require significant manual effort. These tools are not designed to efficiently generate statecharts from unstructured textual input, which creates a challenge in maintaining the flexibility needed in rapidly evolving industrial settings. Despite advancements in large language models (LLMs), their potential for automating statechart generation has not been fully explored or used in real-world contexts, with a focus on the automotive industry.

Solution: This thesis proposes a framework that takes advantage of Artificial Intelligence (AI), specifically Natural Language Processing (NLP) and Transformer-based models, to automatically generate statecharts from textual requirements. The approach integrates pre-trained language models with fine-tuned domain-specific data, enabling the identification of states, transitions, and the generation of valid statecharts directly from natural language input. This framework represents a significant step towards automating the statechart creation process, making it more efficient and reliable.

Results and Contribution: The results of this research show that fine-tuning LLMs on domain-specific data significantly improves the quality of the LLM generated statecharts in terms of functional correctness and understandability. By automating statechart generation, this work improves productivity, minimizes human error, and offers a scalable solution for the automotive industry. The proposed approach enhances design and validation workflows, enabling faster and more accurate system development in critical domains such as automotive engineering.

Keywords: Statechart Generation, Automated Model Generation, Artificial Intelligence, Natural Language Processing (NLP), Large Language Models (LLMs), Automotive Software Engineering

Acknowledgements

I would like to express my sincere gratitude to the university supervisor, Farnaz Fotrousi, for her invaluable guidance, encouragement, and support throughout this research. I am also deeply grateful to the industrial supervisors at Volvo Cars Corporation, Martin Hedström and Mohammad Reza Haghshenas, for their invaluable guidance, continuous support, and insightful feedback throughout this research. Their expertise and encouragement have played a crucial role in shaping this work, both academically and practically. Furthermore, I extend my appreciation to the examiner, Jennifer Horkoff, and examiner-in-practice, Khan Mohammad Habibhullah, for their time, constructive evaluation, and thoughtful suggestions, which have helped enhance the quality of this study. Lastly, I would like to extend my appreciation to my colleagues, peers, and family for their constant support and encouragement during this journey.

Lakshmi Sri Rupa Kurukuri, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

Statecharts are essential tools in software and systems engineering, enabling a structured approach to modeling and validating the behavior of dynamic systems [16]. Statecharts, particularly UML statecharts, serve as a crucial tool in this workflow by providing a clear visual representation of system behaviors [1]. This workflow typically involves interpreting the textual system requirements to identify states and transitions by manually creating statecharts using specialized modeling tools. These models bridge the gap between textual requirements and executable designs, enabling effective communication among stakeholders and facilitating early validation of functional concepts [14]. In industries such as automotive engineering, where safety and functionality requirements are critical, statecharts play an important role in ensuring system reliability [14]. However, manual creation of statecharts from textual requirements is not only time-intensive but also prone to human error and heavily reliant on domain-specific expertise [6]. This challenge is particularly evident in the automotive industry, which is currently experiencing significant changes due to a shift towards advanced technologies that streamline development processes and enhance product quality. These challenges highlight the need for efficient and automated solutions to address the increasing complexity of system design and validation.

At Volvo Cars, a global leader in automotive innovation, statecharts are extensively used for modeling and validating functionalities across systems. The product simulation team within the product validation & function architecture department plays an important role in ensuring robust system functionality through early testing and validation. A critical aspect of this process is the ability to validate product functions at every stage of a vehicle's lifecycle, from requirement definition to over-the-air (OTA) software updates [7].

1.1 Problem Description

Engineers currently interpret textual requirements and manually create statecharts using specialized tools. As systems grow more complex and interconnected, this manual approach struggles to scale efficiently, leading to inconsistencies, delays, and increased validation efforts. Automating this process holds the potential to signifi-

cantly enhance productivity, reduce costs, and ensure consistency in critical system design and validation [26].

In the context of automated statechart generation from textual requirements, the challenge arises from the manual and often error-prone steps involved in the current process at Volvo Cars. Typically, textual requirements are provided by the product function owner through the Car Weaver Inbox, followed by analysis from system engineers to manually identify states, transitions, events, and conditions. This process is time-consuming, requires expertise, and is subject to human error. The current practice involves the use of IBM Rhapsody, a modeling tool, to define and generate statecharts based on these manual inputs, which can delay product development and increase the risk of inconsistencies between requirements and the final model.

1.2 Purpose of the Study

This thesis aims to develop an AI/ML-based solution capable of automating statechart generation from textual requirements. Using advanced Natural Language Processing (NLP) techniques and the capabilities of Large Language Models (LLMs) such as GPT (Generative Pretrained Transformer) [29], the solution aims to analyze unstructured inputs and generate accurate, functional statecharts. The automation of this process is expected to minimize human error and enable stakeholders, even those without specialized expertise, to efficiently generate good-quality statecharts. This capability is anticipated to significantly enhance workflows within Volvo Cars' Product Simulation team.

Recent advancements in LLMs have demonstrated their potential for automating model generation, despite challenges related to ambiguity and requirement inconsistencies [13]. Building on these developments, this research will customize an AI/ML solution to integrate automated statechart generation into Volvo Cars' existing validation and simulation frameworks, furthering their long-term goals of enhancing validation, code generation, and testing efficiency. Additionally, the study will examine how AI can assist in validating statecharts against requirements to ensure functional correctness and completeness. Key evaluation metrics will include time to creation, requirement coverage, and comparison with manually created statecharts, with the goal of enhancing productivity and reducing development time within automotive software engineering workflows.

1.3 Significance of the Study

This work is expected to hold significant value for both academia and the automotive industry. By automating statechart creation, the solution addresses a key bottleneck in validation workflows (such as testing and validating system behav-

ior), potentially enabling faster iterations, improved consistency, and more reliable outcomes. Moreover, it aligns with industry efforts to develop more efficient and scalable validation processes that can scale with increasing system complexity and evolving requirements, ultimately enabling innovation and accelerating the development of high-quality automotive systems.

Aim and Objectives:
The aim of this research is to develop and evaluate an AI/ML solution that uses large language models (LLMs) for generating statecharts from unstructured textual requirements.

- Capability Assessment of LLMs: This objective focuses on evaluating the capabilities of large language models in generating statecharts. The performance of LLM-generated statecharts will be assessed in comparison to those created manually by domain experts. This comparison will analyze understandability, functional correctness, and alignment with industry standards, providing insight into the feasibility of AI-driven statechart generation in industrial workflows.

- Development and Evaluation: This objective involves designing and implementing a modular AI/ML solution using advanced natural language processing (NLP) techniques to process unstructured textual requirements and automatically generate statecharts. Performance will be evaluated through a comparative analysis, where domain experts or human evaluators will assess its understandability, accuracy, usability, and alignment with established standards, ensuring the quality and applicability of the AI-generated outputs.

1.4 Thesis Outline

Following this Introduction chapter, the thesis is further structured into six chapters, each addressing different aspects of the research, from background and theoretical foundations to methodology, results, and conclusions.

- Chapter 2 – Background: It deals with fundamental concepts related to statecharts, natural language processing (NLP), large language models (LLMs), and Model-Based Systems Engineering (MBSE)[9]. It provides the theoretical basis that is needed for proper comprehension of the research.

- Chapter 3 – Related Work: This chapter studies different existing research and tools related to automated statechart generation, NLP-based requirement analysis, and AI-driven modeling approaches. It identifies research gaps and positions this study within the context of existing research literature.

- Chapter 4 - Methods: Details the design of the AI/ML solution, including preprocessing of textual requirements, fine-tuning large language models, methods for extracting states, and evaluation metrics. The chapter also describes the

dataset upon which training and validation rely.

- Chapter 5 - Results: Presents experimental results that compare AI-generated statecharts with those created by humans in terms of accuracy, efficiency, and usability. These results consist of both quantitative measures and feedback from domain experts.

- Chapter 6 - Discussion: Analyzes results, interprets their implications, discusses their limitations, and evaluates the feasibility of AI-generated statecharts in real-world automobile validation workflows.

- Chapter 7 - Conclusion: Summarizes key findings, highlights contributions to AI-assisted software modeling, and outlines directions for future research, including improving statechart correctness, enhancing explainability, and integrating industry frameworks with AI-driven validation.

2

Background

2.1 Statechart Diagrams in Software Engineering

Statecharts are a fundamental tool in software and systems engineering, particularly in industries where modeling dynamic behaviors is critical. These include sectors such as automotive, aerospace, and embedded systems. Statecharts, first introduced by David Harel [16], extend traditional finite state machines with additional features like hierarchy, concurrency, and event-driven transitions, making them an effective mechanism for representing complex system behavior. While statecharts are widely used for modeling and validating simulations in various domains, their application is notably significant in the automotive industry, where they are used to model functions such as powertrain controls, infotainment systems, Advanced Driver Assistance Systems (ADAS), and autonomous vehicle algorithms [22].

2.2 Why Focus on Statechart Generation for Automotive Software

Although statecharts are a versatile tool used across different industries, the primary focus of this research is on automotive software. This is due to the increasingly complex nature of automotive systems, which rely heavily on dynamic, event-driven behaviors. Automotive software typically requires a high degree of interaction between subsystems, such as safety-critical systems (e.g., ADAS) and infotainment. Unlike static diagram types such as use case or class diagrams, statecharts are particularly suited for representing these dynamic, time-dependent behaviors and system transitions. As these systems grow in complexity, the need for more automated, accurate, and scalable modeling methods becomes essential. While statecharts could theoretically be applied to other types of software, the automotive domain presents unique challenges, such as strict safety requirements, real-time validation needs, and the integration of numerous interconnected systems. These challenges make other types of diagrams less effective in capturing the full scope of dynamic behaviors and transitions within the system. Hence, this research aims to streamline the statechart generation process specifically for automotive software, providing a solution customized to its complexity, operational requirements, and the need for real-time,

validated, and executable models.

2.3 Statechart Tools and the Challenges of Manual Creation

In the automotive sector, the increasing complexity of modern vehicles has driven the widespread adoption of model-based design tools, such as MathWorks Stateflow [25], Yakindu Statechart Tools [2], and Enterprise Architect [33], which facilitate statechart modeling. These tools provide essential functionalities like simulation and validation of real-world conditions. For instance, Stateflow integrates with MATLAB/Simulink, enabling engineers to model system behaviors and simulate these models under various real-world scenarios. Yakindu Statechart Tools provide a graphical interface for designing statecharts, along with integrated validation and simulation features. Despite these advancements, the manual creation of statecharts remains a significant challenge, requiring considerable time, expertise, and effort, especially as system designs become complex. Moreover, these tools often fail to address fundamental challenges such as handling incomplete requirements and generating executable statecharts [27].

2.4 The Need for Automation and AI Integration

Manual statechart development presents additional challenges. Engineers must interpret textual requirements, define states and transitions, and ensure logical correctness. However, this process is prone to human error, inconsistency, and significant delays, particularly as software functionalities in vehicles become more complex [26]. One of the critical challenges is the dependency on domain experts. Requirements engineers, system architects, and validation teams are typically not experts in statechart modeling. Consequently, only a few specialized experts are capable of efficiently translating textual requirements into executable models, which often leads to delays in the development cycle. Furthermore, manually created statecharts may differ in structure and completeness, making validation and debugging more difficult [7].

To address these issues, Artificial Intelligence (AI) and Natural Language Processing (NLP) have shown great promise in automating model generation. NLP, a subset of AI, enables machines to process and derive structured data from unstructured textual input [26]. Recent advances in Large Language Models(LLMs) like GPT-4 [29] have demonstrated the potential to process complex textual inputs and generate outputs such as UML class diagrams and statecharts [1]. By training these models on customized datasets, AI systems can automatically generate statecharts from textual descriptions, reducing manual effort, enhancing consistency, and minimizing errors.

In addition to automating statechart generation, validating the generated models is essential to ensure they meet design specifications. Validation is a critical step in ensuring the reliability and effectiveness of the statecharts in practical applications. In the automotive industry, companies like Volvo Cars use statecharts extensively for early-stage testing and validation of system behaviors. For instance, Volvo's product simulation team uses statecharts to ensure that the system's behavior conforms to safety and performance requirements throughout the end-to-end vehicle lifecycle, from requirements definition to over-the-air (OTA) software updates.

3

Related Work

The automation of statechart generation from textual requirements is a rapidly growing research area that brings together advancements in software modeling, artificial intelligence (AI), and natural language processing (NLP). Researchers have explored various ways to automate diagram generation, requirement analysis, and model validation, but many challenges remain. This section reviews key studies that have contributed to this field, highlighting their approaches, limitations, and how this thesis builds upon their work.

3.1 Automation of UML Diagram Generation

Current research has primarily focused on automating easier diagrammatic models, such as Unified Modeling Language (UML) class diagrams through natural language processing (NLP) and deep learning approaches to convert text specifications into structured representations [3]. For instance, Meng and Ban proposed an NLP-based model that analyzes textual requirements to extract relevant information for generating UML class diagrams, achieving a classification accuracy of 88.46% [26]. However, these methods are limited to structural diagrams, which do not fully capture dynamic behaviors and system interactions.

3.2 Challenges in Automating Statechart Generation

Statechart automation remains an under-explored area compared to UML diagram generation, particularly in both academia and industry. While there has been some exploration into automating statechart generation, the complexities involved in modeling dynamic, event-driven systems are significant. One of the significant challenges in automating statechart generation lies in interpreting unstructured textual requirements, which are often vague, incomplete, or inconsistent. Although Harel et al.[17] introduced the concept of statecharts as an enhancement to finite state machines, their work did not focus on automating statechart generation from textual requirements, but instead focused on providing a better way to represent system behavior.

While these approaches show potential, they remain limited by the complexity of capturing system dynamics, event-driven transitions, and hierarchical states, which are integral to statechart modeling.

Despite the advancements in UML diagram automation, statechart automation has received relatively less attention. This is partly due to the higher complexity of statecharts compared to structural models like UML class diagrams. Statecharts require capturing not only the structure of the system but also the behavior, including the state transitions triggered by events, which are harder to model automatically from text. Researchers, such as Meng & Ban [26] have focused primarily on structural diagrams, such as class diagrams, demonstrating successful extraction of entities and relationships from textual requirements. However, their work does not address the modeling of behavioral logic or dynamic transitions, which are essential to statechart creation. Additionally, tools and methods designed for structural models do not easily translate to statecharts due to the added complexity of modeling dynamic behaviors and hierarchical states.

3.3 Integration of NLP and AI in Statechart Generation

Some research has also explored the automatic generation of UML diagrams and statecharts from textual descriptions. For instance, Zhong et al.[40] applied NLP techniques to produce SysML diagrams, a standard system engineering tool, showing the potential of AI in model construction. However, most of these methods focus primarily on structural diagrams, such as UML class diagrams, and do not address statecharts. This gap underscores the need for more focused research on automating statechart generation, specifically designed to handle the complexity of system behavior and transitions.

AI techniques have been increasingly used to generate executable models and code from statecharts. This includes the development of frameworks capable of transforming UML diagrams into executable code, representing an important area of research [11]. These frameworks focus on the implementation stage, where an existing model is translated into code. However, such frameworks typically assume the presence of pre-defined statechart models and do not address the challenge of generating them from natural language requirements. As a result, system-level statechart automation, particularly from unstructured text, remains underexplored.

Natural Language Processing (NLP) has shown potential in extracting structured information from unstructured text. NLP models that handle text for UML class diagrams are typically concerned with identifying entities, relationships, and structural elements, whereas statechart generation requires understanding the temporal and event-driven behaviors of systems. For example, Sandeep et al.[36] used a prob-

abilistic NLP approach to identify actors and use cases in textual requirements for the generation of use case diagrams. Although their approach demonstrates NLP's potential in automating diagram generation, it does not address the challenges specific to statechart automation, such as modeling system transitions and states.

3.4 Large Language Models and Their Application

Recent advances in Large Language Models (LLMs), such as GPT-4 and BERT, have shown increasing potential in supporting model-driven software development. These models are used to extract structured information from natural language, which is essential for automating statechart generation. For instance, transformer-based LLMs have been used to identify entities, events, and conditions from textual requirements, enabling the creation of structured models such as UML class and sequence diagrams. With additional refinement, similar approaches can be adapted to capture the behavioral elements required for statecharts. Liu et al.[39] created ChartifyText, an AI that converts text-based descriptions of data into structured visual charts, illustrating the broader potential of LLMs for generating structured models .

In this study, LLMs are explored for their ability to extract relevant information, such as states and transitions from textual requirements, to support the automation of statechart creation. This aligns with the overall objective of reducing manual effort in the early stages of system modeling.

Empirical experimentation is crucial in evaluating AI-driven techniques for automated model generation. Basili et al.[4] emphasize the need for systematic experimentation to validate software engineering methods, particularly AI-driven solutions. Prior studies have used experimental methodologies to compare AI-generated models with human-created ones, assessing their accuracy, completeness, and usability [32] This approach is critical for validating AI-driven statechart generation methods, which this study seeks to explore by automating statechart generation using AI.

3.5 Challenges in AI-Based Model Generation Techniques

While AI techniques have shown significant potential in automating software modeling tasks, including diagram generation and requirement analysis, several challenges remain unresolved across modeling approaches. Methods designed for structural model generation, such as class diagrams or use case diagrams, often rely on NLP techniques to extract states and transitions from textual input [26, 36]. However,

these approaches often exhibit inconsistent sentence structures, ambiguity in natural language, and variations in domain-specific terminology, which limit their scalability.

Behavioral models such as statecharts are often more complex than structural diagrams due to their dynamic nature. Statechart generation requires capturing dynamic system behavior, event-triggered transitions, and conditional logic, factors that are not typically addressed in structural modeling approaches [17, 36] . For instance, although Sandeep et al. [36] demonstrated the use of probabilistic NLP techniques for use case extraction, their method does not extend to event-driven behavioral modeling. Similarly, approaches like those of Meng and Ban [26], while effective for class diagrams, do not address behavioral interpretation.

Also, most LLM-based model generation techniques are trained on limited and domain-specific datasets, which restricts their generalizability to complex systems. In automotive domain, where accuracy is crucial, these limitations become even more significant. These challenges highlight the need for more targeted research on improving AI models for statechart generation, particularly methods that can connect unstructured requirements with accurate, executable behavioral models.

4

Methods

This study proposes an LLM-based solution for automating the generation of statecharts from unstructured natural language requirements in the automotive domain. The proposed solution involves multiple stage, including data collection and preprocessing of product function requirements, synthetic dataset generation to overcome limited data availability, fine-tuning of large language models (LLMs) such as GPT-3.5, GPT-4, and GPT-4o [30], and generation of structured statecharts in Mermaid.js format based on input requirements. The implementation utilizes tools like Azure OpenAI and Weights & Biases (W&B) to support model training and performance monitoring [5].

The goal of this research is to assess the effectiveness of this solution in a real-world context. Accordingly, the investigation is guided by the following research questions:

- **RQ1:** Does fine-tuning LLMs improve statechart generation?
- **RQ2:** How do LLM-generated statecharts compare to manually created ones?

To address these research questions, the study employs a quantitative experimental approach [8, 38] . The first research question (RQ1) explores whether fine-tuning improves performance [39]. The second research question (RQ2) compares LLM-generated statecharts with manually created ones, evaluating their alignment with the original requirements and their functional correctness and understandability. This comparison is also assessed quantitatively, focusing on performance metrics.

In addition to the quantitative analysis, expert reviews are conducted to evaluate the real-world usability and interpretability of both LLM-generated and manually created statecharts. Using this approach, the study provides a comprehensive assessment of AI techniques for statechart generation, considering both technical performance and practical applicability.

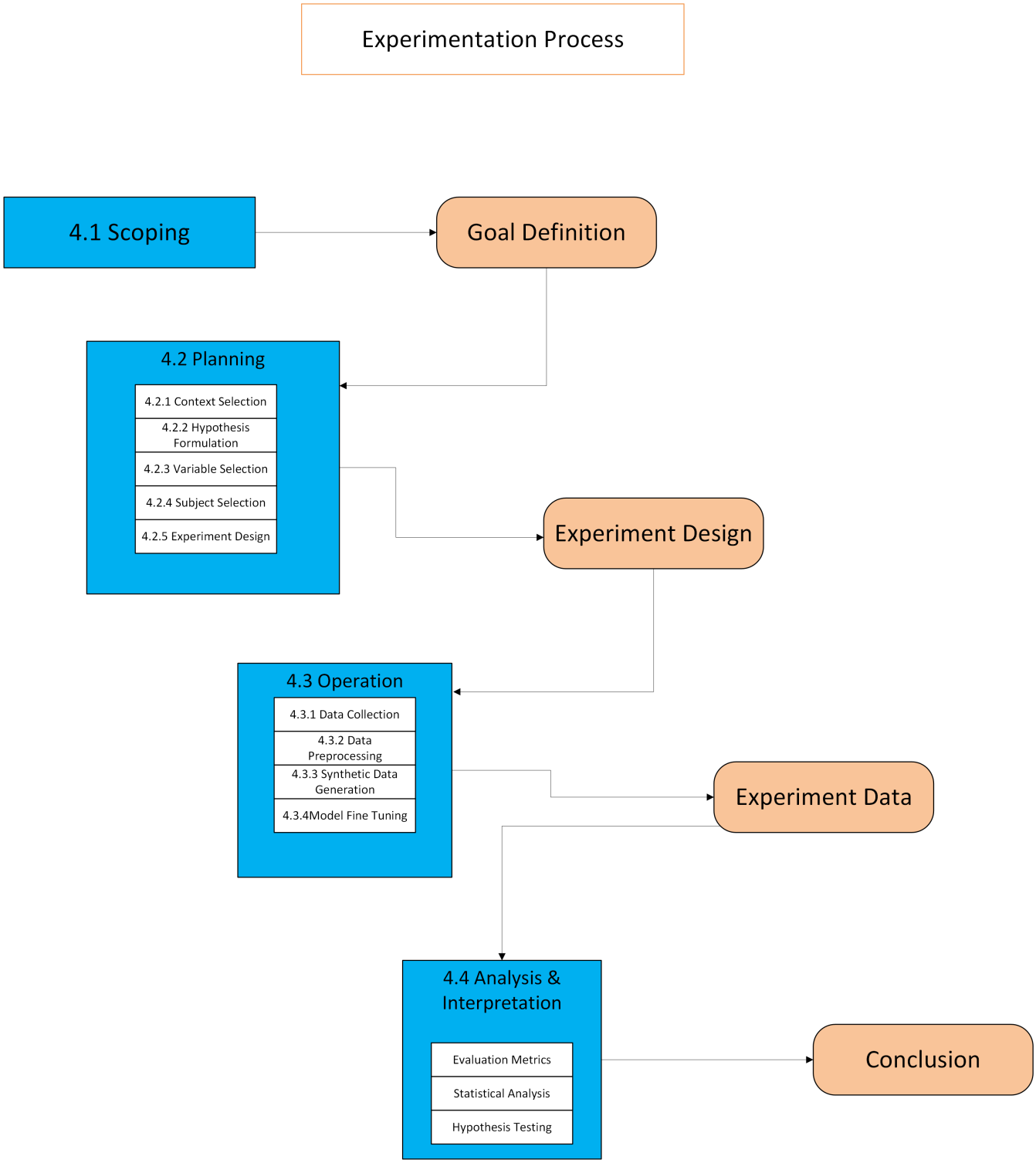

Figure 4.1: Overview of the Experiment Process

4.1 Scoping

The experiment aims to evaluate the effectiveness of fine-tuned LLMs in generating accurate and usable statecharts from unstructured natural language requirements in the automotive domain. To ensure that the intention of the experiment is met, it is essential to define its scope systematically, aligning with established methodologies in experimental software engineering. A structured framework helps establish clear objectives, constraints, and evaluation criteria.

Following the Goal/Question/Metric (GQM) framework by [4] the goal of this experiment can be defined as:

- Analysing AI techniques for generating statecharts from natural language requirements.
- The purpose of evaluating their feasibility and performance.
- In terms of functional correctness and understandability.
- The context of automotive software requirements engineering.

For this research, textual requirements were used to create a customised dataset for experimentation. These requirements are generally provided by product function owners through Car Weaver Inbox [35], a specialized requirements engineering tool that Volvo Cars has adopted to ensure that ART teams use updated requirements. The dataset includes detailed specifications of system states, actions, transitions, and events associated with the textual product function requirements, which serve as the basis for the experimental model. The primary focus of this experiment is to assess the effectiveness of AI techniques in generating accurate statecharts with respective to the product function requirements. The experiment is structured as a multi-object variation study, where different textual requirements are processed to evaluate AI performance with Text and JSON input types. The objectives of this experiment are as follows:

- Develop a structured dataset of textual requirements and corresponding statecharts, ensuring alignment with real-world automotive software practices.
- Fine-tune and adapt a Large Language Model (LLM) to generate statecharts from textual requirements.
- Validate LLM-generated statecharts against manually created ones to assess Functional correctness, understandability, and practical applicability.

4.2 Planning

Following the goal definition established in the scoping phase, the planning of this experiment addresses the steps outlined in experimental methodology: context selec-

tion, hypothesis formulation, variable selection, subject selection, experiment design, and validity evaluation.

4.2.1 Context Selection

The experiment is performed within the Volvo Cars environment, where statecharts are a key component in the validation of automotive systems. The goal was to analyze how large language models (LLMs) can automate statechart generation from textual requirements, an approach relevant to model-based systems engineering (MBSE)[9].

4.2.2 Hypothesis Formulation

Research Question 1 (RQ1): Does fine-tuning LLMs improve statechart generation?

Does fine-tuning LLMs on a domain-specific dataset, created from unstructured textual data, affect their ability to generate statecharts in terms of functional correctness, understandability.

Hypotheses:

- H_0: Fine-tuning does not significantly improve performance metrics (Functional correctness, understandability).
- H_1: Fine-tuning significantly improves performance metrics.

Research Question 2 (RQ2): How do LLM-generated statecharts compare to manually created ones?

Does statecharts generated by LLMs compare to manually created statecharts in terms of Functional correctness , understandability, and alignment with the original requirements.

Hypotheses:

- H_0: LLM-generated statecharts do not achieve comparable levels of functional correctness to manually created statecharts.
- H_1: LLM-generated statecharts achieve comparable or higher levels of functional correctness.

4.2.3 Variable Selection

Independent Variables: The independent variables are those variables that can be dynamically changed in the experiment [38], like the method used to generate the statechart and fine-tuning of LLMs.

- Manually created statecharts (by domain experts).
- Pre-trained LLM (publicly available model like GPT).

- Fine-tuned LLM (on domain-specific dataset for statechart generation).

Dependent Variables: The effect of treatments is assessed through carefully selected dependent variables, each derived directly from the hypothesis [38] and measured using well-defined performance metrics. The primary dependent variable in this study is state chart quality, which is evaluated through several key aspects:

- **Functional correctness:** Comparison of states and transitions between LLM-generated and manually created statecharts.
- **Understandability:** Evaluated by domain specialists according to clarity and interpretability.
- **Efficiency:** Time taken to generate the statechart (LLM generated vs. manually created).

4.2.4 Subject Selection

Domain experts from Volvo Cars with experience in statechart modeling were chosen as evaluators for the experiment since we need to evaluate LLM-generated statecharts. As they are experts in requirement engineering, they will help us evaluate the LLM-generated outputs against the manually created statechart.

4.2.5 Experiment Design

The experiment follows a structured evaluation approach to ensure clarity and consistency in assessing LLM-generated statecharts. The process consists of the following key steps:

- **Dataset Pre-processing:** Converting textual requirements into a structured format for input.
- **Fine-tuning:** Utilizing Azure OpenAI to fine-tune LLMs for statechart generation.
- **Statechart Generation & Comparison:** Generating statecharts using fine-tuned LLM and comparing them against manually created ones.
- **Evaluation:** Measuring Functional correctness , Understandability and efficiency .

4.2.6 Data Analysis

This study employs quantitative data analysis to ensure a detailed evaluation of LLM-generated statecharts.

Quantitative Data Analysis

Quantitative analysis focuses on evaluating performance metrics for LLM-generated statecharts, specifically in terms of functional correctness and understandability. This analysis directly supports both RQ1 and RQ2, with hypothesis testing used to determine whether fine-tuning significantly improves these metrics. In addition

to quantitative analysis, expert reviews and feedback are collected to assess the human evaluation of the statecharts' understandability and practical applicability. This approach addresses RQ2, using expert insights to identify patterns and evaluate whether LLM-generated statecharts align with or outperform manually created ones in terms of functional correctness, understandability, and alignment with original requirements.

4.3 Operation

The operation phase outlines the practical steps involved in executing the experiment. This includes data collection, preprocessing, model training, statechart generation, and evaluation. Each step is designed to ensure that LLM-generated statecharts align with manually created ones, maintaining accuracy and reliability. The structured approach facilitates systematic assessment and performance measurement, contributing to the overall validation of the methodology.

4.3.1 Data Collection

For this research, textual requirements were used to create a customised dataset for experimentation. These requirements are generally provided by product function owners through Car Weaver tool [37] , a specialized requirements engineering tool that Volvo Cars has adopted to ensure that Agile Release Train (ART) teams use updated requirements. The dataset includes detailed specifications of system states, actions, transitions, and events associated with the textual product function requirements, which serve as the basis for the experimental model.

The datasets employed in this experiment are as follows:

- **Primary Dataset**: This dataset is manually prepared by extracting the textual requirements for 20 product functions from the Car Weaver tool. Each requirement outlines a specific feature or functionality of the system.
- **Secondary Dataset**: Synthetic data is used for fine-tuning the model and evaluating the Large Language Model's (LLM) ability to understand the context of a given prompt and generate a relevant output.

These 20 product function requirements were chosen due to their clarity, behavioral depth, and inclusion of well-defined events, actions, and transitions. However, the limited size of the primary dataset is insufficient for effectively fine-tuning a large language model. To overcome this limitation, we adopt a synthetic data generation strategy [24]. This approach involves generating additional data that retains the structure of the original requirements while randomizing the system states, events, transitions, and actions. This method enhances the dataset size, allowing for more accurate model performance without the need for extensive manual data collection.

The following definitions are used throughout this study:

- **Product Functions:** High-level descriptions of vehicle features derived from

the Car Weaver Tool for preparing dataset.

- **Textual Requirements:** Refined natural language statements derived from each product function, specifying expected system behavior, events, and transitions.
- **Test Cases:** New product functions with corresponding textual requirements were used specifically for evaluation purposes. A total of 12 test cases were created by domain experts and excluded from both the training and synthetic data generation phases.

4.3.2 Data Pre-processing

In the context of automated statechart generation from natural language specifications, the initial phase involves data pre-processing of our primary dataset. In general, the data pre-processing stage is important for transforming raw, unstructured textual product functions into a format applicable to machine learning applications [12] . The primary objective of pre-processing is to convert this raw text into a structured representation, that is prompt-completion pairs in JSON (JavaScript Object Notation) format, which is the most common input data type in machine learning implementation due to its standardization and machine readability [18] [10].

The proposed data pre-processing phase involves Feature Extraction, Data Transformation,and Data Labelling procedures.

Step 1: Feature Extraction and Tokenization using spaCy

The process involves extracting relevant components from the text (states, events, transitions, etc.) using Natural Language Processing (NLP) techniques, with Python's spaCy library, a library for advanced NLP tasks. The methodologies implemented are Named Entity Recognition (NER) and Part-of-Speech (POS) tagging [19, 20] .spaCy serves as the NLP tool for automated feature extraction. The application of NER and POS tagging within spaCy can be used for the identification of relevant components within the statechart specifications:

- **Named Entity Recognition (NER):** NER is utilized to identify and categorize key entities within the text that correspond to statechart elements. For instance, entities representing states, events, and potential actions are identified and labeled. In our implementation, we used the pre-trained spaCy model, (`en_core_web_sm`), which provides English language processing capabilities for the specific vocabulary domain of statechart specifications [19] .

```
import spacy
from spacy.tokens import Span

# Load pre-trained model
nlp = spacy.load("en_core_web_sm")

# Define statechart-specific patterns
patterns = {
    "STATE": ["locator", "client", "app", "drive"],
    "ACTION": ["presented", "request", "activation", "
       available"],
    "EVENT": ["user", "horn", "light"]
}
# Add custom component to pipeline
@spacy.language.Language.component("statechart_ner")
def statechart_ner(doc):
    new_ents = []
    for token in doc:
        for label, terms in patterns.items():
            if any(term in token.text.lower() for term in
                terms):
                new_ents.append(Span(doc, token.i, token.
                   i + 1, label=label))
    doc.ents = list(doc.ents) + new_ents
    return doc
nlp.add_pipe("statechart_ner", after="ner")
```

Figure 4.2: Example Implementation of NER for Car Locator Product Function

- **Part-of-Speech (POS) Tagging:** POS tagging is utilized to analyze the grammatical role of each word within the text. Understanding the part of speech helps in determining the relationships between entities. For example in our case, states/events comes under nouns, transitions comes under verbs, and relationships within state transitions comes under prepositions.

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag

# Ensure necessary NLTK components are downloaded
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Car Locator requirements as text
car_locator_text = """
The car location is presented to a remote client.
One such client might be the Volvo Cars Mobile App,
where both the vehicle location is presented relative to
   user position.
User can request activation of horn and light from VOC
   mobile app.
Car locator is not available during drive.
"""

# Tokenization and POS tagging
tokens = word_tokenize(car_locator_text)  # Tokenize
   words
tagged_words = pos_tag(tokens)  # POS tagging
```

Figure 4.3: Example implementation of POS tagging for Car Locator product function

These extracted components, states, events, transitions, and actions represent the fundamental building blocks of a statechart model. These components provide the essential information required for a machine learning model to understand the structural and behavioural aspects described within the textual specifications.

Step 2: Data Transformation

Data transformation is the process of converting data from one format or structure to another. In this research, the major transformation is from unstructured text to a predefined, structured JSON object that explicitly represents a statechart.

Structuring Extracted Features into JSON

The extracted components (states, events, transitions, and actions) obtained from feature extraction step are structured into JSON format. This transformation ensures that the information is presented in a manner that is both human and machine-

readable. The JSON structure is designed to inherit the structure of a statechart, which includes keys for:

- **States:** An array listing all identified states.
- **Events**: An array of events recognized in the specification.
- **Transitions:** An array that defines transitions, including attributes such as `from_state`, `to_state`, and `triggering_event`.
- **Actions:** An array specifying the actions associated with state transitions, which may include execution conditions, effects, or system responses.

Step 3: Data Labeling

The final data pre-processing stage involves data labeling, which is crucial for enabling supervised learning [21]. The labelled data is then organized into prompt-completion pairs to enhance model training and fine-tuning accuracy. Prompt-completion pairs serve as the standard input format for training and fine-tuning Large Language Models (LLMs). These pairs follow a simple structure of $x \rightarrow y$, where x is the input (prompt) and y is the expected output (completion). In general, prompt-completion pairs are question-answer sets provided to the LLM during the training.

Formulation of Prompt-Completion Pairs

The labeling process is implemented by creating pairs of inputs and desired outputs:

- **Prompt (Input):** The refined textual specification of each product function requirements corresponding to a statechart serves as the input to the machine learning model.
- **Completion (Output/Label):** The structured JSON representation of the product function requirements corresponding to a statechart, generated through the preceding pre-processing, feature extraction, and transformation steps. This JSON object is the target output that the model is trained to predict.

By creating these prompt-completion pairs, this research frames the statechart generation problem as a supervised learning task. In each pair, the "completion" serves as the expected output or label for the corresponding "prompt." This labeled dataset is then used to further finetune a base Large Language Model (LLM) from, helping it learn how to convert textual statechart specifications (prompts) into structured JSON representations (completions). With this approach, the model can generalize from the prepared prompt-completion pairs and expect to generate JSON outputs for new, unseen textual specifications.

4.3.3 Synthetic Dataset Generation

This section details the methodology for the generation of a synthetic dataset of prompt-completion pairs. The synthetic dataset was generated by extracting unique elements, states, events, transitions, and actions from the primary dataset and recombining them using controlled randomization. Our approach is designed to im-

plement variations while maintaining the original structural format of the primary dataset. To facilitate this process, we utilized the 'random' library in Python for data randomization and the 'JSON' library for storing and generating data in JSON format [24].This preparation ensured that our data generation process would yield valid and reliable results for later analysis.

Generation of Component Pools

The first phase of our synthetic data generation process involved a detailed analysis of the primary dataset to identify and extract all unique, fundamental components within the structured completions. This step aimed to establish valid elements that could be systematically recombined during the synthesis process. We compiled all component types present across various state machine definitions in the original dataset, including states, events, transitions, and actions. These elements serve as the foundational building blocks for generating new synthetic data points. This analytical approach aligns with the experimental preparation [38] , where appropriate selection of materials is essential for experimental success.

Data Randomization

To achieve this shuffling, we employed Python's 'random' library. This library provides functionalities to randomize given data based on specified rules. We used random shuffling process as our main method of transformation. The method was intended to increase sentence variation while maintaining the original input's meaning. The method involved maintaining all components of an original completion while changing their positions within each set group. The categories were states, events, actions, transitions. The reason for this is that the order of elements in such structures is typically random, and this approach ensured structural consistency while generating diverse representations. By changing the order, we were able to generate a variety of examples that helped to strengthen the model.

This controlled randomization mirrors the execution methodology [38], where experimental conditions must be managed carefully to avoid unwanted influences on the results.

```
# Generate synthetic dataset by modifying and combining
   original pairs
def generate_synthetic_data(dataset, num_samples=500):
    if not dataset:
        print("No valid data available for synthetic
           generation.")
        return []

    action_pool = extract_unique_actions(dataset)  # Get
       all possible actions

    synthetic_data = []

    for _ in range(num_samples):
        base_pair = random.choice(dataset)  # Select a
           random pair
        mutated_completion = perturb_json(
           shuffle_json_structure(base_pair["completion
           "]), action_pool)

        new_entry = {
            "prompt": base_pair["prompt"],
            "completion": mutated_completion
        }

        synthetic_data.append(new_entry)

    return synthetic_data
```

Figure 4.4: Example Implementation of Data Randomisation

Iterative procedure for synthetic data generation

The synthetic data generation process followed an iterative procedure involving controlled variation. This approach ensured the creation of diverse yet structurally valid synthetic examples. The process consisted of the following steps, repeated until the desired dataset size was achieved:

1. **Selection of original pair:** In each iteration, an existing prompt-completion pair was randomly selected from the original dataset. The prompt served as the basis for generating a new synthetic example.

2. **Structure shuffling:** The structure shuffling technique, as described in data randomization step, was applied to the completion part of the selected pair. This step introduced structural variation while preserving the integrity of the state machine definition.

3. **Controlled variation:** The shuffled completion was then selectively modified by adding or replacing states, events, actions, and transitions while ensuring logical consistency. To control this process, simple rules were followed, ensuring that all states could be reached and transitions made sense based on the events. Each modified version was manually reviewed to ensure that no incorrect or meaningless statechart structures were created.

4. **Retention of original prompt:** The original prompt from the selected pair was retained and paired with the newly modified completion, which enables structured shuffling and controlled adjustments.

This iterative procedure corresponds to both the execution and data validation in experimental execution. We applied shuffling and variation to our data pairs, while maintaining limited direct intervention to avoid biasing results. Additionally, our controlled approach to variation reflects the data validation concerns in [38], ensuring our synthetic data remained consistent and accurate. The approach is particularly valuable for augmenting limited datasets of structured outputs such as state machines.

To ensure data quality, the generated synthetic completions were manually reviewed and validated by domain experts. This manual validation helped confirm that the recombined scenarios were meaningful, realistic, and suitable for model fine-tuning. The synthetic dataset was used exclusively for training purposes and was not included in the evaluation.

4.3.4 Model Fine Tuning:

Model Selection:

Selection of an appropriate LLM plays a critical role in fine-tuning systems for specialized tasks such as converting natural language requirements into statecharts. After thorough consideration of the available options in the Azure AI foundry platform , this research uses three variants from OpenAI's suite of large language models: GPT-3.5 Turbo, GPT-4, and GPT-4o [30].

The strengths of these models lie in their ability to handle complex language tasks, process structured and unstructured text, and generate accurate outputs. These models were considered suitable for tasks requiring advanced reasoning, precision, and contextual understanding, all of which are critical for generating accurate state charts. The evaluation of these models and their comparative performance will be discussed in the results section.

Process of Fine-Tuning

The fine-tuning process represents a crucial phase in adapting pre-trained models to domain-specific tasks, ensuring performance and accuracy.

Setup of Model and Optimization Parameters

For the fine-tuning of GPT-3.5, GPT-4, and GPT-4o models, multiple hyperparameters were configured to optimize performance outcomes. For each model, various combinations of hyperparameters were tested to determine the most suitable configuration for our specific task domain.

Table 4.1: Training parameters for different GPT model runs

Model Run	Batch Size	Learning Rate	No. of Epochs	Seed
GPT-4 1st Run	32	1.01	8	42
GPT-4 2nd Run	16	1.01	8	42
GPT-4 3rd Run	16	1.05	5	42
GPT-4o 1st Run	32	1.01	8	42
GPT-4o 2nd Run	16	1.01	8	42
GPT-4o 3rd Run	16	1.05	5	42
GPT-3.5 1st Run	32	1.01	8	42
GPT-3.5 2nd Run	16	1.01	8	42
GPT-3.5 3rd Run	16	1.05	5	42

- **Learning Rates:** Values of 1.01 and 1.05 were evaluated to examine their impact on the convergence speed and stability of the model training process.
- **Epochs:** Training iterations of 3 to 10 passes through the dataset were conducted to ensure acceptable exposure to data while minimizing overfitting risks.
- **Batch Size:** Sizes of 16 and 32 were number of samples per training setup.
- **seeds:** Common seed was used to ensure consistency in training outcomes.

Dataset Description and Usage

For effective fine-tuning of the models a systematic data partitioning strategy is implemented to ensure robust model training and evaluation. The dataset was divided into training and validation sets in an 80:20 ratio [23] using Random and JSON libraries.

The training set, comprising 80% of the total data volume, was utilized for the primary model training process. This portion provides sufficient examples for the

models to learn relevant patterns and features necessary for converting natural language requirements into accurate statechart representations.

The validation set, constituting the remaining 20% of the data, was used to periodically assess model performance during the training process. This evaluation component is essential for monitoring learning progress, detecting potential overfitting, and guiding adjustments to hyperparameters as necessary to optimize model performance.

Fine-Tuning Implementation

Fine-tuning was implemented using the Azure AI Foundry platform, which provided the computational infrastructure for scalable model training. To monitor and manage experiments, we integrated Weights & Biases (W&B) [5], a popular tool for tracking machine learning workflows. For each training configuration, we recorded key elements including the hyperparameters used (learning rate, epoch count, batch size,seed), loss metrics (training and validation loss curves), and task-specific performance scores such as accuracy. This systematic tracking enabled transparent evaluation and comparison across multiple training runs.

4.4 Model Evaluation

4.4.1 Quantitative Model Evaluation

This section presents an evaluation method of fine-tuned language models (LLMs) on the task of automated statechart generation from natural language requirements. After fine-tuning GPT-3.5 turbo, GPT-4, and GPT-4o for this domain-specific task, it is crucial to assess their performance to ensure that the models are well-suited to generating accurate statecharts. Model evaluation plays a key role in AI/ML tasks, as it helps verify that the models deliver reliable results. To achieve this, we compare different runs of the models using quantitative metrics tracked throughout the training process.

Evaluation Setup

Three base models GPT-3.5, GPT-4, and GPT-4o were fine-tuned using the Azure AI Foundry platform. The training datasets consisted of paired inputs (natural language requirements) and target outputs (statechart representations in structured JSON). The following tools and techniques were used:

Azure AI Foundry was used for model fine-tuning and deployment.

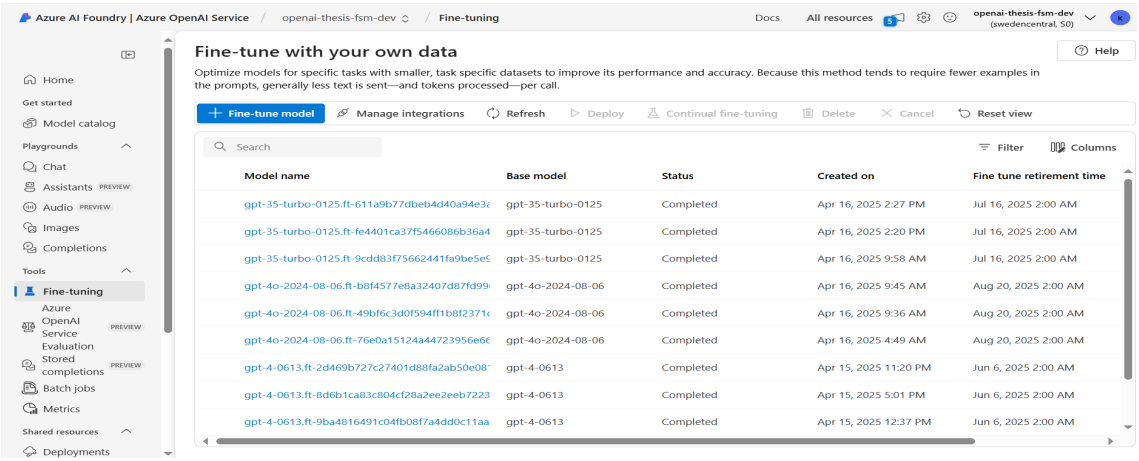

Figure 4.5: Azure AI Foundry Platform

Weights & Biases (W&B) is a robust AI development platform integrated with Azure AI foundry for tracking metrics and visualizing results of training/validation metrics and experiment comparisons.

Each model was fine-tuned across multiple runs, and the following core metrics were logged after the fine-tuning process:

Table 4.2: Description of Evaluation Metrics Used in the Model

Metrics	Description
train_loss	Represents the training loss, which shows how well the model is minimizing errors in predicting Token Accuracy.
train_mean_token_accuracy	The token-level accuracy on the training set, indicating how well the model is learning from the data.
valid_loss	The validation loss during the validation phase. Lower values indicate better performance.
valid_mean_token_accuracy	The accuracy of token generation during validation. Higher values indicate better performance.
full_valid_loss	The loss on the entire validation set, typically cross-entropy loss. Lower values indicate higher accuracy and generalization.
full_valid_mean_token_accuracy	The token-level accuracy on the validation set, averaged across all tokens in a sequence.

Model Selection:

To determine the most suitable language model for our tasks, we conducted a comparative evaluation of three fine-tuned models: GPT-3.5-turbo, GPT-4, and GPT-4o. Each model was fine-tuned using identical training/validation data, hyperparameters, and experimental settings to ensure a fair comparison. The evaluation was performed across three independent training runs per model to assess consistency and robustness.

For model selection, we employed Analysis of Variance (ANOVA) to compare the performance of the models. ANOVA was chosen because it allows for the comparison of means across multiple groups and helps determine if there are statistically significant differences in model performance. Since ANOVA requires normally distributed data, the data set was tested using the Shapiro-Wilk test to confirm normality before proceeding.

Tukey's Honestly Significant Difference (HSD)

Post-hoc test is usually conducted after an ANOVA test to determine which specific groups are significantly different from each other, while controlling the Type I errors. Tukey's HSD test can be performed using python's "pairwise_tukeyhsd()" of "statsmodels.stats.multicomp" library [15] .

The formula for Tukey's Honestly Significant Difference (HSD) is:

$$\text{HSD} = q_{\alpha,k,\text{df}_{\text{within}}} \times \sqrt{\frac{\text{MS}_{\text{within}}}{n}}$$

Where:

- $q_{\alpha,k,\text{df}_{\text{within}}}$ is the critical value from the studentized range distribution, based on the significance level α, number of groups k, and degrees of freedom within groups.
- $\text{MS}_{\text{within}}$ is the mean square within groups (from ANOVA).
- n is the number of observations per group (assuming equal group sizes).

If the difference between the means of two groups exceeds the HSD value, the difference is considered statistically significant:

$$|\bar{x}_i - \bar{x}_j| > \text{HSD} \Rightarrow \textbf{Statistically Significant Difference}$$

Additionally, to compare expert reviews of the models, we utilized the Wilcoxon Signed-Rank Test [34], which is particularly useful when dealing with paired samples and ordinal data, such as those obtained from Likert-scale ratings. This non-

parametric test was used to compare expert reviews between the fine-tuned model and the base model, as well as between the fine-tuned model and human-generated statecharts. The Wilcoxon Signed-Rank Test was selected due to its ability to handle non-normally distributed data and the ordinal nature of the Likert-scale responses.

The statistical analyses, including ANOVA and Wilcoxon Signed-Rank Tests, were performed using Python and the SciPy library. For data visualization and further analysis, we used Matplotlib to plot results and evaluate trends in model performance and expert reviews.

4.4.2 Expert Reviews

Following the selection of the best-performing model, based on the quantitative model evaluation, an expert review and feedback process was conducted to assess the applicability and usability of the fine-tuned model in the practical application. A total of **twelve test cases** were prepared in collaboration with domain experts from the Product Simulation Team at Volvo Cars. Each test case consisted of a natural language requirement and a corresponding expert-created statechart, which served as the reference output. The test cases represent typical automotive functions, such as direction indication, mirror adjustment, hood operation, and so on, as shown in Table 5.3.

The selected fine-tuned model was deployed using Azure AI Foundry. The textual requirements were provided as input to the model, and the output was requested in Mermaid.js syntax. The resulting code was then used to generate visual statecharts via the Mermaid.js editor.

After generating the twelve model-produced statecharts, an expert review was carried out through semi-structured interviews with four domain experts. All participants were experienced software developers with a background in system modeling, each with more than five years of experience in automotive system design and model-based development. Their responsibilities include creating and validating system requirements and functional statecharts for simulation and testing. Each expert was familiar with the end-to-end statechart creation manually, ensuring their ability to critically assess the LLM-generated statecharts. Their domain knowledge and modeling experience make them well-qualified to evaluate the technical correctness and usability of LLM-generated statecharts. The interviews are aimed at evaluating the functional correctness and understandability of the generated statecharts compared to the expert-created references.

Each interview lasted between 30 and 45 minutes, during which experts reviewed the generated statecharts based on the following key aspects guided by the interview questions listed in Appendix 2:

- Clarity of state transitions, especially for systems with multiple interaction modes.
- Logical representation of control flows in safety-relevant features.

- Visual consistency in handling repetitive patterns.
- Alignment of event-based behaviors with the expectations of the automotive user interface.
- Understandability of system behavior in response to simultaneous or conflicting inputs.
- Identification of ambiguities or modeling gaps, especially in time-bound or condition-based logic.
- Suggestions for improving the accuracy, readability, or maintainability of the model.

5

Results

This section presents the outcomes of the experimental evaluations conducted to assess the effectiveness of the fine-tuned LLMs in generating statecharts from natural language requirements. Quantitative results are explored, with an importance on how fine-tuning has improved model performance in generating statecharts. The results show the alignment between LLM-generated and manually created statecharts, as well as the functional correctness and understandability of the outputs. Through a careful analysis of the model's performance in a real-world automotive software engineering context, the findings highlight key strengths and limitations of the fine-tuned LLMs, ultimately offering experimental evidence of their potential for automated model generation in this domain.

5.1 Model Selection:

This section describes the results of visualizations and statistical analyses conducted to compare model performance and support the selection of the most suitable model.

Visual Analysis

While multiple evaluation metrics were computed during training and validation as shown in **Table 4.2**, `full_valid_mean_token_accuracy` and `full_valid_loss` from the W&B tool were selected for visual analysis and model comparison because they provide the most detailed indicators of overall model performance on the validation set. These metrics reflect the final performance across the entire validation dataset, offering a clearer picture of how well the model generalizes.

Other metrics, such as `train_loss` or `train_mean_token_accuracy` are useful for monitoring learning during training but may not accurately represent generalization. Therefore, focusing on the two full validation metrics allows for a more objective and meaningful model selection process.

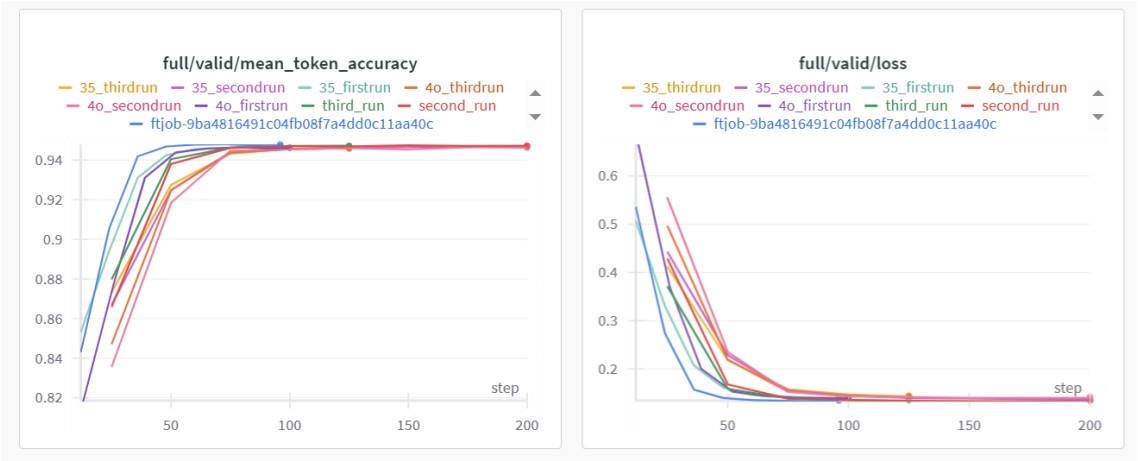

Figure 5.1: `full_valid_mean_token_accuracy` and `full_valid_loss` from the W&B tool.

The above figure demonstrates that GPT-4 achieved the highest average mean token accuracy (0.94728) and the lowest average validation loss (0.13422), outperforming GPT-4o and GPT-3.5-turbo, which showed similar but slightly lower performance. To validate these findings statistically, we will conduct parametric tests to confirm the significance of these results.

Statistical Analysis:

In Machine Learning approaches, parametric tests are conducted to statistically evaluate the performance of the models. To determine the model with the best performance, we conducted one-way ANOVA test and Post-hoc Tukey's Honestly Significant Difference (Tukey's HSD) test.

The input for this analysis was the valid mean token accuracy and valid loss across three experimental runs for each model, as presented in the table below:

Table 5.1: Valid_Mean_Token_Accuracy Across Models

Models	1st Run	2nd Run	3rd Run
GPT 4	0.94788	0.94728	0.94723.
GPT 4o	0.94621	0.94629	0.94593.
GPT-3.5-TURBO	0.94648	0.94698	0.94597.

Table 5.2: Valid_Loss Across Models

Models	1st Run	2nd Run	3rd Run
GPT 4	0.13451	0.13422	0.13532.
GPT 4o	0.14088	0.14180	0.14289.
GPT-3.5-TURBO	0.14071	0.13829	0.14465.

- **ANOVA test:** Analysis of Variance is a statistical test used to compare means of three or more groups (models in our case) to determine if there is any

statistically significant difference between them. The result for the ANOVA test is called "p-value". A p-value below 0.05 typically indicates that there is a significant difference between the groups. An ANOVA test can be performed with Python's scipy library using "f_oneway()" [31].

Before conducting an ANOVA test, it is important to check if the data is normally distributed. A Shapiro-Wilk test was performed to find this. The results of this test proved that the data is normally distributed for all three models ($P > 0.05$). Now that the data are normally distributed, an ANOVA test can help determine whether there is a significant difference between the models' accuracies.

For mean token accuracy:

A one-way ANOVA was conducted to determine whether there were statistically significant differences in accuracy among the three language models: GPT-3.5, GPT-4, and GPT-4o. The analysis revealed a significant effect of model type on accuracy, $F(2, 6) = 10.06$, $p = 0.0121$, indicating that at least one model differed significantly from the others.

Post hoc comparisons using the Tukey HSD test:

- GPT-4 showed significantly higher accuracy than GPT-3.5 (*mean difference* = 0.001, $p = 0.0414$, 95% CI [0.0000, 0.0019]).
- GPT-4 also outperformed GPT-4o (*mean difference* = 0.0013, $p = 0.0119$, 95% CI [0.0004, 0.0023]).
- The difference between GPT-3.5 and GPT-4o was not statistically significant (*mean difference* = −0.0003, $p = 0.5543$, 95% CI [−0.0013, 0.0006]).

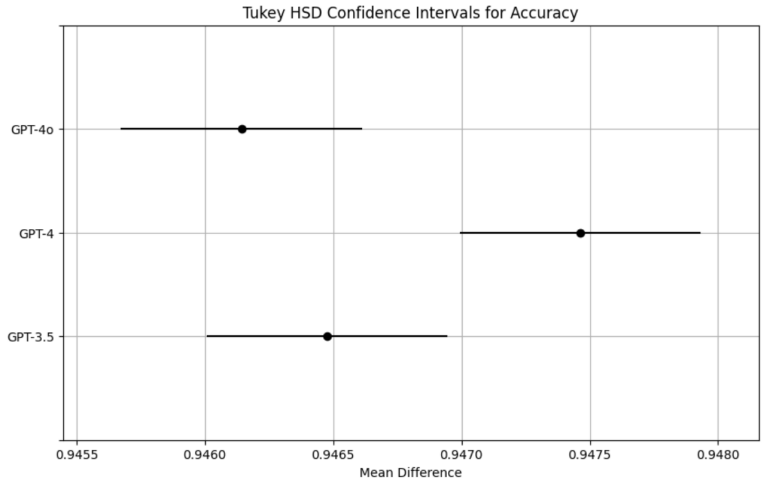

Figure 5.2: Tukey HSD Confidence Intervals for Accuracy

For validation loss:

A one-way ANOVA was conducted to assess whether there were statistically significant differences in loss values among the three fine-tuned GPT models (GPT-3.5, GPT-4, GPT-4o), based on three training runs per model. The results indicated a significant effect of model type on loss, $F(2, 6) = 12.18$, $p = 0.0077$, suggesting that the models differed in their optimization performance.

Post hoc comparisons using the Tukey HSD test:

- GPT-4 achieved significantly lower loss than GPT-3.5 (*mean difference* $= -0.0065$, $p = 0.0156$, 95% CI $[-0.0115, -0.0016]$).
- GPT-4 also had significantly lower loss than GPT-4o (*mean difference* $= 0.0072$, $p = 0.0102$, 95% CI $[0.0022, 0.0121]$).
- No significant difference in loss was observed between GPT-3.5 and GPT-4o (*mean difference* $= 0.0006$, $p = 0.9176$, 95% CI $[-0.0043, 0.0056]$).

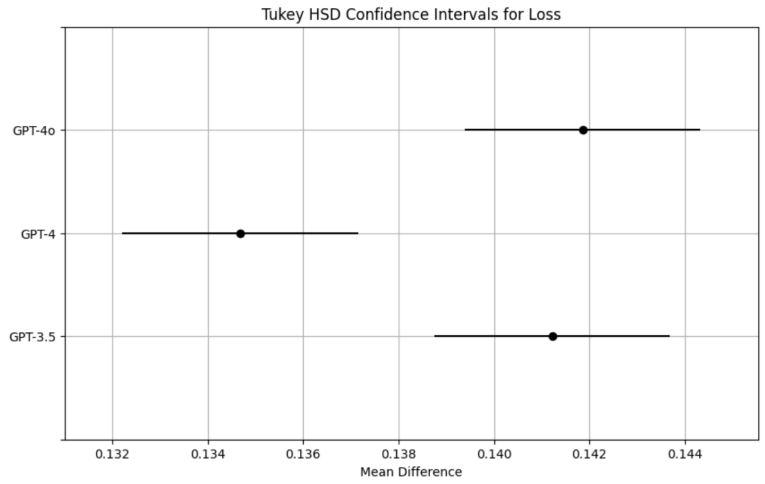

Figure 5.3: Tukey HSD Confidence Intervals for Loss

These results confirm that GPT-4 slightly outperformed both GPT-3.5-turbo and GPT-4o in terms of both accuracy and loss, while GPT-3.5-Turbo and GPT-4o did not differ significantly from each other.

Based on both the visual and statistical analyses, it is evident that GPT-4 is the highest-performing model.

Although GPT-4's fine-tuned version with the hyperparameters batch_size = 16, learning_rate = 1.01, epochs = 8, and seed = 42 has the second-best accuracy, it was selected for final deployment. The primary reason for this choice is that the model with the highest accuracy uses a batch size of 32, which would make deployment and computation more challenging and resource intensive. Additionally, the fact that the model with batch_size = 16 completed the full fine-tuning schedule until step 200 and achieving better convergence also proves that this model is more suitable for deployment and application.

5.2 Analysis of Fine-tuned vs Non-Fine-tuned LLM outputs

Overview of the Test Cases and Data Collection

This section summarizes the 12 test cases provided by experts at Volvo Cars. The test cases are designed to verify the functionality and performance of various features handled by the product simulation team, with an emphasis on user interactions and system responses. These test cases cover a broad range of product functions, and each test case defines a specific scenario for how the system should behave under different conditions.

Table 5.3: Test Cases Description

Test Case ID	Test Case Title	Short Description
TC 1	Direction Indication System	Four types of direction indication: continuously left, temporarily left, continuously right, and temporarily right, each with specific behaviors for activation and deactivation.
TC 2	Hazard Warning Light	Hazard warning light activation through both central display and overhead console, with synchronization between the two. All direction indicator lamps blink with a 500-millisecond interval when activated.
TC 3	Glove Box	Glove box operation using the central display. The glove box can be opened and closed, with locking mechanisms preventing access when locked.
TC 4	Charge Lid	Charge lid operation and warning system. The charge lid can be opened manually and will trigger a visual indicator and audible warning if the car is moving faster than 0 km/h.
TC 5	Windscreen Wiping	Windscreen wiper activation using the right stalk, with multiple levels for manual control (off, level 1, 2, or 3) and automatic control based on user-selected sensitivity levels (low, medium, or high).
TC 6	Windscreen Washing	Windscreen washing function activation by pulling the right stalk for more than 1 second. This function sprays washer fluid and continues wiping after the stalk is released, with a pre-defined number of wipes.
TC 7	Rear Window Wiping	Rear window wiper operation with manual and automatic settings. The wiper operates in 2 levels and automatically activates when the car is shifted into reverse.
TC 8	Rear Window Washing	Rear window washing function, similar to the windscreen washing process. Washer fluid is sprayed, and wiping continues for a fixed number of wipes after the stalk is released.
TC 9	Hood Operation	Hood operation, including the ability to open to a secondary position and fully open, as well as closing the hood. A safety mechanism is required to fully open the hood when it is in the secondary position.
TC 10	Mirror Folding Function	Mirror folding functionality, which automatically folds the mirrors when the car is locked from outside and unfolds when unlocked, if the setting is enabled. If locked or unlocked from inside, the mirrors do not fold/unfold.
TC 11	Mirror Adjustments	Mirror adjustments through the central display, with separate controls for adjusting the left and right mirrors.
TC 12	Steering Wheel Adjustment	Steering wheel adjustment functionality, which allows the user to adjust the steering wheel position via the central display with four directional options: up, down, left, or right.

StateChart Generation Process

Upon deploying both the fine-tuned and non-fine-tuned versions of the GPT-4 models, they are tested for functional correctness and understandability using the above 12 test cases given by Volvo Cars. Both the models were given the same prompt, that is, to generate a mermaid.js code for the given textual requirement [28]. The resulting mermaid.js code is then used to generate a statechart in mermaid.js editor. After generating the statecharts with both the fine-tuned and non-fine-tuned versions, the results were stored to be used in the expert ratings for accurate evaluation.

Results of Statechart Generation

To evaluate the performance of the fine-tuned and base language models in generating statecharts for the "windscreen washing" functionality, we compare the Mermaid.js syntax produced by both models. The following two syntax examples represent the output of the base and fine-tuned models for the same test case, TC6 as example, which simulates the user interaction of activating the windscreen washer.

- **Fine-tuned Model Syntax (GPT-4 Fine-tuned) for Test Case (TC6): Windscreen Washing**

```
stateDiagram-v2
    state "Idle" as idle
    state "WindscreenWiping" as wiping
    state "WindscreenWashing" as washing
    state "WasherFluidSpraying" as spraying
    state "WiperContinues" as continues

    idle --> washing : User pulls right stalk for > 1 second
    washing --> wiping : Function activated
    washing --> spraying : Function activated
    spraying --> washing : User stops pulling the stalk
    washing --> idle : User stops pulling the stalk
    wiping --> continues : User stops pulling the stalk
    continues --> idle : After 3 more wipes
    wiping --> spraying : Windscreen wiper active
    spraying --> wiping : Washing function deactivated
```

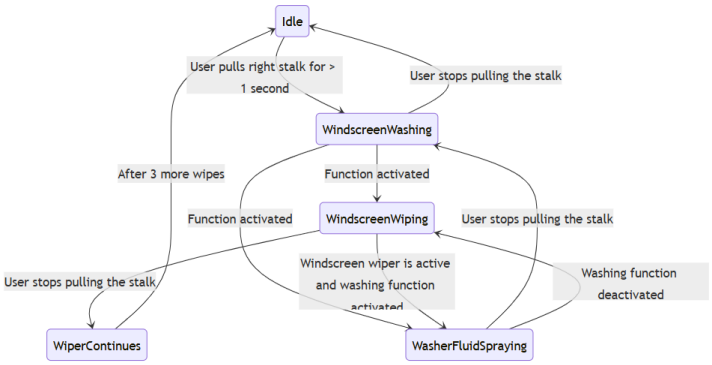

Figure 5.4: Finetune Model

- **Base Model Syntax:**

```
stateDiagram-v2
    [*] --> StalkNotPulled: Start
    StalkNotPulled --> WashingEnabled: StalkPulled > 1s
    WashingEnabled --> StalkNotPulled: StalkReleased
    WashingEnabled --> WiperActive: WiperActivate
    WiperActive --> WiperContinue: WashingDisabled
    WiperContinue --> [*]: 3WipesDone

    alias StalkNotPulled: Stalk Not Pulled
    alias WashingEnabled: Washing Enabled
    alias WiperActive: Wiper and Washer Active
    alias WiperContinue: Wiper Continues After Washing
    alias [*]: End
```

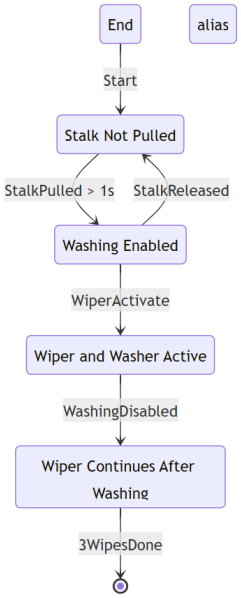

Figure 5.5: Base Model

The figures above compare the statecharts generated by the base model and the fine-tuned model for test case TC6 (*windscreen washing*). The statechart produced by the base model follows a linear structure with minimal branching, capturing only the primary functional steps. It uses general state labels such as *Washing Enabled* and *Wiper and Washer Active*, which provide limited insight into the system's behavioral flow. Transitions are abstract and do not reflect specific user actions or system conditions, reducing the model's interpretability.

In contrast, the fine-tuned model generates a more detailed and context-aware statechart. It incorporates intermediate states such as *WindscreenWashing*, *WindscreenWiping*, and *WasherFluidSpraying*, along with clearly defined transitions triggered by specific user inputs (e.g., "User pulls right stalk for > 1 second") or system responses (e.g., "After 3 more wipes"). The structure includes multiple branches, bidirectional transitions, and logical sequencing aligned with real-world use cases. This level of detail supports better traceability, enhances model clarity, and improves its suitability for validation and early-stage testing.

Expert Evaluation on Statechart Quality

To evaluate the quality of statecharts generated by both the base and fine-tuned LLM models, a group of four experts from the product simulation team, Volvo Cars assessed the outputs based on two main criteria functional correctness and understandability. Experts rated the statecharts on a Likert scale from 1 to 5, where 1 represented poor performance and 5 indicated excellent performance. The evaluations were conducted separately for the fine-tuned and base LLM models for each test case.

The following table summarizes the average ratings from four experts for each test case, comparing the Fine-tuned LLM and Base LLM outputs for Functional Correctness and Understandability:

- **Average Rating:** Each value represents the average rating from 4 experts based on a Likert scale ranging from 1 to 5.

- **Functional Correctness:** The statechart accurately represents the system's behavior, showing all the necessary states, transitions in line with the requirements.

- **Understandability:** The statechart is easy to follow, with clear and well-organized elements that are simple for experts to understand.

Table 5.4: Comparison of Fine-Tuned and Base Models Based on Expert Ratings

Test Case ID	**Functional Correctness (Base LLM)**	**Functional Correctness (Fine-tuned LLM)**	**Understandability (Base LLM)**	**Understandability (Fine-tuned LLM)**
TC 1	1.75	4.25	2.0	4.25.
TC 2	1.75	3.0	2.5	3.0.
TC 3	2.5	3.25	2.75	3.25.
TC 4	3.25	2.75	3.5	3.75.
TC 5	1.0	3.0	1.0	3.5.
TC 6	2.5	3.0	3.25	3.25.
TC 7	2.5	2.5	3.5	3.25.
TC 8	2.25	3.5	2.75	3.25.
TC 9	4.5	4.5	4.5	4.5.
TC 10	1.75	4.0	2.5	4.25.
TC 11	3.5	4.5	3.25	4.5.
TC 12	4.0	4.0	4.0	4.25.

The results presented in the above table indicate that the fine-tuned model consistently outperformed the base model across all test cases in terms of both *functional correctness* and *understandability*. The best performance was observed for TC9 and TC11, where both models achieved high scores, but the fine-tuned model consistently maintained higher or equal ratings across both evaluation criteria. TC1 also demonstrated strong results, with the fine-tuned model achieving an average score of 4.25, significantly higher than the base model's average of 1.88. However, TC5 reflected the improvement following fine-tuning. While the base model received the lowest possible score of 1.0 across both metrics for TC5, the fine-tuned model improved this to an average of 3.25, indicating improved ability to handle less structured inputs. These results highlight the effectiveness of fine-tuning in improving the quality of statechart generation.

Hypothesis Testing Results

To evaluate the effectiveness of fine-tuning, the Wilcoxon Signed-Rank Test was employed to compare the base and fine-tuned models on two evaluation metrics: functional correctness and understandability.

Functional Correctness

The results of the Wilcoxon Signed-Rank Test revealed a significant difference between the base and fine-tuned models (Statistic = 39.5, p-value = 0.00002). Given that the p-value is well below the threshold of 0.05, we reject the null hypothesis H_0 and conclude that there is a statistically significant difference in performance between the two models.

Model Comparison

Table 5.5: Descriptive Statistics of Functional Correctness Scores

Model	Mean Score	Median Score
Base Model	2.60	2.5
Fine-Tuned Model	3.52	4.0

The fine-tuned model demonstrated higher average functional correctness scores than the base model, with a mean improvement of 0.92 points on the evaluation scale. These results indicate that the fine-tuning process led to a statistically and practically significant improvement in the model's ability to generate functionally correct outputs, reinforcing the effectiveness of the fine-tuning methodology applied in this study.

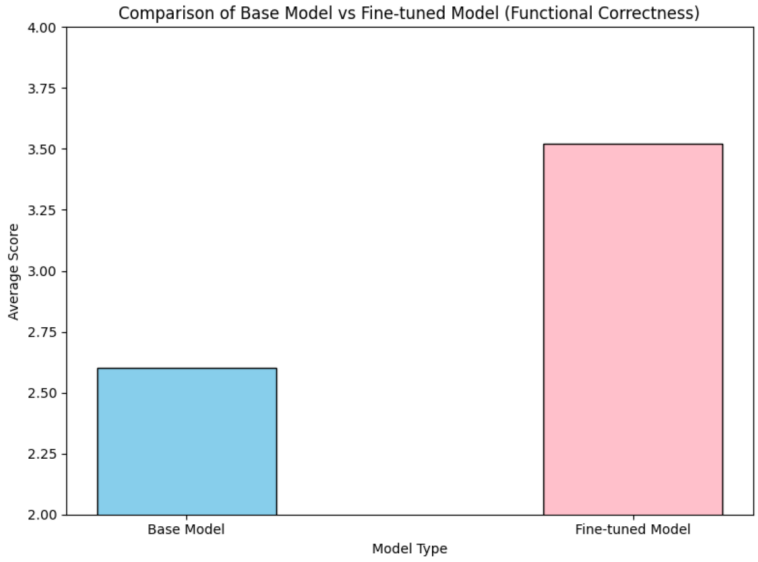

Figure 5.6: Likert scale averages for Functional Correctness

Understandability

The Wilcoxon Signed-Rank Test results for understandability indicated a significant difference between the base and fine-tuned models (Statistic = 56.0, p-value = 0.00036). Since the p-value is below the standard threshold of 0.05, we reject the null hypothesis H_0, concluding that there is a statistically significant difference in understandability between the two models.

Model Comparison

Table 5.6: Descriptive Statistics of Understandability Scores

Model	Mean Score	Median Score
Base Model	2.96	3.0
Fine-Tuned Model	3.75	4.0

These results demonstrate that the fine-tuned model achieved higher average understandability scores compared to the base model, suggesting that it is likely the more effective model in terms of overall understandability.

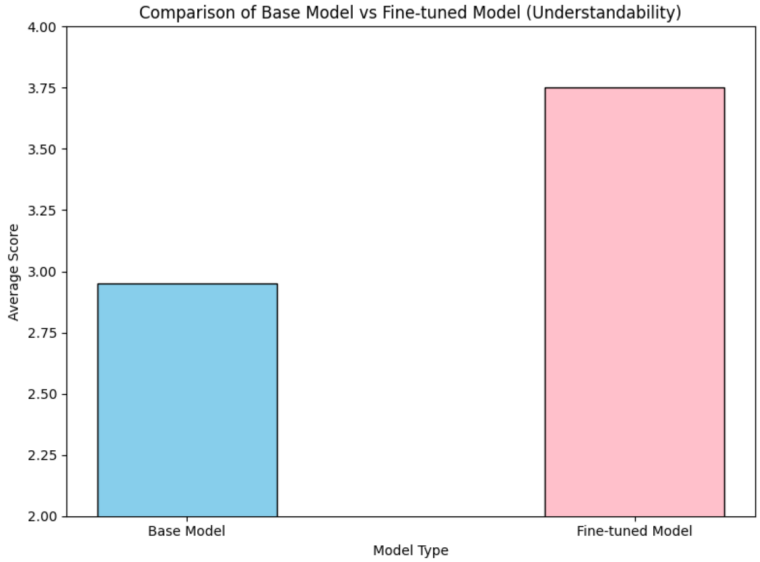

Figure 5.7: Likert scale averages for Understandability

Based on the above results, it is evident that the fine-tuned model is better performing at both functional correctness (average rating = 3.52) and understandability (average rating = 3.75) based on the expert review ratings.

Summary of results: Fine-tuned Vs Non-fine-tuned

Therefore, based on the results of domain experts' ratings, it is statistically evident and demonstrated that the fine-tuned model exhibits higher functional correctness and improved understandability when compared to the non-fine-tuned base model. This verdict proves that fine-tuning an LLM trained with domain-specific data improves its ability to generate accurate and interpretable statecharts, which suggests its reliability in real-world applications where functional correctness and understandability are very critical factors.

5.3 Analysis of Fine-tuned vs Human created statecharts comparison

This section presents a detailed comparison between statecharts generated by a fine-tuned GPT-4 model (via Azure AI Foundry) and those created manually by domain experts. To assess the functional correctness and quality of both sets of statecharts, expert opinions were collected through semi-structured interviews. Four experts were asked to evaluate various attributes of the statecharts (e.g., functional correctness, understandability and Alignment to Requirements) based on 12 test cases derived from textual requirements.

The opinions were collected using a Likert scale with the following options 'Very Unclear, Somewhat Unclear, Neutral, Somewhat Clear, and Very Clear', and the resulting data were analyzed through a statistical method called Wilcoxon Signed-rank test to assess functional correctness and effect size calculations to measure the significance of any differences observed.

Expert Evaluation:

Table 5.7: Expert Evaluation Scores: Average Scores

Metrics	**Average Score**	**Interpretation**
Functional Correctness	2.88	Neutral.
Understandability	2.5	Somewhat Unclear.
Requirement Alignment	2.75	Slight to Moderate.

Statistical Testing: Manual vs Fine-Tuned Model

Wilcoxon Signed-Rank Test

A Wilcoxon Signed-Rank Test was conducted to assess whether there is a statistically significant difference between the scores of the manual model and the fine-tuned model. The test statistic was 0.0, and the p-value was found to be 9.72×10^{-11}.

Given that the p-value is less than the significance level of $\alpha = 0.05$, we reject the null hypothesis H_0. This indicates that there is a significant difference between the two groups, indicating that the fine-tuned model exhibits performance differences compared to the manually created model.

Model Comparison

The comparison of the mean and median scores of functional correctness for the two models is presented in the table below:

Table 5.8: Descriptive Statistics: Manual vs Fine-Tuned Model of functional correctness

Model	**Mean Score**	**Median Score**
Manual Model	4.00	4.00
Fine-Tuned Model	2.81	3.00

The manual model exhibited a higher mean (4.00) and median (4.00) score compared to the fine-tuned model, which had a mean of 2.81 and a median of 3.00. This suggests that, on average, the manual model performs better in terms of the evaluated metric.

However, it is important to note that while the statistical analysis shows a significant difference between the two models, the fine-tuned model may still exhibit benefits in specific contexts or areas that were not captured by the current metric.

Based on the statistical results and the model comparison, we conclude that the manual model has a higher average score. Nevertheless, the fine-tuned model's performance could be further optimized in future iterations. The models are comparable, with the manual model currently outperforming the fine-tuned model in the given evaluation criteria.

Feedback from Experts:

The following table summarizes the expert feedback gathered on the LLM-generated statecharts. It highlights key areas of concern identified by the experts during the interviews, along with suggested improvements.

Table 5.9: Feedback from Experts

Aspect	**Issue Identified**	**Improvement Suggestion**
State Transitions	Incomplete or unclear transitions	Clearer definition of state transitions.
Loop Logic	Difficulty understanding loop conditions and end points	Clearer loop handling and end conditions.
Alignment with Requirements	Generic interpretation of requirements	Adapt output to domain-specific needs.
Terminology Consistency	Inconsistent terms across statecharts	Standardized naming conventions and terminology.

Summary of results: Fine-tuned Vs Manually created statecharts

Therefore, based on the results of semi-structured interviews with domain experts, it is statistically evident and proven that the manually created statecharts are more functionally correct and more understandable when compared to the statecharts generated by the fine-tuned model. Although the fine-tuned LLM demonstrated clear improvements over the base model, especially in handling simpler scenarios, it still lacks the overall quality when compared to the manually created statecharts.

Experts acknowledged that the fine-tuned model was effective in generating

moderately accurate statecharts, particularly for simpler product functions that involve linear or limited state transitions. In such cases, such as TC9 and TC11, the model achieved high ratings and provided noticeable time savings by automating. However, for more complex functions, such as TC5, that include conditional logic, timing constraints, or overlapping state behaviors, the model exhibited limitations. Expert feedback identified specific issues, including incomplete transitions, missing event-handling logic, unclear loop conditions, and ambiguous start or end states.

Overall, while fine-tuning large language models with domain-specific data enhances their performance and usability, particularly for less complicated functions, the findings emphasize that expert involvement remains crucial for validating and refining outputs in safety-critical and behaviorally complex systems. Additionally, the lack of clarity in start/end points and ambiguous state transitions were cited as areas requiring improvement to enhance usability and consistency in professional applications.

5.4 Addressing Research questions

Research Question 1 (RQ1): Does fine-tuning LLMs improve statechart generation?

Hypotheses:

- H_0**: Fine-tuning does not significantly improve performance metrics (Functional correctness and understandability).**
- H_1**: Fine-tuning significantly improves performance metrics.**

To answer whether fine-tuning improves the performance of LLMs in generating statecharts, an expert rating process was conducted involving domain experts. Each of the 12 test cases provided by Volvo cars, was used to generate two statecharts, one using the non-fine-tuned base LLM and the other using the fine-tuned version. Four experts evaluated the outputs based on functional correctness, understandability. The experts consistently preferred the fine-tuned model, stating that its statecharts had better logic and matched the requirements more clearly. To quantitatively support these observations, a Wilcoxon signed-rank test was performed on expert ratings, which revealed a statistically significant difference ($p < 0.05$) in favor of the fine-tuned model. These results indicate that fine-tuning leads to significant improvements in the quality of statechart generation across key performance metrics. This supports the Hypothesis, "H_1: Fine-tuning significantly improves performance metrics."

Research Question 2 (RQ2): How do LLM-generated statecharts compare to manually created ones?

Hypotheses:

- H_0**: LLM-generated statecharts do not achieve comparable levels of functional correctness to manually created statecharts.**
- H_1**: LLM-generated statecharts achieve comparable or higher levels of functional correctness.**

To determine how LLM-generated statecharts compare to manually created ones, a series of expert evaluations was conducted using 12 test cases. The experts assessed the statecharts based on functional correctness, understandability, and alignment with requirements, using a Likert scale. Using the Wilcoxon signed-rank test, the results pointed to a significant difference between the two groups. LLM-generated statecharts consistently scored lower, with a p-value < 0.001, indicating that they did not achieve comparable levels of functional correctness to manually created statecharts. Expert feedback identified issues including unclear transitions, poorly aligned requirements, and inconsistent terminology in the LLM-generated charts. These findings support the hypothesis H_0 that LLM-generated statecharts currently do not match the performance of manually created ones in terms of accuracy and clarity.

6

Discussion

The results of this study provide significant insights into the impact of fine-tuning Large Language Models (LLMs) for generating state charts in the automotive domain. Specifically, the findings from Research Question 1 (RQ1) strongly support the hypothesis that fine-tuning improves performance metrics, such as functional correctness and understandability. Reviews from domain experts, in conjunction with statistical evidence from the Wilcoxon Signed-rank test, demonstrates that the fine-tuned model outperforms the non-fine-tuned version on key performance metrics. These results align with previous research, such as Liu et al.[39] , which has shown that domain-specific fine-tuning enhances the interpretability and accuracy of LLM-generated outputs, particularly in complex domains.

For example, in test cases such as TC9 and TC11, where the system behavior was relatively linear and state transitions were limited and sequential, the fine-tuned model achieved high expert ratings. These charts exhibited clear state transitions, consistent event handling, and syntactic correctness, reflecting the model's capability to capture straightforward functional logic efficiently. Domain experts noted that such outputs would significantly reduce development time in early-stage system modeling.

In contrast, the findings from Research Question 2 (RQ2) indicate that LLM-generated state charts still fall short in terms of accuracy compared to manually created charts. While improvements were observed with the fine-tuned model over the non-fine-tuned version, the LLM-generated charts did not reach the functional correctness or clarity levels of manually crafted state charts. These results are consistent with those of Harel et al.[17] , who also noted that generating statecharts from textual requirements presents challenges due to the complexity of event-driven transitions and dynamic behaviors. In more complex test cases, such as TC5, which involved conditional logic, nested transitions, and timing constraints, the fine-tuned model exhibited notable limitations. Expert reviews highlighted missing transitions, ambiguous loop structures, incomplete event-handling routines, and unclear entry/exit conditions. These deficiencies significantly impacted the usability and functional interpretability of the generated charts.

The statistical difference observed in this study, along with expert feedback, emphasizes the ongoing challenges in generating high-quality state charts automatically. Issues such as unclear transitions, vague requirement interpretations, and inconsistent terminology were highlighted as key factors affecting the overall quality of LLM-generated charts, similar to the challenges identified by Meng and Ban [26] in automating UML diagram generation.

However, despite these challenges, the study highlights the positive aspects of LLM-generated state charts, such as time efficiency in generating simpler charts. This suggests that LLMs could be particularly useful in the early stages of state chart creation, where AI could generate draft versions that could then be refined by human experts. This hybrid approach, combining the strengths of AI and human expertise, is in line with recent findings by Basili et al.[4], who emphasized the value of combining AI-driven methods with human input to enhance model quality and efficiency. The potential for such a collaborative approach in the early phases of model generation aligns with current trends in AI-assisted software development.

In conclusion, this study contributes to the state of the art by validating the potential of LLMs for automating state chart generation in the automotive domain. While LLMs show promise, especially in improving efficiency during initial chart drafting, there remains significant room for refinement, particularly in terms of ensuring the accuracy and clarity of generated state charts. Future research could explore methods to address the challenges identified in this study, such as improving the handling of system dynamics and event-driven transitions, areas that are crucial for generating high-quality state charts. These advancements will build upon the work of previous studies Meng and Ban [26], Harel et al.[17], and Zhong et al.[40] , contributing to the evolution of state chart automation.

6.1 Future Work, Limitations and Threats to Validity

This study provides a meaningful contribution to the application of fine-tuned LLMs for generating statecharts in the automotive industry. However, several limitations must be considered when interpreting the findings, and there are various threats to the validity of the study that should be addressed in future research. These aspects are important for refining the approach and enhancing the applicability of the results to real-world settings.

Threats to Validity

Internal validity is one of the main threats to lies in the limited size and scope of the dataset used for fine-tuning. The dataset included only a small number of requirement samples, which may not fully capture the diversity and complexity of real-world automotive requirements. This could lead to

overfitting, where the model performs well on the requirements but struggles to generalize to new, unseen data. Furthermore, the synthetic data used for training, although useful, might not represent the wide variety of scenarios encountered in practical automotive systems, which could further limit the model's generalizability.

External validity is another concern, as the findings of this study may not directly apply to other industries or system domains beyond the automotive sector. The requirements used were specifically designed around automotive functions, meaning that the results may not hold for other fields or use cases. Expanding the dataset to include a more diverse set of real-world examples, particularly from different sectors, would help enhance the generalizability of the findings.

In terms of **construct validity**, while the study showed improvements in certain areas like functional correctness and understandability, the model faced challenges when dealing with more complex requirements. The LLMs were not able to capture all the intended meaning and detailed specifications that come with automotive systems. This indicates that the models may not yet fully reflect the complex constructs required for real-world applications, particularly those involving dynamic and project-specific requirements.

To address the **conclusion validity**, the study found that LLM-generated statecharts were efficient but not always of the same quality as manually created charts. The functional correctness and clarity of LLM-generated charts did not consistently match those of expert-created charts, suggesting that the conclusion regarding the superiority of the model in statechart generation may require further validation and refinement in future work.

Limitations and Future Work

Several limitations should be considered when interpreting the results of this study. The primary limitation lies in the dataset used for fine-tuning the models. The dataset was relatively small and did not fully account for the complexity and variety of automotive requirements. As such, the models may have been overfitted to the limited requirement samples provided, which could affect their performance when faced with more diverse or complex requirements. Future work should focus on expanding the dataset by including a larger number of real-world cases, which would improve the model's ability to generalize and handle a broader range of requirements.

Another limitation observed in this study is the model's difficulty in handling more complicated and dynamic requirements. Although the fine-tuned models showed improvements in terms of functional correctness and understandability, they struggled with requirements that were complex or had evolving contexts. This suggests that the models need further refinement to better understand and process such complex, project-specific details. Future research could fo-

cus on incorporating more advanced modeling techniques or enhancing the contextual understanding of the models to address these challenges.

Furthermore, while LLM-generated statecharts provided time-saving benefits, the quality of the generated outputs did not always match the level of functional correctness and clarity found in manually created charts. This indicates that further improvements are needed in the generation process. Future work could explore hybrid approaches, combining LLM-generated statecharts with human expert feedback to refine the outputs. This approach would likely improve the alignment of the generated statecharts with the required quality standards, making them more suitable for real-world applications. Although LLMs can generate statecharts more efficiently than human experts, the trade-off between speed and quality needs further investigation. Future work should examine ways to balance these two aspects to ensure that time efficiency does not come at the cost of the functional quality or clarity of the statecharts.

In summary, this study highlights the potential of LLMs for generating statecharts in the automotive industry, while also identifying key areas for improvement. By expanding the dataset, addressing the challenges posed by complex requirements, and exploring hybrid models that combine LLMs with human feedback, future research can significantly enhance the utility and performance of LLM-generated statecharts in real-world applications.

7

Conclusion

In conclusion, this study provides strong evidence that fine-tuning LLMs with domain-specific data can significantly improve the quality of LLM-generated statecharts. The results confirm that fine-tuned models outperform non-fine-tuned models in terms of functional correctness and understandability, making them a useful tool for generating statecharts in automotive engineering and similar domains. However, while the fine-tuned LLMs show noticeable improvements over the baseline, they still do not match the functional correctness and clarity of manually created statecharts, primarily due to issues with requirement alignment and inconsistency in terminology. These limitations indicate that LLM-generated charts are not yet ready to fully replace human expertise in complex, domain specific use cases.

Overall, the findings suggest that LLM-generated statecharts offer potential for time efficiency and ease of use, particularly for less complex scenarios. However, further advancements are needed to improve contextual understanding and requirement alignment to make LLM-generated charts more reliable and suitable for professional use.

Bibliography

[1] E. A. Abdelnabi, A. M. Maatuk, and M. Hagal. Generating uml class diagram from natural language requirements: A survey of approaches and techniques. In *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA)*, pages 288–293, Tripoli, Libya, 2021.

[2] Itemis AG. Yakindu statecharts, 2024. [Online]. Available: `https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide`.

[3] A. A. Almazroi, L. Abualigah, M. A. Alqarni, E. H. Houssein, A. Q. M. AlHamad, and M. A. Elaziz. Class diagram generation from text requirements: An application of natural language processing. In V. Kadyan, A. Singh, M. Mittal, and L. Abualigah, editors, *Deep Learning Approaches for Spoken and Natural Language Processing*, Signals and Communication Technology. Springer, Cham, 2021.

[4] V.R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[5] Weights & Biases. Weights & biases for academic research, 2025.

[6] A. Borshchev and I. Grigoryev. *Chapter 7. Designing State-Based Behavior: Statecharts*, pages 287–319. AnyLogic North America, May 31 2015.

[7] Volvo Cars. Background on product validation department. In *Thesis Work: Generate Statechart from the Written Requirements Using AI*, unpublished internal document, 2024.

[8] John W Creswell and J David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[9] J. D'Ambrosio and G. Soremekun. Systems engineering challenges and mbse opportunities for automotive system design. In *2017 IEEE Inter-*

national Conference on Systems, Man, and Cybernetics (SMC), pages 2075–2080, Banff, AB, Canada, 2017.

[10] Alexander Dunn, John Dagdelen, Nicholas Walker, Sanghoon Lee, Andrew S Rosen, Gerbrand Ceder, Kristin Persson, and Anubhav Jain. Structured information extraction from complex scientific text with fine-tuned large language models. *arXiv preprint arXiv:2212.05238*, 2022.

[11] Sunitha Ev Ev and Philip Samuel. Automatic code generation from uml state chart diagrams. *IEEE Access*, PP:1–1, 01 2019.

[12] Cheng Fan, Meiling Chen, Xinghua Wang, Jiayuan Wang, and Bufu Huang. A review on data preprocessing techniques toward efficient and reliable knowledge discovery from building operational data. *Frontiers in energy research*, 9:652801, 2021.

[13] A. Ferrari, S. Abualhaija, and C. Arora. Model generation from requirements with llms: An exploratory study. In *Proceedings of the International Requirements Engineering Conference*, 2023.

[14] R. R. Ferreira, A. B. R. Viana, C. A. L. Lisbôa, L. Carro, and F. R. Wagner. Reliable execution of statechart-generated correct embedded software under soft errors. In *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 147–152, Warsaw, Poland, 2014.

[15] Maurice A. Geraghty. Post-hoc analysis: Tukey's honestly significant difference (hsd) test, 2022.

[16] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[17] David Harel, Hillel Kugler, and Amir Pnueli. *Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements*, pages 309–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[18] Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X Wang, and Sadid Hasan. Does prompt formatting have any impact on llm performance? *arXiv preprint arXiv:2411.10541*, 2024.

[19] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

[20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spacy: Industrial-strength natural language processing in python. 2020.

[21] Qianyu Huang and Tongfang Zhao. Data collection and labeling techniques for machine learning. *arXiv preprint arXiv:2407.12793*, 2024.

[22] Christian Kaiser, Alexander Stocker, Andreas Festl, Marija Djokic Petrovic, Efi Papatheocharous, Anders Wallberg, Gonzalo Ezquerro, Jordi Ortigosa Orbe, Tom Szilagyi, and Michael Fellmann. A vehicle telematics service for driving style detection: Implementation and privacy challenges. In *Proceedings of the 6th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*, pages 29–36, 2020.

[23] Vladik Kreinovich. Why 70/30 or 80/20 relation between training and testing sets: A pedagogical explanation. Technical report, University of Texas at El Paso, 2014.

[24] Zhuoyan Li, Hangxiao Zhu, Zhuoran Lu, and Ming Yin. Synthetic data generation with large language models for text classification: Potential and limitations. *arXiv preprint arXiv:2310.07849*, 2023.

[25] MathWorks. Stateflow, 2024. [Online]. Available: `https://se.mathworks.com/products/stateflow.html`.

[26] Yang Meng and Ainita Ban. Automated uml class diagram generation from textual requirements using nlp techniques. volume 8, 2024.

[27] Tom Mens, Alexander Decan, and Nikolaos I. Spanoudakis. A method for testing and validating executable statechart models. *Software and Systems Modeling*, 18:837–863, 2019.

[28] Mermaid.js. Mermaid: Generation of diagrams and flowcharts, 2025.

[29] OpenAI. Gpt-4, 2024. [Online]. Available: `https://openai.com/research/gpt-4`.

[30] OpenAI. Openai api models documentation, 2024. [Online; accessed 13-May-2025].

[31] Susan M. Ross. One-way anova. In *Introductory Statistics*, pages 159–185. Springer, 2017.

[32] Kurt Schneider, Farnaz Fotrousi, and Rebekka Wohlrab. A reference model for empirically comparing llms with humans. *IEEE — Accepted to the 47th International Conference on Software Engineering Companion (ICSE-C), Software Engineering in Society Track (SEIS)*, 2025.

[33] Sparx Systems. Enterprise architect, 2024. [Online]. Available: `https://sparxsystems.com/`.

[34] Amit Thombre. A new statistical test and its comparison with the wilcoxon signed rank test, 06 2024.

[35] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.

[36] Sandeep Vemuri, Sisay Chala, and Madjid Fathi. Automated use case diagram generation from textual user requirement documents. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–4, 2017.

[37] Volvo Cars Internal. *How to find requirements and guidelines in CarWeaver*, 2025. Internal tool, not publicly available.

[38] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. *Experimentation in software engineering*, volume 236. Springer, 2012.

[39] Songheng Zhang, Lei Wang, Toby Jia-Jun Li, Qiaomu Shen, Yixin Cao, and Yong Wang. Chartifytext: Automated chart generation from data-involved texts via llm. *arXiv preprint arXiv:2410.14331*, 2024.

[40] Shaohong Zhong, Andrea Scarinci, and Alice Cicirello. Natural language processing for systems engineering: Automatic generation of systems modelling language diagrams. *Knowledge-Based Systems*, 259:110071, January 2023.

A

Appendix

Appendix 1: Domain Expert Evaluation of AI-Generated Statecharts Using Likert Scale

Demographic Information

1. Full Name: _____
2. Email: _____
3. Job Title: _____
4. How many years have you been in your current position? _____
5. How many years have you been with this company? _____

Evaluation Instructions

For each of the 12 test cases, please rate the **functional correctness** and **understandability** of the AI-generated statecharts using the following Likert scale:

Rating	Description
1	Very Poor
2	Poor
3	Fair
4	Good
5	Excellent

Note: You will be asked to give two ratings per test case – one for Functional Correctness and one for Understandability.

Appendix 2: Domain Expert Feedback on AI vs. Manual Statecharts (Semi-Structured Interviews)

Interview Questions

1. Could you briefly describe your background and experience with statecharts or system modeling?

2. When evaluating the quality of a statechart, which characteristics do you consider most important (e.g., functional correctness, understandability, efficiency)?

3. How would you rate the **functional correctness** of the AI-generated statecharts compared to the manually created ones?
   (Very Poor, Poor, Neutral, Good, Excellent)

4. How would you rate the **understandability** of the AI-generated statecharts compared to the manually created ones?
   (Very Unclear, Somewhat Unclear, Neutral, Somewhat Clear, Very Clear)

5. How closely do the AI-generated statecharts align with the original textual requirements?
   (Not Aligned at All, Slightly Aligned, Moderately Aligned, Mostly Aligned, Fully Aligned)

6. Were there specific elements or behaviors within the AI-generated statecharts that you found unclear, incorrect, or inconsistent? (Select all that apply)

   - Naming Conventions
   - States/Transitions
   - Logic Flow
   - Visual Layout
   - Terminology
   - Other (please specify): _____

7. How usable do you find the AI-generated statecharts for tasks such as system design, validation, or documentation?
   (Not Usable, Slightly Usable, Moderately Usable, Very Usable, Fully Usable)

Follow-up: What modifications or efforts would be required to make them fully usable?

---

8. How much time or effort would you estimate is required to adapt the AI-generated statecharts for actual professional use?

9. Based on your expertise, what improvements would you recommend to enhance the quality or usefulness of the AI-generated statecharts?

10. Is there anything else you would like to share about the comparative quality or relevance of the statecharts you reviewed?

Appendix 3: statecharts of finetune and manual created statecharts

Test case 11: mirror adjustments

1. Users can adjust the mirrors using central display buttons.

2. There are 6 buttons: users can select adjusting the right mirror, left mirror, or adjust directionally toward up, down, left, or right.

3. When the left mirror is selected, the up, down, left, or right buttons will affect the movement of the left mirror.

4. When the right mirror is selected, the up, down, left, or right buttons will affect the movement of the right mirror.

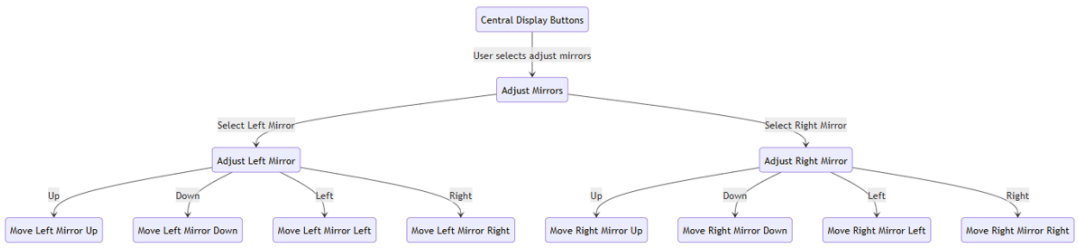

Figure A.1: Statechart of Test case 11 for finetune model

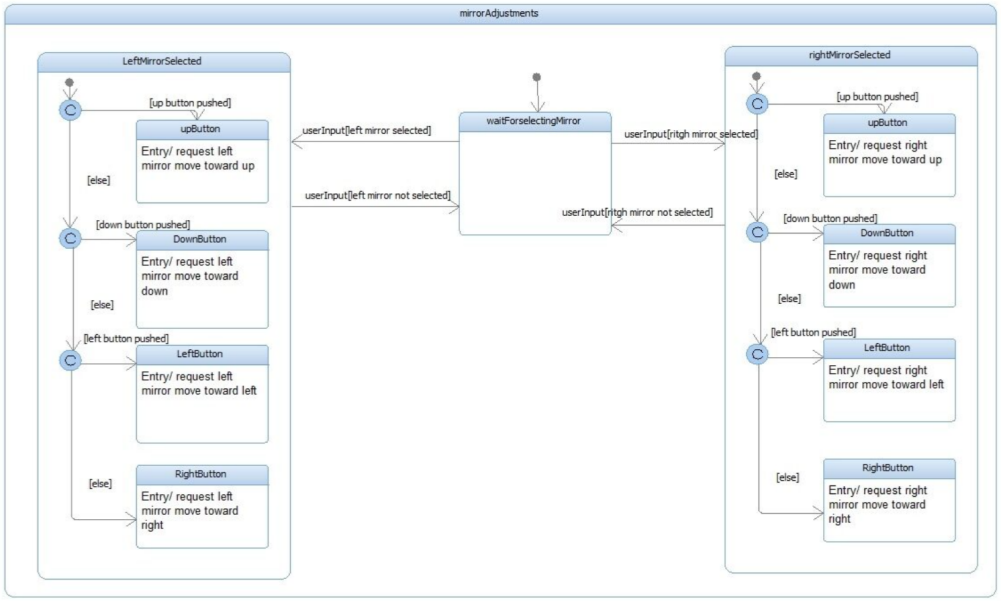

Figure A.2: Statechart of Test case 11 for Manual created statechart

Test case 9: Hood frunk opening closing holding

1. Hood has 3 states. Fully open, open and in secondary position (means hood is open but safety mechanism need to be released to be fully open), and close.

2. Users can open the hood to the secondary position using the hood handle inside the car.

3. If the hood is closed, by pulling the handle, the hood opens to the secondary position.

4. If the hood is fully open or in secondary position, by pulling the handle nothing happens.

5. User can fully open the hood if the hood is in secondary position by releasing the safety mechanism.

6. User can close the hood when the hood is in fully open state or when hood is in secondary position.

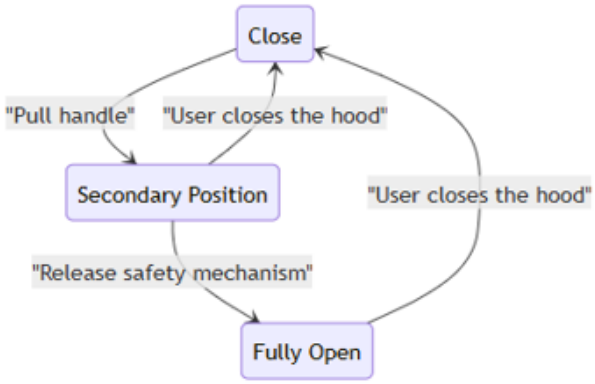

Figure A.3: Statechart of Test case 9 for finetune model

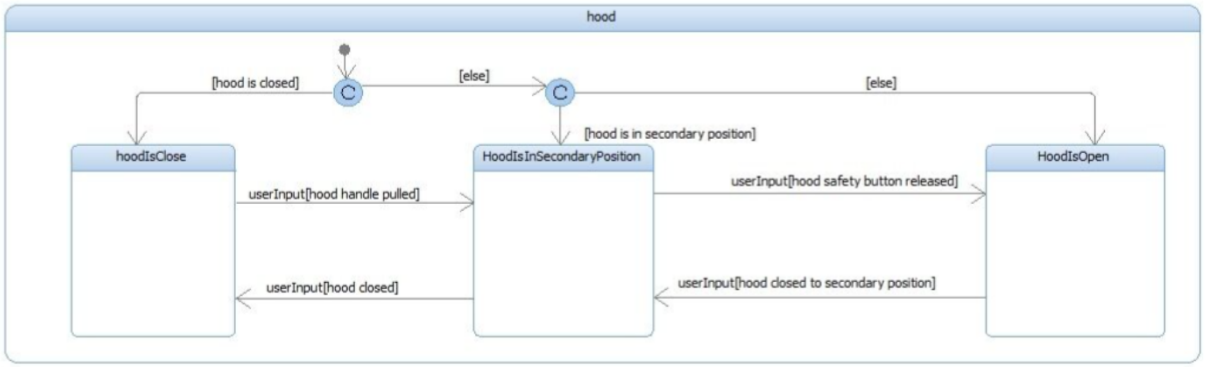

Figure A.4: Statechart of Test case 9 for Manual created statechart

Test case 5: windscreen wiping

1. Windscreen wipers can be started either manually or automatically.
2. It should be possible to activate windscreen wiper manually using right stalk to 3 different levels. It's either off, or level 1, or level 2 or level 3.
3. Level 1 means 2 seconds interval between each wipe.
4. Level 2 means 1 second interval between each wipe.
5. Level 3 means 500 milliseconds interval between each wipe.
6. It should be possible to activate the windscreen wiper automatically using the right stalk.
7. Users can set the wiper sensitivity in central display in 3 different levels. Low, medium and high.

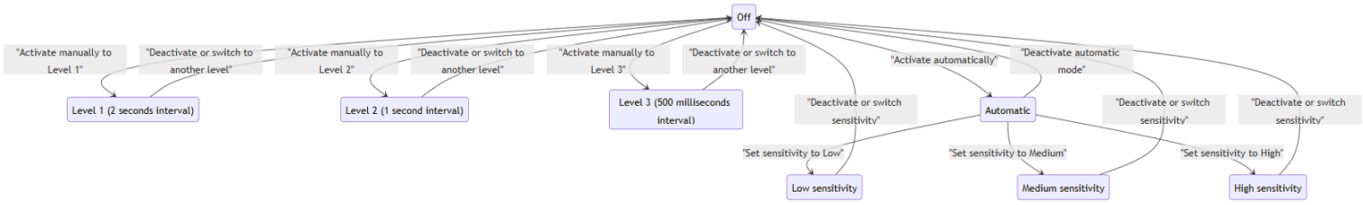

Figure A.5: Statechart of Test case 5 for finetune model

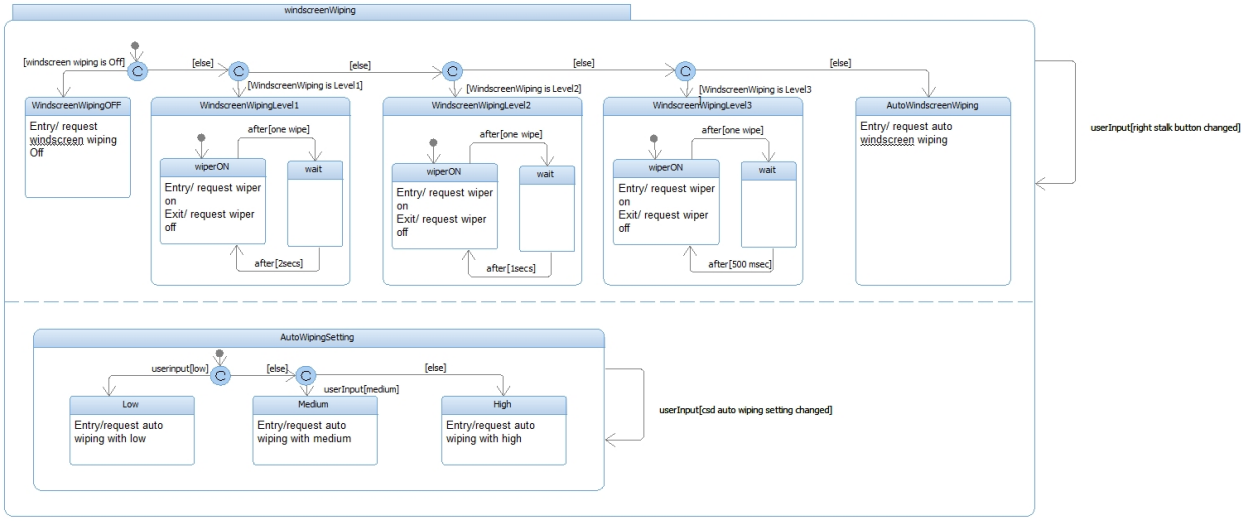

Figure A.6: Statechart of Test case 5 for Manual created statechart