Generating APIs through Library Learning using Large Language Models

Master's Thesis in Computer Science and Engineering

Erdem Halil

Shaphan Manipaul Sam Kirubahar

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

Master's Thesis 2025

Generating APIs through Library Learning using Large Language Models

Erdem Halil
Shaphan Manipaul Sam Kirubahar

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Generating APIs through Library Learning using Large Language Models
Erdem Halil & Shaphan Manipaul Sam Kirubahar

© Erdem Halil & Shaphan Manipaul Sam Kirubahar, 2025.

Supervisor: Yinan Yu, Department of Computer Science and Engineering
Advisor: Dhasarathy Parthasarathy, Volvo Trucks
Examiner: Hans-Martin Heyn, 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

Generating APIs through Library Learning using Large Language Models
Erdem Halil & Shaphan Manipaul Sam Kirubahar
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The development of APIs in vehicular systems, such as trucks, is a manual and labor-intensive process that involves multiple teams and iterative coordination, which makes it a prime candidate for automation. In this thesis, we develop a SPAPI Coder, a system that synthesizes high-level API endpoint implementations from low-level Controller Area Network (CAN) signals and OpenAPI specifications. Inspired by the manual workflow but restructured for automation using inductive program synthesis and library learning techniques, the system decomposes the manual software development workflow into three more manageable abstraction levels.

SPAPI Coder utilizes LLMs for code generation at each level while employing a Retrieval-Augmented Generation (RAG) pipeline for efficient and accurate mapping of API properties to CAN signals. The system features automated test script generation and execution for LLM-produced code, as well as LLM-assisted judging for evaluation.

The results of the study indicate that the proposed system can successfully generate functional API components if correct prompt engineering, library learning, and program synthesis techniques are used. While LLMs show promise as evaluators, their standalone accuracy for code evaluation tasks remains a limitation.

Findings suggest automating API development in complex domains like automotive systems using LLMs and program synthesis is viable. The approach feasibly addresses all dimensions of program synthesis and demonstrates the potential of library learning to automate API generation. It also provides a foundation for future work in incremental synthesis and intelligent software process automation.

Keywords: LLMs, Automotive Software, Software Engineering, Library Learning, Program Synthesis

Acknowledgements

We extend our sincere gratitude to everyone who supported and guided us throughout the course of this thesis.

First and foremost, we would like to thank our industrial supervisor at Volvo Group, Dhasarathy Parthasarathy, for his continued expert guidance and insights. Without him constantly challenging our work, there would not have been a research of this breadth and depth.

Special thanks go to our academic supervisor Yinan Yu, and Shuai Wang for providing constructive feedback and support during the thesis. Their comments were invaluable in refining our work.

We are also thankful to Hans-Martin Heyn, our examiner, for his rigorous review and evaluation of our work.

Last but not least, we would like to express our deepest acknowledgements to our circle of family and friends, whose selfless encouragement kept us going until the end.

Erdem Halil & Shaphan Manipaul Sam Kirubahar, Gothenburg, June 2025

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1
1.1 Problem Description . . . 2
1.2 Purpose of the Study . . . 3
1.3 Significance of the Study . . . 4
1.4 Research Questions . . . 5

2 Background 7
2.1 Automotive E/E system . . . 7
2.2 Controller Area Network . . . 7
2.3 Connectivity and Automotive APIs . . . 8
2.4 APIs . . . 9
2.4.1 API-first design . . . 9
2.4.2 OpenAPI Specifications . . . 10
2.4.3 RESTful APIs . . . 10
2.4.4 REST APIs in vehicles . . . 10
2.5 Large Language Models . . . 11
2.5.1 LLM Parameters . . . 11
2.5.2 Prompting . . . 12
2.6 The LLMs at hand . . . 13
2.6.1 GPT-4o . . . 13
2.6.2 Llama Models . . . 13
2.6.3 Qwen 2.5 Coder 32B . . . 13
2.6.4 DeepSeek R1 . . . 14
2.7 Retrieval-Augmented Generation . . . 14
2.7.1 Embedding Models . . . 15
2.7.2 Reranker Models . . . 16
2.7.3 Vector Databases . . . 16
2.8 LLM Agents . . . 17
2.9 LLMs as Judge . . . 18
2.10 Compound AI Systems . . . 19
2.11 Automatic Programming . . . 20
2.11.1 Program Synthesis . . . 20

2.11.2 Inductive Programming 21
2.11.3 Key Dimensions of a Synthesizer 21
2.11.4 Program Synthesis with ML Models 22
2.11.5 Program Synthesis with LLMs 22

3 Related Work 23
3.1 Library Learning 23
3.1.1 Library Learning Complements Program Synthesis 23
3.2 LLMs for Programming in Software Engineering 24
3.3 LLM-based Systems in API Development 24
3.4 Gaps in Research 25
3.4.1 Program Synthesis in Software Workflow Automation 25
3.4.2 Enhancing AI-powered Program Synthesis 25
3.5 Addressing the Gaps 25

4 Methods 27
4.1 Research Methodology 27
4.2 Automating SPAPI Development 28
4.2.1 Decomposing the Manual SPAPI Development Process 28
4.2.1.1 The Guiding Premise for Process Decomposition . . 28
4.2.2 Decomposed Levels of the Manual SPAPI Development Process 29
4.2.3 The Ideal SPAPI Development Automation System 30
4.3 Data Sources 31
4.3.1 CANdb 31
4.3.2 OpenAPI Specifications 31
4.3.3 Proprietary Data Sources 32
4.3.4 Ground Truth Data for Testing and Evaluation 32
4.3.4.1 Signal-Property Mappings 32
4.3.4.2 API Endpoint Test Scripts 32
4.4 Level 3: Signal Abstraction 32
4.4.1 Data Preprocessing: CAN Signal Representation 33
4.4.2 SPAPI Coder at Level 3 33
4.4.3 SPAPI Tester at Level 3 34
4.4.4 Data Postprocessing 35
4.5 Level 2: Signal - API Property Mapping 35
4.5.1 Data Preprocessing: API Property Representation 35
4.5.2 SPAPI Coder at Level 2 35
4.5.3 SPAPI Tester at Level 2 37
4.5.4 Data Postprocessing 37
4.6 Level 1: SPAPI Endpoint Implementation 37
4.6.1 Data Preprocessing: API Router Boilerplate 37
4.6.2 SPAPI Coder at Level 1 38
4.6.3 SPAPI Tester at Level 1 38
4.6.3.1 Data Preprocessing: Sanitizing of API Test Scripts . 39
4.6.3.2 Testing SPAPI Endpoints 39

5 Results 41

5.1 Experiment Scope 41
5.2 Experiment Design 42
5.3 Level 3 43
5.4 RAG 44
5.5 Level 2 46
5.6 Level 1 48

6 Discussion **53**
6.1 Alignment with Design-Science Research 53
6.2 Research Questions 54
6.2.1 RQ1: Capturing User Intent 54
6.2.2 RQ2: Search Space and Search Technique 55
6.2.3 RQ3: Evaluating Synthesized Artifacts 56
6.3 Threats to Validity 57
6.3.1 Conclusion Validity 57
6.3.2 Internal Validity 57
6.3.3 External Validity 58
6.4 Future Work 58

7 Conclusion **61**

Bibliography **63**

A Appendix **I**
A.1 Example OpenAPI specification I
A.2 Example CAN signal information III
A.3 Example Level 3 code III
A.4 Example Level 2 code IV
A.5 Example Level 1 code V
A.6 Prompt for signal interface synthesis VI
A.7 Prompt for property synthesis VII
A.8 Prompt for mapping correctness judge VIII
A.9 Prompt for specification alignment judge IX
A.10 Prompt for API router synthesis X
A.11 Example Level 3 test case XI
A.12 Example Level 1 test case XII

List of Figures

2.1 Example of domain-centralized E/E system in a truck 8
2.2 A high-level overview of a simple LLM agent 18

4.1 Architecture of the system that automates the SPAPI development and testing process 31
4.2 Architecture of the system at the CAN signal interface level 33
4.3 Architecture of the system at the CAN signal to property mapping level 36
4.4 Architecture of the system at the API router level 38
4.5 System workflow for SPAPI automation 39

List of Tables

2.1 Language model parameters, context length, and source comparison . 14
2.2 Embedding models' supported dimensions and parameters comparison 16

5.1 Level 3 LLM performance evaluation 44
5.2 Retrieval evaluation 46
5.3 Performance metrics for the judges while generating API properties . 47
5.4 Number of correctly generated API properties per LLM 48
5.5 API endpoint test results with LLM performance 49
5.6 API-level test results summary with LLM performance 51

1

Introduction

The rapid advancement of generative Artificial Intelligence (GenAI) and particularly Large Language Models (LLMs) in recent years has revolutionized our ability to interpret, generate, and process natural language data. These generative language models, trained on large corpora of text, excel in their ability to generate text in multiple forms to tackle different problems. Powerful Machine Learning (ML) models have been demonstrating their ability to automate complex tasks across a wide range of domains, including and especially software engineering and development [3, 33]. Among the substantial applications in software engineering where traditional ML frameworks are used, one emerging application area where LLMs excel is in their ability to generate code [33]. The use of LLMs in software engineering is further growing proficient by the day in applications of varying complexities, ranging from code completion for simple coding tasks, to synthesizing programs to solve more advanced problems [3].

Another technological advancement over the last few years has been happening in the automotive industry, where vehicle manufacturers are offering more and more software as services, not only for the entertainment of the end user, but also to control vehicular systems at ease. Nowadays, automotive software in the form of infotainment systems and connected mobile applications has become quite popular among consumers, making it a strong selling point for manufacturers. Modern trucks and vehicles, in general, rely on a large number of Controller Area Network (CAN) signals to transmit and receive data about different components of the vehicle. This facilitates quick and efficient communication across multiple Electronic Control Units (ECUs) through a CAN bus [42]. In the domain of vehicular systems, and particularly in heavy duty vehicles like trucks, the integration of Application Programming Interfaces (APIs) to act as a gateway between the client applications used by the user and the CAN bus, opens up opportunities to provide users with services to not only get vital information about several complex components of the truck, but also control such components with ease.

APIs that adhere to the Representational State Transfer (REST) architecture [18] play a crucial role in enabling seamless communication between various components of a system. REST API documentation in the form of OpenAPI specification (OAS) [30], formerly known as Swagger documents, has become the industry standard for designing and documenting web microservices due to their consistency and scala-

bility. As such, state-of-the-art LLMs are trained on a large corpus of code from different datasets and OpenAPI specifications, reflecting their comprehensive nature, just like the "whole" internet being a massive data repository for people to source information from. So, these models can identify underlying patterns and relations between API specifications and API implementations in code [11]. Besides software engineers, this shows potential for software engineering workflows that incorporate an LLM, the best agents to synthesize and test REST API endpoints [88].

LLMs have shown substantial progress in tackling several problems in software engineering [33]. But, the complete potential and limitations of the use of such models as part of a system that automates a conventional software development workflow in a complex industrial setting [88], remain a frontier of research. This thesis examines the common ground and probes the connections between the generative capabilities of LLMs, REST APIs, and low-level communication protocols in trucks, by developing a system that automates the process of synthesizing functional REST API implementations through library learning [17, 21]. The proposed solution will address the challenges of mapping CAN signals to their corresponding API definitions, so as to create and test these high-level API implementations. Building such a software system will not only aim to synthesize APIs that bridge the gap between hardware-level communication and human-readable interfaces, but also make the entire process (semi-)automatable. The successful realization of this work could streamline the process of API development in vehicular systems, and also set the stage for similar innovations in the Software Development Life Cycle (SDLC) with the use of such compound AI systems [96] in other embedded systems domains and software engineering in general.

1.1 Problem Description

In modern Volvo trucks, an internal web server exposes SPAPIs (REST APIs) to client systems that are Android applications. Each API communicates with different ECUs through the CAN bus, which enables communication between different vehicle components by transmitting or receiving a set of CAN signals. These CAN signals are used to access or change the states of a particular component of the vehicle. For instance, the *CabinClimate* API exposes the climate control related CAN signals as an API. This allows the driver in a truck to control the cabin climate through an Android application.

The development of SPAPI endpoints is a manual and labor-intensive process that involves multiple teams and iterative coordination. Developing an SPAPI endpoint begins with gathering requirements from an application team, which are formalized in a Swagger specification (OpenAPI 2.0) Specification. For instance, consider that the *CabinClimate* application team is tasked with enabling users to adjust the truck's cabin climate. The requirements are provided to the SPAPI team in the form of an OAS, detailing the desired API endpoints and their properties. Then, based on these requirements, the engineers in the SPAPI team coordinate with the control system teams, responsible for specific ECUs and their associated CAN signals, to map each

requirement in the form of an API property from the OAS to its corresponding CAN signal(s).

For example, to support a feature for changing the air conditioning mode, the OAS might specify an API property ***acMode*** with possible values ***[ON, OFF]***. SPAPI engineers, in coordination with control system teams, map the property and its corresponding values to the CAN signals ***[ACMode.On, ACMode.OFF]***, drawing on documentation such as the CANdb (CAN Database) or proprietary information sources. This mapping process is repeated for each property in the OAS. Subsequently, another set of SPAPI engineers implements the API endpoints based on these mappings. And once the endpoint implementations are developed, the SPAPI testing team validates the implemented endpoints to ensure that they meet the specified requirements. If successful, this lengthy process results in the ***/cabinclimate*** endpoint being deployed and able to provide the *CabinClimate* application with the ability to monitor or adjust the vehicle's climate states.

This workflow involves multiple teams, including the application (UI) team, SPAPI team, control system teams and the SPAPI testing team, each contributing specialized knowledge. The process demands precise coordination to align the OAS requirements with the underlying CAN signals and ensure functional endpoints.

At Volvo, managing over 100 API endpoints requires the efforts of 15–20 full-time engineers, further showing the scale of the task. Additionally, the process is complicated by the need for continuous coordination among teams. This communication relies on a mix of formal specifications (e.g., OAS and CAN databases) and informal, team-specific documentation. Translating these specifications into fully-implemented, well-tested, and usable endpoints requires significant effort from engineers, especially when requirements change, new features are added, or existing APIs are refactored. These updates often trigger a cascade of tedious revisions across mappings and implementations.

To mitigate this problem, automating this otherwise cumbersome and complex process raises the need for a system that synthesizes functional vehicular function API endpoints which adhere to the API specifications, by mapping the API properties to its corresponding CAN signals, combining several data pipelines and the program synthesis capabilities of LLMs [3, 33].

1.2 Purpose of the Study

The primary purpose of this thesis is to explore the potential of developing a system that automates the process of synthesizing API endpoint implementations by leveraging the programming capabilities of state-of-the-art LLMs and library learning characteristics [21, 17].

The manual development of SPAPI endpoints is a repetitive but consistent process that makes it a prime candidate for automation, particularly in an industrial setting where numerous client applications require tailored APIs. The repetitive nature

of this workflow, alongside the semi-structured yet diverse data sources involved, presents significant opportunities for automation. The recent yet swift advancements in LLMs' capabilities, as part of complex agent-like systems, open doors for such workflow automation. The automation of such a workflow was not seen as feasible in the past, as traditional ML models necessitate curating data and training an ensemble of models to perform the same simple tasks.

By analyzing the manual process, patterns can be identified, such as the consistent mapping of API properties to CAN signals or the generation of endpoint boilerplate code, that can inform the design of an automated system. The goal is not only to replicate the manual process but to rethink it by leveraging insights from the current workflow to develop a more efficient, scalable approach for automation. Such a software system, to synthesize useful and functional API endpoints, must be able to emulate the conventional processes involved in SPAPI development, such as the mapping of low-level CAN signals in the trucks, leading to the synthesis of high-level API endpoints through different levels of abstractions.

Since the system automates a software engineering workflow using LLMs, evaluating the correctness and functionality of the LLM generations becomes an absolute necessity. This is because LLMs can be non-factual and produce undesired results at times [35]. This renders evaluation and testing of the LLM-generated artifacts an additional, yet crucial requisite of the system to not only measure its effectiveness in arriving at the required API definitions, but also to measure the functionality and usefulness of the generated API endpoints. An effective system for API development automation should be able to produce functional API endpoints.

1.3 Significance of the Study

This study aims to automate the process of designing and creating API endpoints using program synthesis and library learning characteristics. In practice, doing so will significantly reduce the development time for SPAPI design and implementation. The successful implementation of this pipeline will set the stage for the use of such a system in other software engineering workflows that fall under the scope of automation.

When it comes to significance in the approach used for the synthesis of these API endpoints, as of now library learning characteristics have been studied upon and applied onto several traditional ML models to learn new programs, in order to solve predefined problems and tasks using domain-specific languages [21, 4, 9, 16]. The application of these characteristics in systems that automate software engineering workflows to develop API endpoints using program synthesis still remains an avenue to be further explored, making this a novel study.

1.4 Research Questions

With the ultimate aim of creating a system that synthesizes API endpoints, this thesis aims to answer the following research questions:

- **RQ1:** *What is an effective way to capture the user intent so as to synthesize functionally valid API endpoints?* There arises a need in program synthesis for the system to capture the user's intent for which a program is being synthesized, based on its formal specifications and constraints [9]. To synthesize functional high-level API endpoints, it is crucial that the system does the correct API property to CAN signal mappings. A system that captures user intent in an effective way must be able to map the signals to its API properties accurately. If not, the synthesizer may tend to make incorrect mappings. Hence, it is crucial to study the means by which the CAN signals are presented to the system so that it captures the intent and makes the correct mapping.

- **RQ2:** *How can an effective search space be constructed to enable both efficient and accurate synthesis of API endpoints? What technique can be used to perform a computationally efficient search within this space?* Synthesizing programs requires the system to perform a search over a large space of potential program candidates to come to a solution. An effective search space in this context is one that is constrained enough to reduce the number of irrelevant or invalid candidates while still retaining the expressiveness needed to cover valid solutions. In the case of synthesizing APIs, each API property should be mapped to its corresponding CAN signal(s) over a large selection of signals. This raises the need to investigate how an effective search space of CAN signal abstractions can be constructed. Furthermore, an efficient search refers to a strategy that minimizes computational overhead in terms of time while maximizing the likelihood of finding correct, functional mappings early in the process. This research investigates how to structure the search space to reflect domain-specific constraints and traverse it efficiently to meet synthesis goals.

- **RQ3:** *How can the quality and utility of the generated API endpoints be evaluated?* In the event that the system that automates the API development process is able to synthesize API endpoints successfully, it becomes trivial to verify whether the LLM-synthesized API endpoints are valid and functional. Also, evaluating whether the LLM-generated artifacts follow the specification by which it was synthesized emphasizes the need for a robust verification solution.

2

Background

2.1 Automotive E/E system

Automotive Electrical/Electronic (E/E) systems form the backbone of modern vehicles, integrating hardware, software, and networking technologies to control, monitor, and enhance vehicle functionality. Having started as simple electrical circuits for ignition systems and basic lighting, these systems have since evolved into highly complex and software-driven architectures, thanks to advancements in semiconductor technology and software engineering. The central element of the E/E system is the ECU, which is in the form of a microcontroller. Sensors and actuators in a vehicle are connected to various ECUs, and these enable converting physical phenomena to the digital domain and vice versa.

For instance, there could be a dedicated *CabinClimate* ECU which is connected to a temperature sensor. If the value read from the sensor exceeds a certain threshold, it could trigger the cabin control actuator and turn the air conditioning on/off. These ECUs communicate via standardized in-vehicle networks, including CAN, Local Interconnect Network (LIN), and high-speed Ethernet. These networks ensure efficient real-time data exchange and system coordination. Today, modern premium vehicles could feature over 100 ECUs interconnected with thousands of signals transmitted through the different network buses [39]. Each of these ECUs is dedicated to specific functions such as powertrain optimization, advanced driver-assistance systems (ADAS), and battery management in electric vehicles (EVs). Moreover, the system architecture is often of a hierarchical nature, meaning that there could be multiple "main" ECUs serving a gateway purpose to other ECUs within a particular domain, as shown in Figure 2.1.

2.2 Controller Area Network

The Controller Area Network is a communication protocol that enables ECUs and other components in a vehicle to communicate with each other. ECUs usually communicate by transmitting/receiving CAN Messages. Each CAN Message encapsulates a certain number of CAN signals, each representing a certain vehicle state [42]. A CAN database, which is commonly referred to as CANdb, is a text file that adheres to a format known as DBC. This *DBC* file contains information to inter-

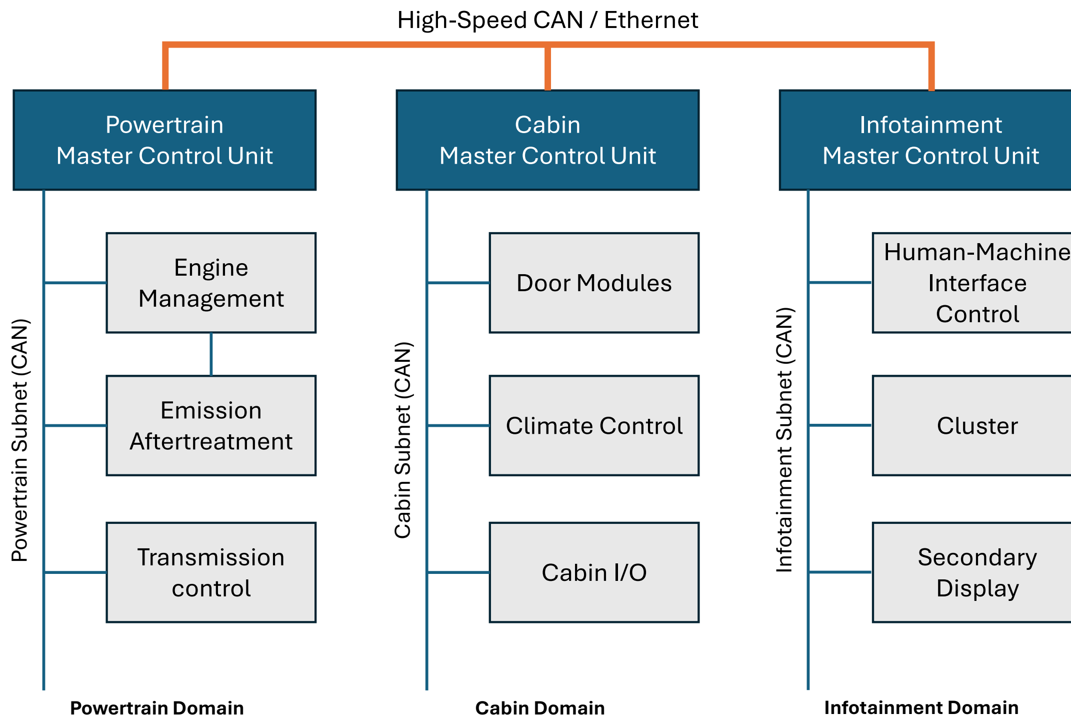

Figure 2.1: Example of domain-centralized E/E system in a truck

pret the data that is transmitted through a CAN bus. This includes information about CAN messages, CAN signals encapsulated in messages that are used to get information about certain vehicle states or update them. Also, information about the frequency at which a signal is transmitted across the bus, which ECUs send and receive a particular signal, etc, is present in a CANdb. Hence, this file plays a huge role in developing and testing systems in a vehicle that relies on CAN bus communication. Even though CANdb provides a standardized way of documenting the decoding information of CAN bus data transmissions, different Original Equipment Manufacturers (OEMs) curate their own .dbc files, each catering to their needs. A vehicle can have multiple CAN buses, each connecting different ECUs to instigate communication between them. These files are maintained by different teams of engineers using tools like CANdb++ [1].

2.3 Connectivity and Automotive APIs

The emergence of connected technologies with 5G, edge computing, and vehicle-to-everything (V2X) architectures has transformed and is continuing to transform the automotive industry [60]. This led to vehicles from being seen as isolated mechanical systems to nodes with a broader purpose for digital ecosystems. This required revisiting and modernizing the traditional E/E architecture [58]. This movement has been largely driven by the consumer demand for seamless infotainment, real-time

[1] https://www.vector.com/se/en/products/products-a-z/software/candb/. Accessed: 04/04/2025

navigation, remote diagnostics, and over-the-air (OTA) software updates. As software and connectivity are becoming a key differentiator in automotive design, proactive manufacturers are seeking to integrate internet-based services, smartphones, and cloud platforms into vehicles [2]. To enable secure and interoperable interactions between in-vehicle systems, external devices, and cloud infrastructure, modern automotive architectures increasingly adopt standardized communication protocols, including REST APIs. Unlike legacy automotive protocols (e.g., CAN, LIN), which focus on low-level hardware communication, REST APIs operate at the application layer, abstracting complex vehicle functions into simple HTTP-based endpoints. This architecture allows remote access and control, third-party integration, and scalability as it enables automakers to update or expand services without overhauling entire vehicle architectures.

2.4 APIs

An application programming interface is a set of defined rules and specifications that software programs can follow to interact with each other, share data, and perform operations efficiently. APIs enable developers and clients to use external functionalities or data without the need to understand the underlying implementation details. They essentially provide a standardized way for different software components to interact, which promotes modularity, reusability, and interoperability. Especially with the advent of cloud computing, APIs have become fundamental in modern software development as their philosophy facilitates the creation of complex, scalable, and maintainable systems by integrating diverse services [18].

2.4.1 API-first design

API-first design is a software development approach that prioritizes the design of APIs before the development of the applications that will consume them. This approach treats APIs as first-class citizens by putting an emphasis on their importance as a contract between different software components [15]. The API-first approach involves defining the API's structure, functionality and behavior early in the development lifecycle. This is often done using API description languages such as OpenAPI [2]. Creating an API definition early on serves as a contract. In the traditional sense, this not only provides clearer communication between frontend and backend development teams but also with other stakeholders. With a well-defined API contract, frontend and backend teams can work in parallel, which reduces development time significantly [6]. Frontend developers can use mock APIs to start building the user interface, while backend developers implement the actual API logic. However, API-first design comes with its challenges. Most notable candidates include the need for rigorous API governance, documentation consistency and the risk of over-engineering.

[2]https://www.openapis.org/. Accessed: 28/03/2025

2.4.2 OpenAPI Specifications

An OpenAPI specification is a standardized description language for documenting RESTful APIs. For API-first design, the OAS is curated either as YAML ("YAML Ain't Markup Language" or "Yet Another Markup Language") or JSON (JavaScript Object Notation) representations. An OAS usually contains information about paths (endpoints, methods), components (schema), versioning, HTTP status codes etc. These standardized specifications of REST APIs are human-readable, making it easy to understand the services without actually looking at the actual API implementation. The generalizability of the OAS also allows developers to create and use tools to generate server and client code stubs, which are code structures of the API implementation for the API developers to build upon.

2.4.3 RESTful APIs

Representational State Transfer is one of the most widely adopted architectural styles for designing networked applications. RESTful APIs are defined as APIs that adhere to the REST principles. These types of APIs have become the standard for web services due to their simplicity, scalability, and stateless nature [18]. The key characteristics of REST include:

- **Client-Server Architecture** - promotes separation of concerns as client and server can evolve independently
- **Statelessness** - No information is stored on the server; each request from the client must contain all the information necessary to understand the request
- **Cacheability** - The client can cache responses from the server to improve performance and reduce server-side load
- **Layered System** - The server can be composed of multiple layers, but the client only communicates through a single endpoint and is not made aware of the complexity
- **Uniform Interface** - Each resource is uniquely identified, manipulations happen through uniform representations, messages contain enough information to describe how to process, and available actions are discoverable through links in the responses

2.4.4 REST APIs in vehicles

The automotive industry is undergoing a significant transformation with the increasing demand for connectivity and advanced infotainment. Considering the adaptability, ease of use, and their widespread applications, RESTful APIs are playing a crucial role in this transformation by enabling complex functionalities and services in modern vehicles. REST APIs in vehicles are used for a wide range of applications: telematics to allow location tracking, remote diagnostics for predictive maintenance, over-the-air software updates, infotainment to enable third-party

application integration and interactions with the internal system of the vehicle and vehicle-to-everything (V2X) communication. The use of REST APIs gives vehicle manufacturers an easier way to integrate with the already-existing broader digital ecosystem. Despite their advantages, just like anywhere else, RESTful APIs in vehicles face challenges such as cybersecurity risks, latency issues, and data privacy concerns [80].

2.5 Large Language Models

In recent years, large language models represent the most significant advancement in Natural Language Processing (NLP), and they are usually what the average person refers to as AI. They leverage the power of deep learning to process and generate human-like text, and in some cases, images, videos, and audio. These models are typically based on the transformer architecture, which relies on self-attention mechanisms to capture long-range dependencies in text efficiently [85]. LLMs are typically pre-trained on vast corpora of text data, enabling them to learn linguistic patterns, syntactic structures, and semantic relationships. The pre-training phase is often followed by fine-tuning on specific tasks, such as text generation, classification, or question answering, and this makes them highly versatile tools in AI research and application. Applications of these models range from conversational agents and automated content generation to code synthesis and scientific discovery, aligning with the goals of using library learning techniques to synthesize API generation as explored in this thesis.

The scale of LLMs has grown dramatically in recent years, with models featuring billions of parameters, such as GPT-3 (175 billion parameters) [10] and later iterations like GPT-4 [1]. These enormous models push the boundaries of language understanding and generation. The increase in size alongside improvements in training techniques like unsupervised learning and reinforcement learning from human feedback (RLHF) [67], has enabled LLMs to perform complex tasks with minimal task-specific training.

A key characteristic of LLMs is their ability to generalize across domains. This is largely driven by their exposure to all sorts of datasets during pre-training. Techniques such as prompt engineering — crafting specific inputs to guide model behavior — have emerged as the most used methods for taking advantage of LLMs effectively without extensive retraining [56]. This ability to adapt makes LLMs particularly relevant for various tasks, including but not limited to writing, translation, sentiment analysis, classification, and code generation, where the model must infer structured outputs from unstructured or semi-structured inputs.

2.5.1 LLM Parameters

LLMs generate text based on the probability distribution of the following tokens. How the LLMs generate these texts can be tuned by adjusting certain output parameters. "Maximum tokens" and "temperature" are two such parameters.

- **Maximum tokens** - An LLM-generated response is measured in tokens. A token can be an individual character, part of a word, or a complete word [75]. The tokenization of inputs and outputs is usually model-specific. The "max tokens" parameter enables adjusting the maximum tokens that the LLM can generate based on the specific need. The maximum tokens parameter also contributes to the model's context window. It is vital that the sum of input tokens to the LLM, i.e, the input prompt tokens, and the output tokens do not exceed the model's context length. If this length is exceeded, the LLM will fail to produce the generation.

- **Temperature** - The "temperature" parameter controls the randomness of the LLM in choosing the next token based on its probability distribution. If the temperature is set to a low value (0-0.5), the LLM will tend towards selecting the highly probable next token. In the case of setting a higher temperature, the likelihood of the model selecting a token of low probability is increased. This results in the LLM providing a diverse variation of text at the risk of being nonsensical while increasing the possibility of hallucinations.

2.5.2 Prompting

Prompting is the process of providing specific inputs (prompts) to language models to guide their output. This method allows users to leverage pre-trained LMs for various tasks without the need for additional training or fine-tuning, i.e., without altering the underlying model weights. It's a really flexible and efficient way to interact with these models, which, considering LLMs' ability to handle a wide range of tasks, makes them adaptable to various applications. Early language models relied on fine-tuning for task-specific performance, but the introduction of models like GPT-3 and subsequent research demonstrated that well-designed prompts could achieve similar results with little to no effort [10].

Research shows that effective prompting often involves different strategies such as *zero-shot*, *one-shot*, *few-shot*, *chain-of-thought*, *chain-of-code*, and/or *self-consistency prompting* [56, 49]. Zero-shot prompting describes the task without examples, relying entirely on the models' pre-trained knowledge. On the other hand, one- and few-shot prompts provide one or many input-output examples to guide the model in producing the desired output. The chain-of-thought (CoT) prompting technique is used to encourage the model to reason step-by-step to complete more complex tasks accurately [91]. In self-consistency prompting (often paired with CoT), multiple responses are generated and the model is free to select the most consistent one [89].

However, prompt engineering remains an empirical art as the outcomes are often sensitive to phrasing and model architecture [7]. Challenges include inconsistency across models and the need for domain expertise to design optimal prompts, especially for specialized applications like code synthesis. However, this seems to be becoming less of an issue as state-of-the-art LLMs evolve.

In the context of our thesis, specialized prompting techniques could be adopted to

guide the LLM in understanding and translating API documentation into specific code implementations. For instance, a prompt might specify the programming language, the frameworks to be used, the structure of the API, any particular functions or methods to include, and how to handle edge cases or errors.

2.6 The LLMs at hand

These are all the LLMs provided to us by Volvo Group for this thesis.

2.6.1 GPT-4o

GPT-4o, released by OpenAI in May 2024, represents a significant advancement in LLM technology [37]. It is a multimodal and multilingual model, meaning that it is capable of processing and generating content in text, images, and audio in various languages. Key features include faster response times compared to its predecessor, GPT-4 Turbo, and improved performance across voice, multilingual, and vision benchmarks. At the time of release, the performance of both GPT-4o and GPT-4 significantly surpassed that of then-available open source models such as Llama-2 [83] and Mixtral 8x7b [38].

GPT-4o and GPT-3.5 were the only available closed-source LLMs at Volvo Trucks at the time of writing this thesis. Eventually, we decided to use GPT-4o only as it beats its predecessor, GPT-3.5, across all benchmarks, including code generation related ones. As a result, we naturally expected GPT-3.5 to provide inferior results compared to GPT-4o, and by not using GPT-3.5, we made space for more open-source model benchmarking in our experiments.

2.6.2 Llama Models

The Llama family, developed by Meta AI, includes models like Llama 3.3 70B and Llama 3.1 405B, each with distinct characteristics for language processing tasks [82, 22]. Llama 3.3 70B, with 70 billion parameters, is the successor of Llama 3.1 70B and is designed for better efficiency, performing well on standard hardware. It is suitable for a range of complex language tasks and is definitely more practical for general use than the other. The open-source nature of the Llama model family promotes collaboration, allowing researchers and developers to modify and distribute the model.

2.6.3 Qwen 2.5 Coder 32B

Qwen 2.5 Coder 32B, developed by Alibaba, is a 32-billion-parameter model specialized for coding tasks [36]. It is based on the Qwen2.5 architecture [94] but optimized for understanding and generating code in various programming languages. This makes it ideal for code generation tasks. Its balance of performance and manageability suits practical coding applications. This specialization aligns closely with our thesis's focus, so we have decided to include this code-specialized open-source

model for our benchmarks. This is mainly done to see the performance difference between the generic foundation and code fine-tuned models when it comes to code generation for our case.

2.6.4 DeepSeek R1

DeepSeek R1 is an open-source reasoning model by DeepSeek that performs exceptionally well in tasks requiring logical inference and mathematical problem-solving [26]. Released in January 2025, it has quickly gained attention for its ability to provide transparent reasoning processes. What makes this feature crucial is the fact that users of the model are able to follow its logic step-by-step. Built upon the DeepSeek-V3-Base model with 671 billion parameters [55], to balance performance and computational efficiency, DeepSeek R1 also offers distilled versions with 32 billion and 70 billion parameters based on Qwen2.5 and Llama 3.1, respectively.

Unlike base models like GPT-4o, which excel in general language tasks, DeepSeek R1 is specifically designed for reasoning tasks. Key features of DeepSeek R1 include its hybrid training approach, which combines reinforcement learning with supervised fine-tuning to enhance readability and coherence. DeepSeek R1's strong performance in reasoning and mathematical tasks makes it a valuable candidate for code generation applications and even more valuable for judging other LLM outputs based on specific criteria. Its open-source nature also allows for greater flexibility and customization, which aligns with the thesis's focus on exploring diverse LLM capabilities.

Model	**Parameters**	**Context Length**	**Source**
GPT-4o	-	128,000	Closed [3]
Llama 3.3	70B	128,000	Open [4]
Qwen 2.5 Coder	32B	128,000	Open [5]
DeepSeek R1	671B	128,000	Open [6]
DeepSeek R1-Distill-Llama	70B	128,000	Open [7]

Table 2.1: Language model parameters, context length, and source comparison

2.7 Retrieval-Augmented Generation

Even though LLMs are great at generating text, they have limitations. Some of these limitations include computational constraints, hallucinations, lack of long-term memory, and bias [29, 35]. Current state-of-the-art LLMs produce highly coherent texts during inference, but that does not mean their answers are always factually true. What's more, their knowledge base is cut at a certain point, and it usually

[3]https://platform.openai.com/docs/models/gpt-4o. Accessed: 28-03-2025
[4]https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct. Accessed: 28-03-2025
[5]https://huggingface.co/Qwen/Qwen2.5-Coder-32B. Accessed: 28-03-2025
[6]https://huggingface.co/deepseek-ai/DeepSeek-R1. Accessed: 28-03-2025
[7]https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-70B. Accessed: 28-03-2025

does not change over time unless they are continuously trained, but that usually requires allocating immense computational resources. Retrieval-Augmented Generation (RAG) is a hybrid approach that enhances LLMs by combining generative capabilities with external data (knowledge) retrieval. RAG addresses the limitations of standalone LLMs, such as factual inaccuracies or "hallucinations," by retrieving relevant documents from an external corpus to inform the generation process [48]. This is achieved through a two-step mechanism: a retriever fetches contextually relevant information, and a generator (LLM) produces an output conditioned on both the input prompt and retrieved data [20]. For API generation via library learning, RAG can be used to retrieve domain-specific data such as documentation, code examples, or specifications. By doing this, the model should be able to synthesize more accurate and context-specific APIs without relying much on memorized training data.

RAG's strength lies in its ability to ground LLM outputs in verifiable knowledge, and this is why it is particularly useful for knowledge-intensive tasks. Unlike traditional LLMs that depend only on their pre-trained weights, RAG enables dynamic access to external sources to improve factual consistency and adaptability [48, 28, 20]. In the context of this thesis, RAG could enhance API generation by retrieving relevant function signatures and documentation from the signal library, allowing the model to map API properties to CAN signals correctly. Nevertheless, RAG also has its challenges, including the computational cost of retrieval, the quality of the knowledge base, and the integration latency, which can hinder real-time applications [8].

2.7.1 Embedding Models

Embedding models are the backbone of RAG's retrieval phase as they convert text into dense vector representations that capture semantic meaning. These models, often based on architectures like BERT or its variants, encode queries and documents into a shared vector space where similarity can be measured (e.g., via cosine similarity, Euclidean distance, dot product) [13]. The Dense Passage Retrieval (DPR) model is an example of this approach, using dual encoders — one for queries and one for passages — to optimize retrieval accuracy in open-domain settings [45]. For API generation, embedding models can encode CAN signal documentation and code snippets. This way, the retriever could identify relevant CAN signals and their functions based on a prompt like "Generate a function for the API property X that does Y".

The choice of embedding model significantly impacts RAG performance. Pre-trained models like Sentence-BERT improve efficiency by producing embeddings tailored for sentence-level semantics [70]. At Volvo Group, we have access to two open-source embedding models - Instructor-XL and BGE-M3 [79, 61], and two closed-source ones developed by OpenAI - text-embedding-ada-002 and text-embedding-3-large [65, 66]. In the case of the thesis, embedding models must balance semantic accuracy with the ability to handle technical jargon and code syntax, making them a critical area of study for optimizing RAG in API synthesis.

Embedding Model	Dimensions	Parameters
Instructor XL	768	1.5B
BGE-M3	1024	569M
text-embedding-ada-002	1536	-
text-embedding-3-large	3072	-

Table 2.2: Embedding models' supported dimensions and parameters comparison

2.7.2 Reranker Models

Even though retrieval techniques in RAG systems are often designed to be efficient and computationally scalable, they may not always rank the most contextually relevant items highest in the list. To address this limitation, rerankers are often used as a subsequent step after retrieval to further refine the ordering of retrieved candidates with the goal of "pushing" the most appropriate items to the top of the list for the generative model [63].

Rerankers often leverage advanced machine learning models, such as cross-encoders. Bi-encoders, which are commonly used in initial retrieval, independently encode the query and the documents into embeddings of fixed length. This helps achieve fast similarity comparisons. On the other hand, cross-encoders process the query and each candidate document together as a single input. This could help capture more intricate semantic interactions between their tokens. Joint processing typically produces more accurate relevance scores, but this comes at a drastic increase in computational complexity [62].

The retrieve-and-rerank approach is widely used in RAG systems, and compound AI systems are no exception [73]. To help with the problem explored in this thesis, we plan to employ a similar approach to the agent subsystem, leveraging RAG. We leverage a BERT-based model for reranking - TinyBERT [40] - as this was the model provided by Volvo Group.

2.7.3 Vector Databases

Vector databases complement embedding models by storing and indexing the dense vectors for efficient retrieval. Contrary to traditional keyword-based databases, vector databases like FAISS (Facebook AI Similarity Search) or HNSW (Hierarchical Navigable Small World) use approximate nearest neighbor (ANN) search to identify documents closest to a query vector quickly [43, 14, 59]. This scalability is usually crucial for RAG as it enables "real-time" access to extensive information corpora, such as millions of documents, without exhaustive and slow linear searches.

The effectiveness of vector databases is associated with their indexing strategy and update mechanisms. For instance, FAISS supports GPU acceleration and clustering to handle high-dimensional vectors efficiently [43], whereas HNSW offers tradeoffs between speed and accuracy suitable for more dynamic datasets [59]. The chal-

lenges of vector databases include maintaining up-to-date embeddings as libraries evolve and managing memory for large-scale deployments [47]. In the context of this thesis, vector databases enable RAG to search for candidate solutions for program synthesis, ensuring that retrieved knowledge remains current and semantically relevant for generating robust and functionally correct APIs. This thesis utilizes a Volvo-provided instance of ChromaDB [8], an open-source vector database, to store our embeddings. This database uses the L squared norm (L2) metric to measure similarity.

2.8 LLM Agents

Following LLMs' success and building upon their foundational capabilities, a new class of AI systems known as "LLM Agents" has emerged in the past few years. Unlike standalone LLMs that primarily function as sophisticated text generators based on a single prompt, LLM agents are capable of autonomous decision-making and the execution of complex actions to meet specific goals by integrating tools like web/file search, code interpreters, and more [93].

LLMs are stateless, meaning that during inference, they cannot logically connect one prompt to another and learn from it. Recent studies regarding frameworks and architectures for creating language agents, such as ReAct and CoALA, show the importance of agents featuring different types of memory, like short-term and long-term memory, to tackle this shortcoming of LLMs [95, 81]. As shown in Figure 2.2, the architecture of an LLM Agent typically includes four components: the LLM itself, a set of external tools, a memory module, and a planning mechanism. The LLM processes natural language inputs and generates decisions, while tools extend its capabilities by providing access to relevant data or computational resources. When it comes to an agent's memory, it is often implemented as a context window or vector store and used to maintain task history and state. This is particularly important for long-running and data-intensive tasks. The planning mechanism, such as CoT prompting, breaks complex problems into sub-tasks to guide the agent through iterative cycles of reasoning, action, and observation [95, 27].

LLM-based AI agents have been explored and are being explored in various applications, such as virtual assistants and autonomous systems in simulated environments [86]. Some combine the code-generating capabilities of LLMs with the independence and tool-using skills of agents [74]. This combination allows the creation of systems that can repeatedly improve and check generated code until it matches the required standards [95, 19]. For example, an agent could take an instruction in natural language from the user: "Create a RESTful API implementation for serving weather data.". The agent would ideally use the LLM to draft initial code, test it with appropriate tools, and refine it based on the results and/or some other feedback [57]. By employing this approach, one could reduce the manual work needed in development and help developers build products more quickly.

[8]https://www.trychroma.com/. Accessed: 16-05-2025

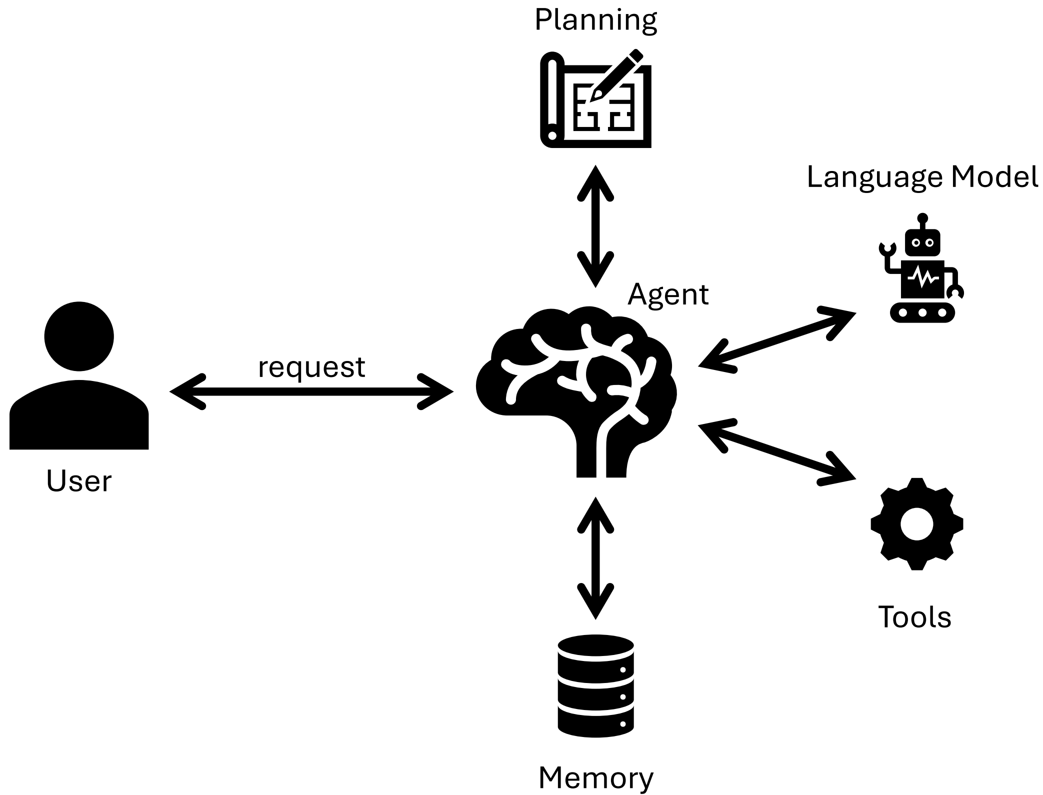

Figure 2.2: A high-level overview of a simple LLM agent

2.9 LLMs as Judge

"LLM as Judge" is a paradigm under which LLMs are involved in assessing artifacts. Traditional evaluation methods, such as static metrics like BLEU, often fall short in capturing the correctness, functionality, and creativity of code [51]. Human-based evaluations are considered thorough but time-consuming and costly, making them quite impractical for assessments [52]. To address this, LLMs could be used as automated evaluators by leveraging their natural language understanding and contextual reasoning capabilities to assess generated artifacts. When it comes to software engineering, LLMs can be invoked to assess code, specifications, and requirements.

One key approach in employing LLMs as judges for code generation is direct scoring. In this approach, language models assign numerical ratings to code outputs based on predefined criteria like correctness, readability, and efficiency. As an example, the CodeUltraFeedback dataset uses LLMs as judges to rank code responses based on coding preferences while offering detailed textual feedback and numerical scores [92]. Similarly, the CodeJudge-Eval benchmark assesses LLMs' code understanding abilities from a code-judging perspective by attempting to address the limitations of traditional code generation benchmarks [99]. These evaluations' aim is to go beyond assessing functional correctness and to include evaluations of code quality, efficiency, and adherence to coding standards. This form of evaluation is capable of a more comprehensive assessment of generated code, making sure that the output adheres to the user-defined coding preferences and software engineering best practices [87].

Extending the LLM-as-a-judge concept, the agent-as-a-judge framework has been proposed to evaluate agentic systems by utilizing agentic systems themselves [100]. This approach introduces intermediate feedback mechanisms throughout the task-solving process. It attempts to address the limitations of traditional evaluation methods that often focus solely on final outcomes. Applied to code generation tasks, having a judge agent has shown promising evaluation capabilities and results, claiming to outperform previous methodologies and providing more reliable assessments of agentic systems' performance [100].

Despite showing great potential, the LLM-as-a-judge paradigm is not without challenges. One significant concern is the potential for 'preference leakage,' where the judge shows bias towards LLMs with which it has a close relationship, such as being the same model or from the same model family [50]. This bias could potentially compromise the objectivity and the correctness of the evaluation. Additionally, LLM-generated evaluations could also require careful validation, which might require mixed-initiative approaches that combine LLM judges with slight human supervision [76]. Ensuring the reliability and fairness of LLM-as-judge evaluations remains an area of ongoing research. Efforts seem to focus on developing more transparent evaluation processes and mitigating potential biases.

2.10 Compound AI Systems

Compound AI systems are systems that are used to tackle tasks with higher complexities by combining multiple components [96]. These systems are modular and can include various AI components. While agents use LLMs in a monolithic fashion, Compound AI Systems use LLMs and other tools like frameworks for information retrieval, system calls to external services, etc. All these components work in unison to achieve the objective for which this system was created, while overcoming the limitations of a monolithic AI Agent. These components range from ML models, LLMs, Reinforcement Learning (RL) models to information retrieval mechanisms like RAG, web search, specialized knowledge bases, and even other agents. This allows for the creation of such systems tailored for each need [96, 72].

While agents are systems where the potent LLMs are applied to automate tasks to achieve certain objectives, compound AI systems orchestrate interactions between multiple components, which can sometimes be multiple agent systems as well. Instead of having one generic multi-purpose AI agent, multi-agent systems, where multiple autonomous agents collaborate to achieve a common goal, are becoming more and more popular [27]. These systems perform diverse tasks in a coordinated fashion, such as automating software engineering workflows including code generation, testing, and documentation [41]. In the domain of software engineering, particularly for API implementations, multi-agent systems can be employed to increase efficiency and performance by distributing responsibilities among highly specialized agents. For example, one agent could be tasked with understanding the OpenAPI specifications provided, another with generating the corresponding API code, and a third with validating and refining the output based on feedback from a human

or execution logs and results [41]. Nevertheless, the deployment of multi-agent systems faces several challenges. These include managing the inherent complexity of coordination between agentic interactions, ensuring that all agents work towards a unified goal, and high coupling between agents' I/O [27, 41]. Despite these obstacles, multi-agent systems hold significant potential for advancing automation in software development processes.

2.11 Automatic Programming

Automatic programming is an ever-evolving programming paradigm in which some underlying mechanism generates a computer program. This concept has been relevant ever since the inception of AI. This paradigm synthesizes a program from a specification that belongs to a higher level of abstraction and is easy for humans to specify. For automatic programming to be effective, the specification must be concise when compared to a computer program written in a conventional programming language. Writing or rewriting a computer program involves the programmer being knowledgeable in concepts such as algorithms, data structures, and identifying design patterns in programs [44].

Automatic programming involves the use of AI for:

- Representing, finding, and instantiating design patterns in computer programs.
- A search technique to find the combination of different program components to synthesize a program that accomplishes the task [44].

While automatic programming refers to the automation of code creation in the broader sense, there are other interconnected programming paradigms that are specializations of automatic programming, such as inductive programming and program synthesis, which follow the underlying principles of automatic programming, that is, automatically generating code from some specification.

2.11.1 Program Synthesis

Program synthesis is a programming paradigm that follows the underlying principle of automatic programming: the ability to derive code from some formal specification. However, program synthesis emphasizes differences from other programming paradigms, like inductive programming, so that the program synthesized satisfies the user's intent while being consistent with the specification. This differs from program verification, where an existing program is verified against expected behavior. Whereas, in the case of program synthesis, a program is correctly created based on the specification. Also, program synthesis requires that the specifications be more formal and complete. Program synthesis utilizes deductive reasoning and formal methods for synthesis, where, based on the specification, the synthesizer performs a search through a program space to compose a program that fulfills the user intent [23, 46].

2.11.2 Inductive Programming

Inductive programming is yet another paradigm that follows the underlying principles of automatic programming. For program synthesis, the specification needs to be formal and complete, and the programs are synthesized using deductive reasoning. Inductive programming, on the other hand, can use specifications that can be incomplete and informal, like input/output examples and constraints. Then, based on the input specification, a search is done within a program space. Subsequently, the candidate program is synthesized based on evolutionary or systematic search methods and verified against the input-output examples using the generate and test approach [25].

2.11.3 Key Dimensions of a Synthesizer

In order to automatically synthesize a computer program, the system that synthesizes the program, i.e, the synthesizer, is characterized by these three main dimensions [23]:

- ***User Intent* - The specification:** The intent for which the program is synthesized should be described to the synthesizer in the form of a specification. Based on the paradigm, the specification can be in natural language, input-output examples, formal and complete specifications like formulae, traces, and higher-order specifications in the form of partial programs [24].

- ***Search Space* - The knowledge base:** The search space is the space that contains the candidate programs, over which the desired program will be selected for synthesis. There also arises the need for the search space to be optimized for expressiveness and efficiency. If the space is not expressive enough, the synthesizer might fail to include programs in the synthesis that might be relevant to fulfill the user's intent. But, the space should also be restrictive enough for the synthesizer to be able to perform an efficient search over a domain of programs that are compliant with efficient reasoning in synthesizing a program to fulfill the intent [24]. A space with programs that fall under the domain of the user's intent plays a pivotal role in the automatic programming process. Automatic programming systems that lack such a knowledge base make the system non-feasible in solving problems in a non-trivial domain [5].

- ***Search Technique*:** The technique used to synthesize the relevant programs is a key in any automatic programming paradigm. These techniques depend on the specification and also the search space over which the search is performed. These techniques are ranged across many methods like search logic deduction, induction, constraint solving, and ML techniques like probabilistic inference and genetic programming [24].

2.11.4 Program Synthesis with ML Models

Recently, applications of Inductive Program Synthesis (IPS) are on the rise, due to the recent insurgence in ML-based search techniques like symbolic and neuro-symbolic search [3]. Some of these applications synthesize programs based on different specifications to solve problems in domains like regex, math, list functions, and string formatting [17, 4, 64, 16]. The knowledge base used by most of these implementations to synthesize programs is either composed of Domain Specific Languages (DSL) [23] or languages that are curated with the scope of only being program synthesis [3]. Program synthesis using ML models with general-purpose programming languages remains an avenue for further research.

2.11.5 Program Synthesis with LLMs

LLMs have shown their ability to generate texts in multiple forms, including code [33, 3], as they are trained on many code datasets. When it comes to using LLMs to synthesize code using a paradigm such as inductive program synthesis, LILO uses LLMs to name functions or libraries that the system learns while the system does a neural-guided search over a search space of DSL programs [21, 17]. LLMs are also being used in assisting program synthesis, acting as a tool for verification to conduct a more efficient search for syntax-guided synthesis [54]. While program synthesis paradigms that don't involve an LLM have a need for formal and complete specifications, those that do can be flexible in this matter. LLMs have shown potential in synthesizing programs across various complexities, from synthesizing functions that carry out a single operation to solving complex math problems [3]. The potential of LLMs in generating code can also be used to create a search space full of programs based on a specification. Moreover, reasoning capabilities could potentially be used to choose the correct programs from the search space that fulfill the user's intent. The use of LLMs in such cases remains largely unexplored.

3

Related Work

3.1 Library Learning

Programmers use function abstraction as a means for hiding the complexity of a program. This is an efficient way of programming as it dismisses the need to reintroduce the underlying primitives (fundamental building blocks of a program) that were abstracted into a function. Library learning is a concept that experiments on building high-level functional abstractions, complementing inductive program synthesis by automatically discovering reusable functional components using predefined low-level primitives. This enables the code generation process for solving problems to be more efficient and abstract [9]. For instance, solving simple math problems using library learning may entail curating the operators, such as addition and multiplication (primitives), needed to solve the given problems. Then, based on input-output examples, an ML model will do a search to find the operator that solves the problem. As the problems get more and more complex, library learning invokes techniques to come up with new solutions that are synthesized using solutions that were used to solve programs of lower complexity.

3.1.1 Library Learning Complements Program Synthesis

The use of inductive programming and ML techniques for program synthesis has been relevant since their inception [23], even before the LLM “boom”. Among the notable advancements, library learning and inductive program synthesis have emerged as powerful tools for abstracting and automating complex problem-solving tasks [16, 17, 21]. For example, ”DreamCoder” demonstrates the capability to bootstrap inductive program synthesis using a ”wake-sleep” library learning approach, effectively enabling systems to refine and expand their libraries of reusable programs over time [17]. This novel iterative approach, which is largely based on the foundational ”wake-sleep” algorithm [32], shows the potential for intelligent systems to improve their generative capabilities through structured feedback and optimization. Dreamcoder employs the "wake-sleep" cycle. During the "wake" phase, it synthesizes solutions, while during the "sleep" phase, it recognizes the kinds of programs that produce certain behaviors and creates a high-level library. In a quite similar fashion, ”DeepCoder” combines neural networks with symbolic reasoning to generate small programs from input-output examples [4].

Furthermore, systematic program synthesis techniques, such as sketching [78], emphasize the importance of creating reusable and comprehensible artifacts. More recently, LILO which is built on top of Dreamcoder, introduces methods for learning interpretable code libraries through compression and automated documentation using LLMs [21]. When LILO learns a new code library, unlike Dreamcoder, it names the library based on its functionality. Therefore, this helps restrict the search space, when the model tries to solve a similar problem. All these approaches to library learning rely upon the primary objective of inductive programming synthesis to solve problems across multiple domains, and ultimately follows the perennial dream in Artificial Intelligence, which is to build a machine that learns like a human does, starting from much less and progressively learn towards solving complex problems [17, 84]. But these implementations of library-learned models solve domain-specific problems using DSL.

3.2 LLMs for Programming in Software Engineering

Most software engineering workflows that make use of traditional ML models are typically designed to solve specific tasks, while being data-hungry during training [17, 21]. Implementations like LILO and Dreamcoder rely on large amounts of input-output examples and curated tasks for the models to be trained on. Even software engineering workflows that use LLMs mostly use these models for the purposes of code generation when compared to program synthesis. Program synthesis is a paradigm where a high-level program is constructed from the ground up using abstract low-level fundamental specifications. Synthesizing a program involves defining the program space, which contains all the possible candidate programs, identifying the intent for which the program is generated from formal specifications, and performing the search in the program space to synthesize the solution based on the given intent [23, 25]. Whereas, code generation takes higher-level representations and converts them into target code without necessarily deriving its functionality from scratch [33]. Most LLM-based agents like QualityFlow [34] and the applications of LLMs in generating programs to solve problems [3] rely upon code generation rather than program synthesis. Systems using RAG to give context to the LLM while writing code could be limited by their sub-optimal performance in finding useful information that helps the synthesis process during retrieval [53, 90]. This highlights the constraints that a pipeline may face in synthesizing code from retrievals. Also, LLM-infused software systems that address all three dimensions of program synthesis remain an avenue for future research.

3.3 LLM-based Systems in API Development

When it comes to automating the generation of microservices implementations using LLMs, there are applications where LLMs are used to convert OpenAPI Specifications into REST API implementations. A related paper presents a CRUD microservice generator implementation which translates a natural language description into an OpenAPI specification [11]. Then, this specification is given to another LLM,

which produces the server code. LLMs are also invoked to fix faults in the generated specifications or code if they do not meet the user's expectations. Moreover, LLMs have shown potential in automating the testing process of RESTful APIs at Volvo [88]. These applications show the ability of LLMs to work with OpenAPI specifications to create and test REST API implementations.

3.4 Gaps in Research

3.4.1 Program Synthesis in Software Workflow Automation

Despite all these notable advancements, the specific application of these methodologies when it comes to the automation of generating vehicular REST APIs, or of any other software workflow, using program synthesis at this scale, particularly in the context of CAN signal specifications abstracted as functional interfaces, remains unexplored.

3.4.2 Enhancing AI-powered Program Synthesis

Even though the program synthesis paradigm has many applications, the contribution of modern advancements in AI tools and techniques in addressing the three dimensions of programming is still at the dawning stage on a horizon of countless possibilities. The search space for program synthesis, to be expressive yet restrictive, has always limited the space to be populated with domain-specific languages. Also, the search techniques for finding the best candidate programs have mostly been traditional search techniques like brute-force or machine learning techniques like probabilistic inference and genetic programming [24]. The use of powerful embedding models and vector databases to construct an effective search space, vector search to perform an efficient search, and the code generation capabilities of LLMs in populating the search space and as a synthesizer remains to be seen.

3.5 Addressing the Gaps

This thesis aims to address these research gaps and push the boundaries of current research while contributing a novel application to the growing field of intelligent software process automation.

LLMs endeavor in code generation, reasoning, and semantic understanding capabilities, even in the context of OpenAPI Specifications, compared to traditional machine learning models. So, incorporating these potent LLMs in a workflow that employs library learning and program synthesis characteristics might make building a system that can learn abstract low-level concepts possible. Through synthesis across different levels with its evolving library of previously learned lower-level concepts, the system can synthesize programs to solve problems at a higher level, using the lower-level programs complementing library learning and program synthesis, where each high-level component of the API will be built upon small, fundamental, reusable

low-level system components. This approach aligns with this study's objective of generating high-level artifacts (APIs) from low-level CAN signals.

4

Methods

4.1 Research Methodology

This thesis follows the design-science research (DSR) methodology. The design-science paradigm is built upon the goal to extend human and organizational capabilities by creating new and innovative artifacts. Design-science promotes developing theories or building artifacts based on business needs using applicable knowledge from existing research. Then the developed theories or artifacts should be evaluated and justified against the business needs for which the artifact was built, on top of making valuable research contributions [31]. The design of an artifact plays a central role in design-science research. In the field of information systems or information technology, DSR necessitates the creation of a functional artifact for a specific problem and evaluation to ensure the utility of the artifact for that problem.

With regards to the creation of the artifact, design-science emphasizes the following guidelines [31].

- Design as an Artifact - It stresses that the research must produce an artifact that can be implemented and applied in an appropriate domain.
- Problem Relevance - The objective of the research is to develop a solution for a real business problem.
- Design Evaluation - The utility of the artifact must be demonstrated via well-defined evaluation methods.
- Research Contributions - Provides contribution to research in areas of design artifact, etc.
- Research Rigor - Appropriate and well-established methods should be applied in the design and evaluation of the artifacts.
- Design as a search process - Design of the artifact requires utilizing available means to reach desired ends.
- Communication of research - The research must be presented to both technol-

ogy and business-oriented audiences.

The problem that our proposed system tries to solve complements DSR's objective, which is to find solutions for practical problems through design. A central piece in DSR is the artifact; in our case, the artifact is the system that automates the SPAPI development process. Also, the approach that we take to solve the problem, i.e, by designing the system that synthesizes RESTful APIs, and then evaluating the utility of the system by testing the synthesized APIs, reflects the DSR guidelines for developing and assessing the created artifact.

4.2 Automating SPAPI Development

Several challenges arise in developing a system to automate such a complex workflow in an industrial setting. A key difficulty lies in determining when and how to integrate automation tools, such as LLMs and scripting techniques, into the development process.

This study proposes an approach that decomposes the conventional development workflow into distinct yet interdependent processes and builds a software system, SPAPI Coder, to emulate and enhance this workflow. By rethinking the manual process, the proposed system aims to improve efficiency and correctness while reducing the need for extensive manual coordination.

4.2.1 Decomposing the Manual SPAPI Development Process

The automation of SPAPI development relies on two complementary principles: inductive program synthesis and library learning. Enabled by LLMs, these approaches provide a framework for decomposing the complex task of generating API endpoints into more manageable subproblems and creating reusable components for efficiency. This section outlines how these principles guide the development of SPAPI Coder, designed to rethink the manual SPAPI development process by leveraging patterns extracted from its workflow.

4.2.1.1 The Guiding Premise for Process Decomposition

The interplay between library learning and inductive synthesis is central to our approach in building our proposed system, SPAPI Coder. Library learning identifies reusable primitives, while inductive synthesis uses these primitives to generate new endpoints that adhere to OAS specifications. Prompt engineering with LLMs facilitates this process and enables the discovery of components and the synthesis of code without extensive manual intervention. Together, these principles rethink the manual SPAPI development process, leveraging its structure to create a scalable, automated solution.

Inductive Program Synthesis - Even though most synthesizers and research sur-

rounding program synthesis are founded upon solving complex proofs and theorems using DSL [23], the advances in AI could push the boundaries of the definition of program synthesis, especially when the scale of the problem requires that it be divided into subproblems [97]. In the context of SPAPI development, the complexity of translating OpenAPI specifications and CAN signal mappings into functional API endpoints necessitates breaking the problem into smaller, manageable subproblems [68].

To address this, SPAPI Coder decomposes the SPAPI development process into three abstraction levels, inspired by the manual workflow but restructured for automation. This method of decomposition mirrors successful applications of task reduction in program synthesis [77, 12, 98], enabling SPAPI Coder to tackle the complexity of API development while ensuring functional correctness.

Library Learning - In the automated SPAPI development, SPAPI Coder employs characteristics of library learning by constructing a hierarchical library of reusable components at different abstraction levels, from low-level CAN signal interactions to high-level API property mappings and endpoint implementations. These components reduce the search space for synthesis by providing predefined building blocks that encapsulate common patterns in SPAPI development.

For SPAPI automation, a high-level library for interacting with CAN signals based on partial or vague signal descriptions, such as those found in CAN databases or proprietary documentation, acts as a primitive. For instance, a component in this library might encapsulate the logic for accessing a CAN signal like ***ACMode.On***, abstracting low-level CAN Signal details into a reusable function. This library is then used during inductive synthesis to map API properties to CAN signals efficiently. For example, when synthesizing the ***/cabinclimate*** endpoint, SPAPI Coder can draw on library primitives that handle climate-related CAN signals, reducing the need to generate such logic from scratch. By constraining the synthesis search space, the library enhances the efficiency of the search and correctness of generated endpoints.

4.2.2 Decomposed Levels of the Manual SPAPI Development Process

The manual implementation of SPAPI endpoints involves three interdependent processes. Each of these processes requires coordination among multiple teams, including the SPAPI team, control system teams, application teams, and testing teams. These processes are not strictly sequential but often iterative, with feedback loops to refine outputs based on stakeholder input or evolving requirements. The processes are:

- **Understanding CAN Signal Properties and API Documentation**: Engineers analyze API specifications alongside CAN signal documentation. This step establishes the foundation for mapping by identifying relevant signals and their properties.

- **Mapping API Properties to CAN Signals**: Based on client requirements in the OAS, engineers map API properties to corresponding CAN signals and their values. This process requires close collaboration with control system teams to ensure accurate and meaningful mappings, often involving iterative refinements or additions to align with application needs.

- **Implementing API Endpoints**: Using the mappings, engineers develop functional API endpoints that integrate with the vehicle's communication infrastructure. This step may necessitate revisiting earlier mappings or signal interpretations to address implementation challenges or specification updates.

These processes are tightly coupled. Inaccuracies in signal understanding can lead to incorrect mappings, which in turn affect endpoint functionality and requirements alignment.

4.2.3 The Ideal SPAPI Development Automation System

Automating the process of synthesizing high-level API endpoint implementations from CAN signals and API specifications raises the need for a system that emulates the manual workflow through three modular abstraction levels:

- **Level 3: Signal Abstraction** - During this part of the pipeline, the system processes CAN signal information from diverse data sources, such as CAN databases and proprietary documentation, and represents it in an abstract, machine-readable format. This abstraction enables the system to interpret signal properties systematically, facilitating their use in subsequent mapping tasks. This functional component acts as the interface that enables communication between the APIs and the CAN bus.

- **Level 2: Signal - API Property Mapping** - This level of the pipeline maps API properties defined in the OAS to their corresponding CAN signals and values, based on client requirements. By automating this process, the system minimizes manual coordination and iterative revisions. It aims to produce mappings that are consistent and aligned with the OAS.

- **Level 1: SPAPI Endpoint Implementation** - Using the mappings and the endpoint definitions, this module generates high-level API endpoint implementations. The system produces functional code that integrates with the vehicle's communication infrastructure.

Each level corresponds to a key process in SPAPI development, with subsequent levels building on the outputs of the previous ones. However, the system allows flexibility to initiate automation from any level if prior outputs (e.g., signal abstractions or mappings) are already available. The modular design of the system allows each level to operate autonomously while maintaining dependencies between levels to ensure coherence and easier per-module evaluation. For instance, accurate signal abstractions at Level 3 are critical for reliable mappings at Level 2, which in turn

inform robust endpoint implementations at Level 1. SPAPI Coder, as shown in Fig. 4.1, is the system that we propose to automate SPAPI development.

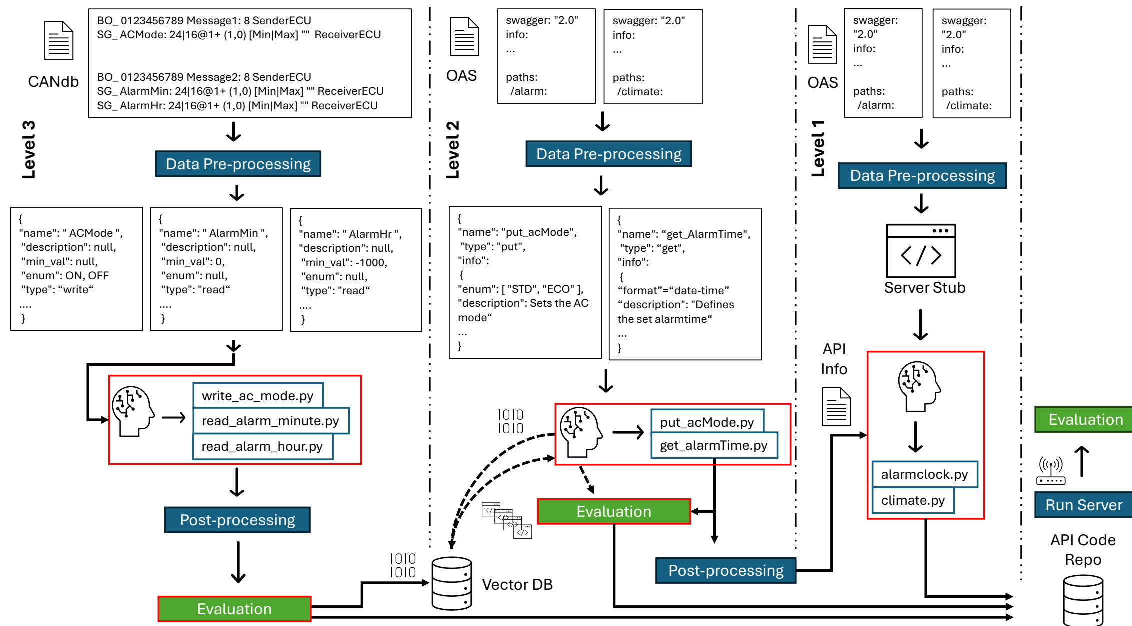

Figure 4.1: Architecture of the system that automates the SPAPI development and testing process. Components enclosed in **Red** rectangular highlights indicate that the particular system component invokes an LLM.

4.3 Data Sources

4.3.1 CANdb

The CAN database provides critical information about CAN signals used in SPAPI development. For this thesis, a CANdb containing approximately 3,500 CAN signals was utilized, covering signals relevant to vehicle communication and control. Information about these signals is used to create CAN signal abstractions at Level 3 of the SPAPI Coder pipeline.

4.3.2 OpenAPI Specifications

OpenAPI specifications in OpenAPI 2.0 YAML format serve as the primary input for defining SPAPI endpoints. These specifications, developed and maintained by the SPAPI teams at Volvo, outline the structure and requirements of REST APIs, including endpoints for methods such as GET, PUT, POST, and DELETE. For this thesis, we analyzed around 50 OASs containing information about around 170 endpoint implementations. These specifications provide the requirements for mapping API properties to CAN signals and generating endpoint implementations, forming the foundation for Levels 2 and 1 of the SPAPI Coder pipeline. A simplified and sanitized version of *CabinClimate* API can be found in Appendix A.1.

4.3.3 Proprietary Data Sources

In addition to CANdb, proprietary software documentation tools within Volvo provide supplementary information about CAN signals, ECUs, and vehicle-specific states. Although CANdb contains information about all the CAN signals regarding the transmission through the bus, these tools contain more descriptive information about the signals representing vehicle states, making it an important part for software development and maintenance of software systems and frameworks that utilize these components. This enriched information is vital for accurately interpreting CAN signals during the signal abstraction phase (Level 3) and ensuring meaningful mappings to API properties (Level 2). By integrating these proprietary sources, SPAPI Coder can better contextualize signal data, improving the robustness of the automation process.

4.3.4 Ground Truth Data for Testing and Evaluation

4.3.4.1 Signal-Property Mappings

Since the system that we build to develop API endpoints maps CAN signals to corresponding API Properties, it is crucial to evaluate the mappings made by the system against the actual signal mappings of human-generated endpoints. This information is procured from a proprietary data source in a tabular form, amounting to around 90 tables containing CAN signal to API property mappings. These tables serve as a reference to assess the accuracy of mappings generated by SPAPI Coder at Level 2, enabling quantitative evaluation of the system's ability to align API properties with corresponding CAN signals.

4.3.4.2 API Endpoint Test Scripts

Due to the fact that SPAPI Coder ultimately generates an API endpoint implementation, there is a need to test the APIs to verify their functional correctness according to the specifications. As there is already a system developed at Volvo that generates test scripts for the manually developed SPAPIs based on OAS [88], we utilize the test scripts generated by this system to test the SPAPI implementations generated by the automated development pipeline.

4.4 Level 3: Signal Abstraction

Since the lowest level of the SPAPI architecture involves the transmission and reception of signals (vehicle states) through the CAN bus, there arises the need for a component in the API implementation that can interact with the CAN bus. On top of that, in order to do the API property synthesis at Level 2, CAN signals should be presented to the system in such a way that it can capture the intent. This intent is mapping the API properties with their corresponding CAN signal(s). Hence, presenting each CAN signal as a functional component will not only help in building the lowest level of the SPAPI implementation, but also help in acting as

the program search space, using which the API property to signal mapping is done in the next level of the SPAPI Coder pipeline.

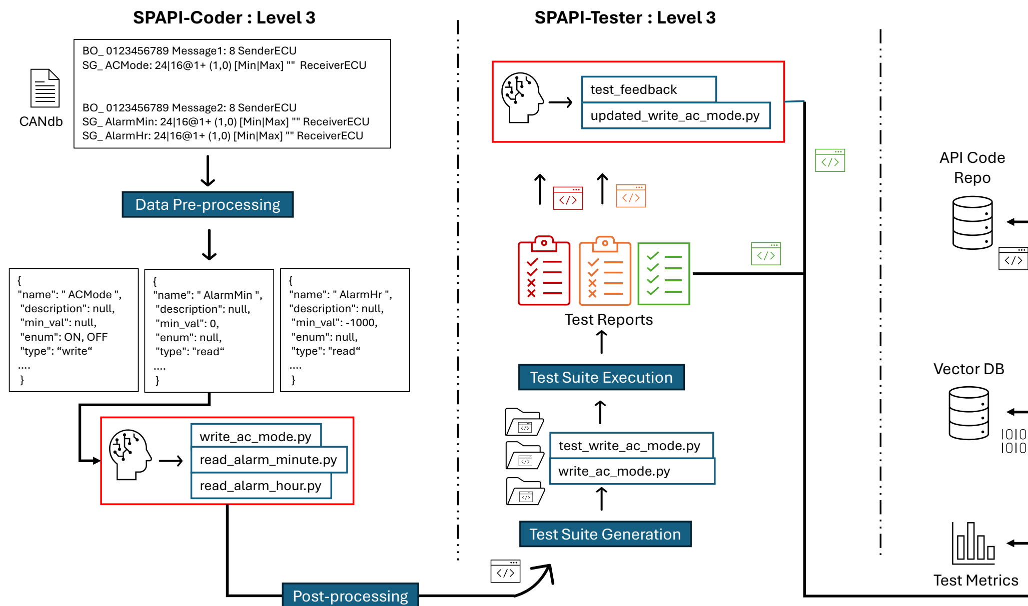

Figure 4.2: Architecture of the system at the CAN signal interface level. Components enclosed in **Red** rectangular highlights indicate that the particular system component invokes an LLM.

4.4.1 Data Preprocessing: CAN Signal Representation

As presenting the whole CANdb to the LLM is not an ideal way of abstracting CAN signal information to the system, we present each CAN signal to the system as a flat dictionary of each signal's properties. This is done so that it is easier for the model to interpret a known datatype, such as JSON, than to present the CANdb as plain text, as LLMs may fail to interpret the corresponding values and properties. Also, a service developed in Volvo is utilized to get the natural language description for signals, if available. This is done with the goal of providing more context to the language model. An example of the signal representation of *ACMode* as a JSON can be seen in Appendix A.2

4.4.2 SPAPI Coder at Level 3

Since the lowest level of the SPAPI architecture involves the transmission and reception of signals (vehicle states) through the CAN bus, SPAPI Coder generates a signal interface library which contains functions to read/write signals from/to the CAN bus. These functions either return a vehicle state or set a certain state. This is so that the high-level API endpoints can get information regarding a certain vehicle state or set a certain vehicle state utilizing these functions.

All the vehicle states in the CAN bus are represented only as integer, float, or enum data types, and are structured in the CANdb. Considering this, there arises the

question whether one could render these functions from a common program template instead of employing an LLM to do so. In the manual SPAPI process, engineers map these CAN signals to API properties. This involves the engineers' understanding of both the CAN signal and the API property. Since we are automating this process using an LLM, it is necessary to generate the signal interface library so that the LLMs can make proper semantic connections between the signal and the property. LLMs might lack the knowledge and semantic understanding to perform this mapping correctly. For example, the LLM can easily map the API property **acMode** to the **ACMode** CAN signal. But it becomes harder for the LLMs to connect the property **AdaptiveFrontlightSystem** to the signal **AFSOn**. So, in cases like this, it proves vital that there is a need for a readable signal interface library.

So, to create this library, SPAPI Coder takes the preprocessed CAN signal information and generates a library full of readable and easily comprehensible signal interface functions. The LLM names these functions accordingly and adds docstrings based on the available signal names and their descriptions, improving readability and searchability. This is done to restrict the search space and improve the search technique during the next level. Appendix A.3 shows the abstracted signal code for the **ACMode** signal, and the prompt for synthesis can be found in Appendix A.6.

4.4.3 SPAPI Tester at Level 3

It is vital to test any software component that is developed, which makes it trivial to evaluate all the artifacts that SPAPI Coder produces, not only because the signal interface library at Level 3 is LLM-generated, but also to test if it is functional. Testing these functions will help verify that these functions' interface with the CAN bus with regard to the particular vehicle states is valid.

The functions that SPAPI Coder generates at Level 3 will either read a vehicle state from the CAN bus or set a vehicle state. Since all the possible vehicle states are documented in the CANdb in a standardized way, it is possible to test these functions with the same signal information used to generate the function without the need for an LLM-generated test suite.

For instance, once the ***write_ac_mode(acmode)*** signal interface function is generated for the ***ACModeRequest*** signal using its signal specification and the same description, we create test cases, where in the case of this signal, the vehicle states are represented as enums. So, in this scenario, we render test scripts to test **write_ac_mode(acmode)**. This includes testing this function with valid enums, invalid enums, and invalid inputs based on the information available from the signal documentation. In the case where the vehicle states are represented as int or float like ***write_alarm_minute(minute)*** for the ***alarmMin*** signal, these signals have a specific range, and setting a value out of this range makes the transmission of this signal through the CAN bus not possible. So in this case, we test ***write_alarm_minute(minute)*** with boundary values, valid and invalid values.

For the read signals, it is set so that the CAN bus transmits only random valid

values. We then test whether the value from the CAN bus falls within the correct range. For enums, we test that the enums that the CAN bus transmits are actually valid enums for that particular signal, based on the specification. For the suite that tests **ACMode** see Appendix A.11.

Then, based on these test results, we flag the functions with failed and flaky test cases and invoke an LLM to assess what might have caused the test to fail. In case the code is wrong, the signal interface function is rewritten.

4.4.4 Data Postprocessing

Once these CAN signal interface functions are generated, they are turned into numerical vector embeddings using embedding models and stored inside a vector database. This is so that the generated CAN signal interface functions can act as the search space for the program search conducted during the next level of synthesis. The function codes as files are also stored inside a git repository.

4.5 Level 2: Signal - API Property Mapping

Now that the signal interface libraries are generated in Level 3 of the SPAPI Coder pipeline, the next level of the pipeline can be triggered to synthesize API Properties by mapping their values to their corresponding CAN Values.

4.5.1 Data Preprocessing: API Property Representation

When synthesizing API properties using the signal abstractions generated at Level 3, we present the API properties in such a way to the system that it is not overwhelmed by all the properties in the OpenAPI specification. A single OAS may comprise several endpoints, each endpoint containing several API properties. Therefore, presenting the whole OAS to the LLM might make it difficult for the LLM to do the mappings for each API property in an OAS at a single instance.

4.5.2 SPAPI Coder at Level 2

This level involves programming functions for each of the SPAPI endpoint properties, using one or more functions from the signal interface library from Level 3. For instance, the AC case requires get and put functions for ***acMode***, which read/write the ***ACModeRequest*** signal. This may be a one-to-one mapping, but mapping might be one-to-many for some cases, like the ***alarmTime*** property. There, the required signals are, for example, ***alarmMin*** and ***alarmHr***. If the signal library contains a large number of signals, mapping a property to its signal becomes a challenge.

To synthesize the API property, we parse through the OpenAPI specification to get the API properties for each endpoint. For each property in the endpoints, we use the API property and its description as a retrieval context for RAG to retrieve

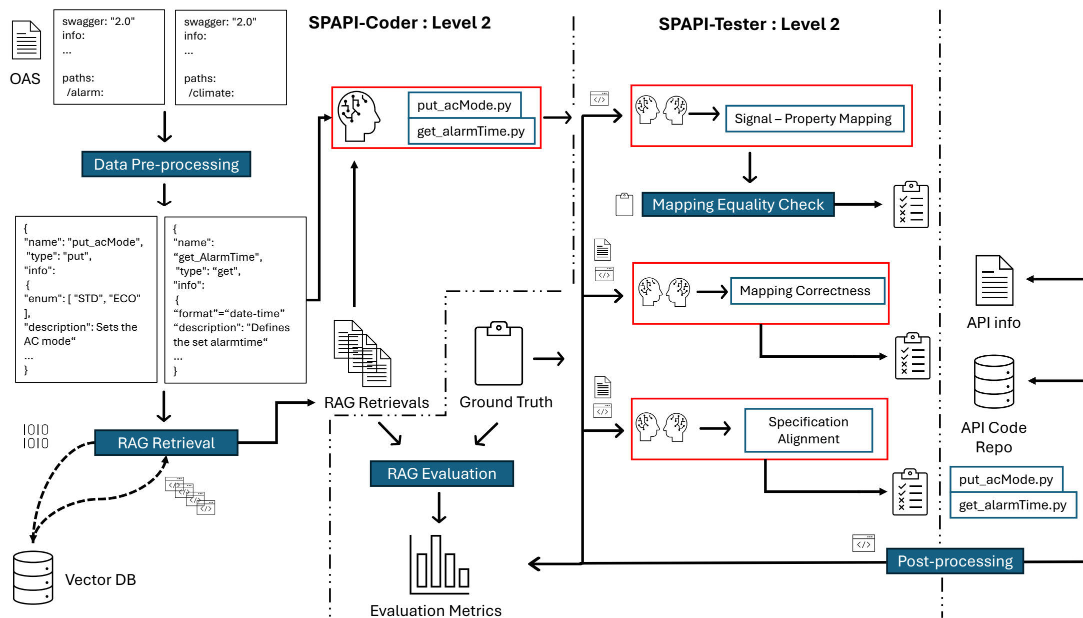

Figure 4.3: Architecture of the system at the CAN signal to property mapping level. Components enclosed in **Red** rectangular highlights indicate that the particular system component invokes an LLM.

only the relevant signals from the signal library stored in the vector database. This shortlists the signals that can be mapped to the API property. We get the top N signal interface functions based on their semantic similarity to the API property and description. Using RAG as a search technique helps conduct an efficient search through the program search space of signal interface functions. Then we invoke the LLM with the task of mapping the signal functions to each API property. So, based on the results from the RAG and the property information from the API documentation, an API property function where the vehicle state to property mapping is done is synthesized.

As part of RAG, vector database querying is used as a search technique because it helps to significantly reduce the search space when compared to other approaches. Other search techniques employed in conventional program synthesis tasks involve mapping each API property to its corresponding signal by going through all of the CAN signals. Whereas in the case of RAG, it has to go through only a number of semantically similar CAN signals to do the correct signal property mapping. A brute force search technique was tried out for previous iterations of SPAPI Coder, but it was not long until it was clear that such methods are not scalable when the system needs to handle hundreds of candidate programs at each generation step.

A synthesized program for **acMode** API property can be found in Appendix A.4, and the prompt for synthesis - in Appendix A.7.

4.5.3 SPAPI Tester at Level 2

For synthesizing the API properties at the second abstraction level, the LLM does the mapping of the API property to a specific vehicle state based on the API specification. For the purpose of testing these mappings, we utilize the ground truth property-state mappings that we already have.

We employ an LLM to extract the mappings done during the synthesis stage. Then we compare the property-state mapping to the ground truth mapping. We invoke two LLMs as judges, one of which checks if the API property value to CAN state mappings are valid. For instance, this LLM checks the code whether the ***"ON"*** API property value is correctly mapped to the state of the signal ***"AcMode.On"***. The other judge checks if the synthesized property follows the API specification from the OAS, using which it was synthesized. An example of failing the specification alignment judge would be that the property function has a branch that returns a string that is not explicitly specified in the specification. The exact criteria that the judges follow can be found in Appendix A.8 and A.9. Binary classification with "YES" and "NO" as verdicts of the judge was selected over having a score such as 1-5. Using score ranges as a metric often led to the need to define what each score represents properly. For example, it is challenging to decide how the LLM should decide whether to give a judgment between 4 and 5. Hence, it becomes essential to define what each score signifies. Whereas in the case of binary classification, there is no need to define each metric to such an extent.

4.5.4 Data Postprocessing

As the properties in an endpoint get synthesized, we group all of the properties of each endpoint together. This grouped API property information will then be used in the synthesis of the API endpoints in the next stage of the pipeline. Furthermore, once each API property gets generated, these property functions are stored in the git repository.

4.6 Level 1: SPAPI Endpoint Implementation

Now that the system has synthesized the API properties, the next step is to synthesize the actual API endpoint implementation.

4.6.1 Data Preprocessing: API Router Boilerplate

OpenAPI specifications are formal specifications for designing RESTful APIs. These specifications are widely used, standardized, and computer-readable while remaining agnostic to the actual API endpoint implementation, making it generalizable. This allows developers to build libraries that utilize these standardized specifications to generate server stub codes and specify types for API properties. So, it is quite common to use libraries to create stub code snippets for an API server in API-first design. This usually involves providing a tool with the OAS for which the tool will

generate a boilerplate server code based on the given specification. Without the utility of such tools to generate boilerplate code for the system, it will be tedious to prompt an LLM to generate an API endpoint implementation. Most libraries that generate the stub code for APIs work only with the latest OAS version, that is OAS 3.x.x. Since the OAS version of SPAPIs is OAS 2.x.x, we use a tool to convert the OAS 2.x.x documents to OAS 3.x.x in order to be compatible with the tools that generate the server code stubs.

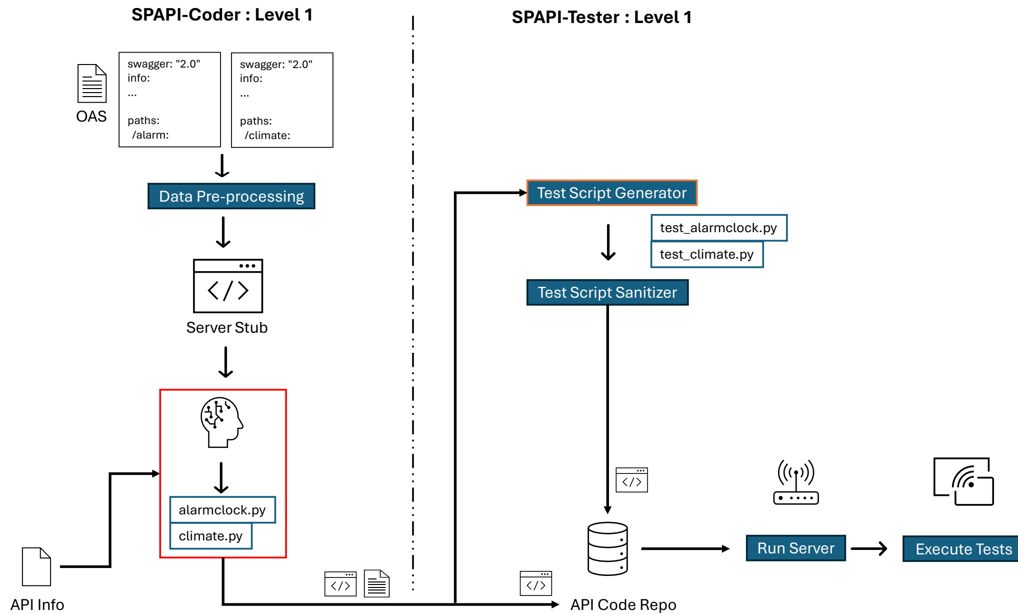

Figure 4.4: Architecture of the system at the API router level. Components enclosed in **Red** rectangular highlights indicate that the particular system component invokes an LLM.

4.6.2 SPAPI Coder at Level 1

The synthesis that happens at this level of the SPAPI Coder pipeline entails organizing the synthesized API properties into practical RESTful APIs for client use. For the ***/cabinclimate*** endpoint, this involves grouping properties like *acMode* and *fanLevel* for both get and put methods. We provide the model with the server stub code and the API property groupings for each endpoint from Level 2, and the model generates a high-level SPAPI endpoint. Once each SPAPI is synthesized, these APIs are stored in the same git repository as before. An example endpoint implementation can be seen in Appendix A.5, and the prompt for synthesis can be found in Appendix A.10.

4.6.3 SPAPI Tester at Level 1

Volvo has developed a tool that can generate test scripts to test SPAPI endpoints which has been in active use [88]. We employ this tool to automatically generate test scripts to test the endpoints that SPAPI Coder generates.

4.6.3.1 Data Preprocessing: Sanitizing of API Test Scripts

Even though the test scripts are readily available and can be generated using the tool to test SPAPIs, the test scripts will fail to work because of the difference in testing dependencies between testing the API developed manually and the implementation developed using automation. So, the test scripts are sanitized by altering the dependencies in order for the test scripts to be functional (see Appendix A.12). After the sanitation step, these test scripts are also saved into the same repository.

4.6.3.2 Testing SPAPI Endpoints

As the automation system has been generating artifacts through all the abstraction levels, they have also been stored in the git repository. Ultimately, the git repository is populated with functional API endpoint implementations as shown in Fig. 4.5.

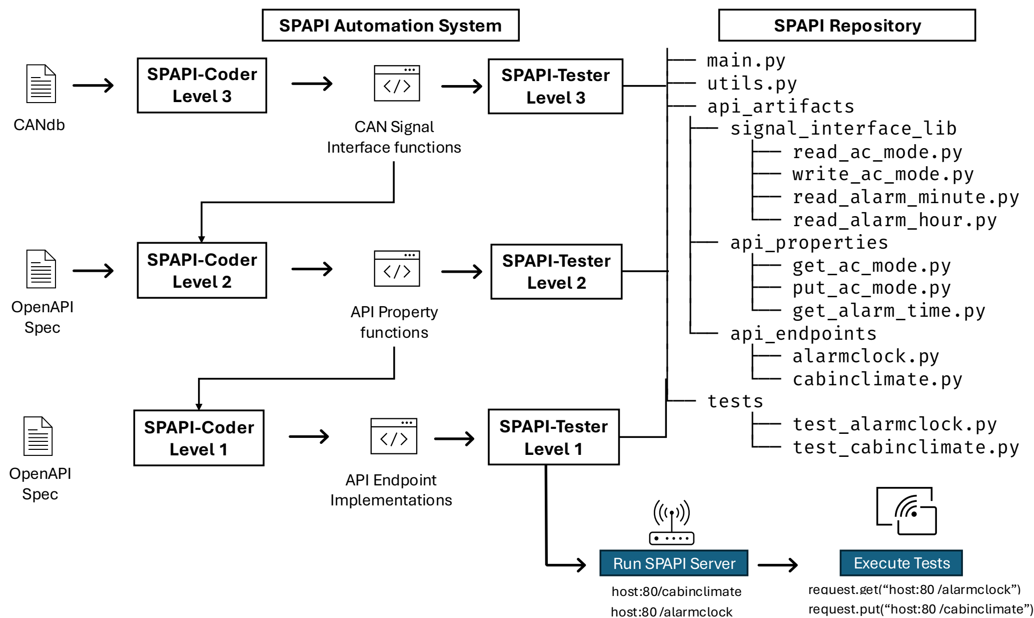

Figure 4.5: System workflow for SPAPI automation

Installing the dependencies and starting the API server allows for exposing the generated SPAPI endpoints, against which the test scripts are executed.

5

Results

While building a Compound AI System that automates the SPAPI development process is the primary motivation behind this thesis, the successful realization of whether the system implementation is actually effective in synthesizing API Endpoints compliant with the specification can only be assessed by evaluating the artifacts that the system synthesizes at each abstraction level. The functional correctness of the components synthesized at all three levels of the pipeline is fundamental to the proper functioning of SPAPI.

For example, suppose an API property for *CabinClimate* synthesized at Level 2 is functionally wrong. In that case, it makes the entire endpoint non-functional even though all the other generated artifacts for this particular endpoint might be functionally correct. This urges the need to select the best combination of tools and techniques to synthesize functionally correct API components at each abstraction level, leading to several experiments to pursue this need.

5.1 Experiment Scope

The automation system is capable of synthesizing API components at all three levels. The data available to us consists of 50 APIs that in total feature around 170 endpoints. Each endpoint contains somewhere between 2 and 10 API properties, or even more in some cases. Considering this, generating APIs using all these specifications would require around 800 CAN signal to API property mappings. If we handpick only the signals from the database that are supposed to be mapped, introducing bias becomes inevitable. To eliminate potential bias, presenting the system with the entire CANdb, containing around 3500 signals, is needed so the system can search to map the correct CAN signal over a search space of a diverse group of related and unrelated signals. Doing so would result in the pipeline making approximately 4000 LLM calls for one iteration.

The goal of the experimentation is to find the best combination of tools and techniques to make the system perform the best by producing functional components at each abstraction level. However, because of time and resource constraints, we opted to scope down the number of APIs to assess the best tools and techniques.

For experimentation with the automation pipeline, we proceeded to do the experimentation with 10 APIs consisting of around 16 endpoints. These APIs were handpicked so that there are APIs of varying complexity based on the number and implementation logic of the API properties. For instance, we picked APIs that had relatively low and high number of properties, and ones that had properties with one-to-one and one-to-many signal mappings. A curated subset of ~120 signals directly mappable to API properties in the selected 10 OASs was extracted for the signals. To mitigate the aforementioned bias toward easily mappable signals and enhance the generalization of the performance of SPAPI Coder in doing the correct mappings, approximately 380 additional signals were included, resulting in a total of ~500 signals. This curated dataset enables the system to learn robust mappings by exposing it to a broader range of signal properties, supporting the signal abstraction and mapping tasks at Levels 3 and 2 of the automation pipeline.

5.2 Experiment Design

The evaluation of the proposed system was conducted at the three abstraction levels: *Signal Abstraction* (Level 3), *Signal–API Property Mapping* (Level 2), and *API Endpoint Implementation* (Level 1). Additionally, the performance of the Retrieval-Augmented Generation component, which is crucial for Level 2 onwards, was specifically evaluated. Different embedding models, LLMs, and prompting techniques were employed to identify the most effective configurations. Due to resource and time constraints, a greedy approach to evaluation was utilized, meaning that the best configuration was picked for each level, and it was used at the subsequent levels. This approach assumed that the combination of local optima would approximate a global optimum, given the impracticality of testing all unique combinations of configurations across levels.

For all the instances where we use an LLM to generate an API artifact, we employ only the GPT-4o, Llama 3.3 70b, and Qwen2.5 Coder 32b models. This is due to the fact that the reasoning models by DeepSeek are significantly slower in generating answers compared to the other models, acting as a bottleneck when running the pipeline. Although, when using the LLMs as judges, we employ a reasoning model to curate findings into whether the reasoning capabilities of LLMs contribute to their ability to judge when compared to non-reasoning models. The Qwen Coder model is not employed as a judge, as it is the smallest model we have, and it is fine-tuned to produce code. Temperature, in the context of LLMs, is often associated with creativity and novelty. But recent research shows that the correlation between temperature and creativity and novelty is much weaker than the claims [69]. Another study shows that temperatures below one do not have a statistically significant impact on LLM performance for problem-solving tasks [71]. Keeping these studies in mind and the need for better reproducibility across different pipeline runs, we decided to set the temperature to 0 for all LLM calls in the pipeline.

5.3 Level 3

The signal abstraction level involves synthesizing interface functions to read or write vehicle states from or to the CAN Bus. The performance of LLMs in generating signal interface functions at Level 3 is evaluated based on test outcomes, categorized as Pass, Fail, or Flaky. In our case, a flaky test case outcome means that the tested object is functionally correct under expected conditions but fails to handle unexpected situations gracefully, often due to insufficient exception handling. Table 5.1 summarizes these results for test results of the SPAPI Coder generated signal interface functions with GPT-4o, Llama 3.3 70b, and Qwen2.5 Coder 32b, using various prompting techniques: zero-shot, one-shot, few-shot, and few-shot with chain-of-thought.

SPAPI Coder's performance using **GPT-4o** demonstrates strong and consistent performance, particularly with one-shot, few-shot, and few-shot with Chain-of-Thought (CoT) prompting, achieving 429 passes and 2 flaky results, with no failures. Unsurprisingly, its zero-shot performance is slightly lower, with 364 passes, 1 fail, and 32 flaky results.

Results of the system with **Llama 3.3 70b** are less consistent compared to GPT-4o. Llama performs best with few-shot chain-of-thought prompting, resulting in 423 passes, 3 fails, and 5 flaky results. The zero-shot performance is comparable with 421 passes, 3 fails, and 7 flaky results. Performance decreases with one-shot (394 passes, 16 fails, 21 flaky) and few-shot (377 passes, 50 fails, 4 flaky) prompting. The pipeline led to an unexpected failure while conducting the experiment for the zero-shot prompting technique with the Llama model, after it had generated around 300 signal interface functions due to a programming error. By writing invalid Python code, the model caused a system failure during the post-processing stage in the Level 3 pipeline. Hence, zero-shot with Chain-of-Thought was used as the prompting technique to carry out this experiment.

SPAPI Coder, paired up with **Qwen2.5 Coder 32b** achieves its highest pass rate with few-shot CoT prompting (428 passes, 0 fails, 3 flaky), keeping it consistent with the previously tested models. It is closely followed by few-shot (427 passes, 1 fail, 3 flaky) and one-shot prompting (425 passes, 2 fails, 4 flaky). The zero-shot performance is considerably lower, with 288 passes, 8 fails, and 135 flaky results.

These results highlight the importance of prompt engineering when employing LLMs for program synthesis tasks. The consistent upward trend of passing test cases as language models are given more input-output examples to work with supports the claims from previous research that LLMs are few-shot learners [10, 56, 91]. Although the correctness of functionality is important, selecting the best configuration solely based on the number of passing test cases is not enough. The other main objective of Level 3 is to produce a meaningful interface from vague information about a CAN signal. For instance, if ***ACMode*** signal is falsely understood as "Alternating Current Mode" by the LLM, the function signature and related docstrings will be semantically wrong, considering that the mapping module will likely look for a signal

LLM	Technique	Test Results		
		Pass	Fail	Flaky
GPT-4o	Zero-shot	364	1	32
	One-shot	429	0	2
	Few shot	429	0	2
	Few shot + CoT	429	0	2
Llama 3.3 70b	Zero-shot*	421	3	7
	One-shot	394	16	21
	Few shot	377	50	4
	Few shot + CoT	423	3	5
Qwen2.5 Coder 32b	Zero-shot	288	8	135
	One-shot	425	2	4
	Few shot	427	1	3
	Few shot + CoT	428	0	3

Table 5.1: Level 3 LLM performance evaluation by prompting technique based on executed tests. * with CoT

related to "Air Conditioning". In this case, the mapping at the next level will likely be incorrect, as the chance that the RAG returns the correct signal is low. With this in mind, an additional evaluation step for the CAN signal abstraction level must be developed.

5.4 RAG

The effectiveness of the RAG component in retrieving relevant signal interface functions is assessed using different embedding models and LLMs, with and without reranking. Moreover, given the results from the previous evaluation step, CAN interface functions generated by few-shot with chain-of-thought prompting are selected for each available language model. The code corpus is then embedded with each embedding model and put into Chroma-powered vector database. Retrieval is evaluated using five retrieval configurations. Three of them do not involve reranking: top 5, top 10, and top 20 results. The remaining two configurations feature a dedicated reranker model, tinyBERT [1], where we first retrieve the top $2 \cdot N$ results, rerank them, and return only half of the reranked results, top N. Findings are presented in Table 5.2. Note that when language models are mentioned in this subsection, it refers to the signal interface code generated by that model in Level 3.

For **Instructor XL** embedding model, without reranking, accuracy for GPT-4o ranges from 0.86 (top 5) to 0.94 (top 20). Signal code generated by Llama 3.3 70b shows accuracies from 0.84 (top 5) to 0.95 (top 20), and Qwen2.5 Coder 32b from 0.88 (top 5) to 0.96 (top 20). With reranking, the accuracies for top 5 and top 10

[1] https://huggingface.co/cross-encoder/ms-marco-TinyBERT-L2-v2

results are generally similar or slightly lower for some configurations compared to no reranking. For instance, with reranking (top 10), GPT-4o achieves 0.92, Llama 3.3 70b achieves 0.91, and Qwen2.5 Coder 32b achieves 0.91.

Using **BGE-M3** embedding model without reranking, GPT-4o accuracies are 0.85 (top 5) to 0.94 (top 20). Llama 3.3 70b ranges from 0.84 (top 5) to 0.94 (top 20), and Qwen2.5 Coder 32b from 0.86 (top 5) to 0.94 (top 20). Reranking with BGE-M3 shows varied results; for example, at top 10, GPT-4o has 0.87, Llama 3.3 70b has 0.87, and Qwen2.5 Coder 32b has 0.91.

With **Text Embedding Ada 002**, without reranking, GPT-4o accuracies are 0.81 (top 5) to 0.94 (top 20). Llama 3.3 70b shows 0.83 (top 5) to 0.94 (top 20), and Qwen2.5 Coder 32b shows 0.83 (top 5) to 0.93 (top 20). Reranking (top 10) yields accuracies of 0.91 for GPT-4o, 0.88 for Llama 3.3 70b, and 0.89 for Qwen2.5 Coder 32b.

Text Embedding 3 Large model generally yields high accuracies. Without reranking, GPT-4o ranged from 0.88 (top 5) to 0.95 (top 20). Llama 3.3 70b achieves 0.88 (top 5) to 0.97 (top 20), and Qwen2.5 Coder 32b shows 0.87 (top 5) to 0.96 (top 20). With reranking (top 10), accuracies are 0.87 for GPT-4o, 0.85 for Llama 3.3 70b, and 0.91 for Qwen2.5 Coder 32b.

The evaluation results reveal several key insights about the RAG component's performance. Notably, reranking does not consistently improve accuracy across configurations but often yields similar or lower accuracies compared to non-reranked setups. This contradicts our expectations that reranking would refine result relevance. As anticipated, increasing the top N results (from 5 to 20) generally improves accuracy across all embedding models and LLMs. The top 20 configurations achieve the highest accuracies (e.g., up to 0.97 for Llama 3.3 with Embedding 3 Large). However, this increase in accuracy comes at the cost of lower precision, as retrieving more results introduces additional, potentially irrelevant context. This poses a challenge for the subsequent mapping step in the pipeline, where selecting the most relevant 1 or 2 signal interface functions from a larger pool (e.g., 20 candidates) becomes more complex compared to a smaller set (e.g., 10 candidates), potentially impacting overall system efficiency.

Despite GPT-4o generating signal interfaces with the best test case results, generations with Llama 3.3 70b language model paired with the Text Embedding 3 Large embedding model and a top 10 retrieval configuration without reranking are selected as the optimal setup for this study. This choice is driven by its strong performance within the RAG pipeline as it achieves a high accuracy of 0.94, the best accuracy at top 10, and is on par with the best top 20 configurations with other embedding models. This balances the trade-off between retrieval accuracy and precision.

Embedding Model	Retrieval	Accuracy		
		GPT - 4o	**LLama 3.3**	**Qwen Coder**
Instructor XL	Top 5	0.86	0.84	0.88
	Top 10	0.92	0.93	0.92
	Top 20	0.94	0.95	0.96
	Top 5 (R)*	0.86	0.86	0.84
	Top 10 (R)*	0.92	0.91	0.91
BGE-M3	Top 5	0.85	0.84	0.86
	Top 10	0.89	0.90	0.92
	Top 20	0.94	0.94	0.94
	Top 5 (R)*	0.81	0.82	0.85
	Top 10 (R)*	0.87	0.87	0.91
Embedding Ada 002	Top 5	0.81	0.83	0.83
	Top 10	0.92	0.92	0.88
	Top 20	0.94	0.94	0.93
	Top 5 (R)*	0.88	0.83	0.85
	Top 10 (R)*	0.91	0.88	0.89
Embedding 3 Large	Top 5	0.88	0.88	0.87
	Top 10	0.92	**0.94**	0.91
	Top 20	0.95	0.97	0.96
	Top 5 (R)*	0.83	0.81	0.85
	Top 10 (R)*	0.87	0.85	0.91

Table 5.2: Retrieval evaluation
* (R) indicates that a reranker was used in the retrieval process.
The LLM names indicate the model using which the Level 3 library was generated.
The selected combination for the next generation phase is in **bold**.

5.5 Level 2

The performance at Level 2 involves evaluating SPAPI Coder's ability to generate API properties, explicitly focusing on specification alignment and mapping correctness. Using the best Level 3 generation picked from the step above, the system creates API properties with the same LLMs as before. Yet, different LLMs are used as evaluators (judges) instead of the previous level's evaluations. Since this evaluation step requires a higher degree of reasoning than pure code generation, Qwen2.5 Coder 32b is replaced by DeepSeek R1 Distill Llama 70b, which is a reasoning model. Moreover, an ensemble of LLMs-as-judge is introduced. It represents a majority voting system including GPT-4o, Llama 3.3 70b, and DeepSeek R1 Distill Llama 70b. Table 5.3 details the performance metrics (Accuracy, F1-score, Recall, Precision) for the judges. To calculate these metrics, manual labelling of the generated API properties was required to establish a ground truth, as automated evaluation alone was insufficient for capturing the nuanced reasoning needed in this step.

For **GPT-4o** generations, specification alignment metrics range from 0.78–0.83 (Accuracy), 0.87–0.90 (F1), 0.89–0.94 (Recall), and 0.84–0.87 (Precision), with Llama as the evaluator achieving the highest accuracy (0.83). For mapping correctness, GPT-4o generations show 0.76–0.84 (Accuracy), 0.86–0.90 (F1), 0.92–0.98 (Recall), and 0.77–0.85 (Precision), with DeepSeek performing best (0.84 Accuracy). **Llama 3.3** generations follow a similar trend, with mapping correctness peaking at 0.85 (Accuracy) and 0.90 (F1) with the ensemble judge, though specification alignment is slightly lower, e.g., 0.70 (Accuracy) with DeepSeek. **Qwen2.5 Coder 32b** generations consistently show lower performance, with specification alignment metrics as low as 0.71 (Accuracy) and mapping correctness peaking at 0.83 (Accuracy) with the ensemble.

LLM	Judge Criteria	Evaluator	Acc	F1	Recall	Precision
GPT-4o	Specification	GPT-4o	0.78	0.87	0.89	0.84
		Llama	0.83	0.90	0.94	0.87
		DeepSeek	0.82	0.89	0.92	0.86
		Ensemble	0.79	0.88	0.92	0.84
	Mapping	GPT-4o	0.76	0.86	0.98	0.77
		Llama	0.81	0.88	0.92	0.85
		DeepSeek	0.84	0.90	0.96	0.85
		Ensemble	0.83	0.90	0.98	0.83
Llama 3.3	Specification	GPT-4o	0.75	0.84	0.91	0.77
		Llama	0.78	0.85	0.90	0.81
		DeepSeek	0.70	0.79	0.80	0.78
		Ensemble	0.77	0.85	0.90	0.80
	Mapping	GPT-4o	0.83	0.90	0.99	0.83
		Llama	0.78	0.86	0.91	0.82
		DeepSeek	0.83	0.89	0.89	0.90
		Ensemble	0.85	0.90	0.96	0.86
Qwen Coder	Specification	GPT-4o	0.72	0.82	0.90	0.75
		Llama	0.71	0.80	0.81	0.78
		DeepSeek	0.73	0.81	0.83	0.79
		Ensemble	0.75	0.83	0.88	0.78
	Mapping	GPT-4o	0.79	0.88	0.98	0.80
		Llama	0.82	0.89	0.94	0.84
		DeepSeek	0.79	0.86	0.89	0.84
		Ensemble	0.83	0.89	0.97	0.83

Table 5.3: Performance metrics for the judges while generating API properties

The results suggest that LLMs may not be reliable as standalone judges for evaluating API property generations due to their inconsistent accuracy, particularly for specification alignment, where metrics ranged from 0.70 to 0.83 across models. However, their high recall, reaching up to 0.99 for Llama 3.3 with GPT-4o judge in

mapping correctness, indicates strong capability in identifying relevant mappings, potentially making them suitable as initial screening tools to capture a broad set of correct outputs. Specification alignment proves more challenging, with lower accuracy and precision, likely due to the complex reasoning required to assess detailed specifications. The ensemble judge provides balanced performance, reducing individual model biases, but it does consistently outperform single judges like DeepSeek or Llama. These findings highlight that while LLMs excel in high-recall scenarios, their effectiveness as standalone evaluators is limited by variable accuracy, particularly for nuanced tasks, with stronger reasoning models like DeepSeek showing promise. The manual labeling process, while labor-intensive, is critical to ensuring reliable ground truth data, and it surely highlights the current limitations of fully automated evaluation for complex reasoning tasks.

Table 5.4 further highlights the number of correctly generated API properties with respect to the ground truth labels. GPT-4o achieved mapping correctness for 96 properties and specification alignment for 102 properties. It successfully met both criteria for 92 properties. Llama 3.3 70b demonstrated mapping correctness for 96 properties and specification alignment for 90 properties, with 83 properties satisfying both. Qwen2.5 Coder 32b achieved mapping correctness for 96 properties and specification alignment for 86 properties. Both criteria were met for 77 properties.

We select GPT-4o as the best generation model for this level due to its superior performance in generating API properties, as evidenced by the highest counts in Table 5.4. Its ability to balance specification alignment and mapping correctness make it the most reliable choice for producing high-quality API properties.

	Number of API properties		
LLM	**Mapping Correctness**	**Spec Alignment**	**Both**
GPT-4o	96	102	**92**
Llama 3.3 70b	96	90	83
Qwen2.5 Coder 32b	96	86	77

Table 5.4: Number of correctly generated API properties per LLM
Best performer and the chosen configuration for the next phase is in **bold**

5.6 Level 1

The results for Level 1, as presented in Table 5.5, demonstrate the performance of SPAPI Coder in synthesizing API endpoint implementations across various endpoints and methods. These combinations of endpoint and methods are tested using scripts generated by Volvo's AI-powered SPAPI Tester tool.

Test case execution results show that the performance across the three LLMs—GPT-4o, Llama 3.3 70b, and Qwen2.5 Coder 32b—is broadly comparable. Most endpoints achieve consistent pass/fail outcomes. For instance, endpoints like /climate (GET

API	Endpoint	Method	Pass/Fail		
			GPT-4o	Llama	Qwen
Alarm	/alarms	get	3/1	3/1	3/1
	/alarms	put	0/3	0/3	0/3
Cabin Climate	/climate	get	3/0	3/0	3/0
	/climate	put	4/0	4/0	4/0
Fuel	/fuel	get	0/3	3/0	3/0
	/fuelsettings	put	3/0	3/0	3/0
	/energy	get	3/0	3/0	3/0
Screen	/screens	get	2/1	2/1	2/1
	/screens	put	3/0	3/0	3/0
Driver	/driversettings	get	2/1	2/1	2/1
	/driversettings	put	0/3	0/3	0/3
Battery	/battery	get	2/2	2/2	2/2
	/batterysettings	put	2/0	2/0	2/0
Charging	/connection	get	3/0	3/0	3/0
	/charging	get	2/1	1/2	1/2
	/chargingsettings	put	0/4	0/4	0/4
Auxiliary Battery	/absettings	get	3/0	3/0	3/0
	/absettings	put	3/0	3/0	3/0
Velocity	/velocity	get	3/0	3/0	3/0
Wiper	/wipers	get	4/0	4/0	4/0
	/wipersettings	put	2/0	2/0	2/0
			47/19	49/17	49/17

Table 5.5: API endpoint test results with LLM performance

and PUT), /fuelsettings (PUT), /energy (GET), /connection (GET), /absettings (GET and PUT), /velocity (GET), and /wipers (GET) achieve perfect pass rates across all models, indicating robust endpoint synthesis. However, specific endpoints, such as /alarms (PUT), /driversettings (PUT), and /chargingsettings (PUT), consistently failed (0/3 or 0/4) across all LLMs, which strongly suggested that the issues are likely not in the endpoint synthesis itself but in upstream processes.

Upon further review, it became apparent that none of the failures are related to endpoint generations at Level 1. In fact, most failures at this level can be attributed to errors originating in Level 2, particularly incorrect signal mappings, wrong setting logic for signals, or misaligned API properties. Although the levels presented are autonomous and abstracted in a hierarchical way, they are not independent of each other. For instance, Level 1 endpoint implementations rely on multiple related API property functions, each of which also depends on one or more signal interface functions. For instance, if a signal function produces unexpected behavior, it will likely impact the functionality of the corresponding API properties and the endpoints it is featured in. Similarly, even if the signal implementation is correct, in case the property is mapped to the wrong signal, the endpoint implementation will return

unexpected results. Problems at lower abstraction levels propagate to Level 1 and cause the generated endpoints to fail validation during testing

What is more, endpoints like /battery (get) and /charging (get) show mixed results. For example, an equal number of test cases pass and fail due to inconsistent property mappings from Level 2 affecting the endpoint implementation.

These results uncovered the dependency of Level 1 success on the quality of Level 2 outputs. Even though generations at this level were solid and had no breaking issues, it is apparent that improved mapping correctness and specification alignment are needed in earlier stages to enhance overall system performance.

Further analysis of the number of passing test cases across all endpoints and methods shows that Llama 3.3 70b and Qwen2.5 Coder 32b achieve identical results - 49 passing and 17 failing test cases. In fact, they pass and fail the exact same test cases. At the same time, although slightly worse, GPT-4o performs very similarly, with 47 passing test cases against 17 failing. A review of the extra two failing test cases shows that endpoint generation with GPT-4o cannot handle the return type of an API property belonging to GET */fuel*.

Lastly, Table 5.6 presents an aggregated view of the test results on the pure API-level. This higher-level summary reveals that four APIs, namely Cabin Climate, Auxiliary Battery, Velocity, and Wiper, are consistently correct across all three LLM generations. On the contrary, some of the endpoints for Alarm, Screen, Driver, Battery, and Charging APIs failed for all LLMs. This suggests systemic challenges or complexities within these APIs. When it comes to Fuel API, there is a visible difference in performance as it fails under GPT-4o, yet both Llama 3.3 70b and Qwen2.5 Coder 32b are successful in passing all endpoint tests. All in all, considering a pass as passing all endpoint tests and otherwise a fail, the passing rate for GPT-4o is 40% while it is 50% for Llama and Qwen.

API	Overall Pass/Fail		
	GPT-4o	**Llama**	**Qwen**
Alarm	F	F	F
Cabin Climate	P	P	P
Fuel	F	P	P
Screen	F	F	F
Driver	F	F	F
Battery	F	F	F
Charging	F	F	F
Auxiliary Battery	P	P	P
Velocity	P	P	P
Wiper	P	P	P
	40%	50%	50%

Table 5.6: API-level test results summary with LLM performance. An API passes (P) if all its endpoints pass; otherwise, it fails (F).

6

Discussion

6.1 Alignment with Design-Science Research

Addressing the DSR guidelines, with regards to the *problem relevance*, this thesis aims to solve a real-world problem, which in this case is to automate an otherwise cumbersome and manual SPAPI development process. This problem is solved by *designing an artifact*, a functional SPAPI automation system that we develop to synthesize RESTful APIs. The design of the SPAPI Coder artifact did not follow a linear path but followed rigorous and iterative research cycles. The development of SPAPI Coder involved a *search process* for the most effective tools and prompting techniques, while encompassing established rigorous research methods like program synthesis, also not limited to library learning and process decomposition.

Initially, SPAPI Coder was designed to generate API endpoints directly from input specifications. However, this approach proved insufficient, as it frequently failed to produce functional and executable API endpoints. This initial finding was crucial in highlighting the need for a more sophisticated artifact design.

Based on these findings, we refined the system to adopt a modular design, incorporating process decomposition into three abstraction levels and utilizing program synthesis. This refinement was informed by insights from existing research on program synthesis and task decomposition [68, 97]. By breaking down the complex problem into smaller, manageable sub-problems and integrating more detailed knowledge of the manual SPAPI development process, the system's ability to generate robust API endpoints significantly improved.

Then, subsequent iterations focused on further enhancing SPAPI Coder's capabilities by exploring different approaches to capture user intent and experimenting with various search techniques and evaluation methods to optimize system performance, thus answering all the research questions. Also, continuously analyzing the generated code through evaluation helped refine and improve the prompts in subsequent iterations. This iterative process, in which each step was designed, built, evaluated, and refined based on previous outcomes, was fundamental to the successful development of SPAPI Coder.

When it comes to evaluating the utility of the designed artifact, it can be assessed

using a number of terms like functionality, completeness, consistency, accuracy, performance, etc. DSR suggests observational, analytical, experimental, testing, and descriptive methods for evaluating the artifact [31]. Evaluating the utility of the SPAPI Coder system implies evaluating the effectiveness of the system in generating REST APIs. So, we employed evaluation methods such as *testing* for the levels 3 and 1 and the pipeline, and also performed controlled *experiments*, where we studied the artifacts' performance in generating API endpoints with different tools and techniques. The successful realization of this research, including the design and evaluation of the artifact, can further *communicate this research*, to broaden this research space on top of contributing by meeting the technological and business needs for further work that falls under a similar scope of automating software development workflows.

6.2 Research Questions

The ideas behind SPAPI Coder's design and implementation were guided by the need to address these three research questions.

6.2.1 RQ1: Capturing User Intent

Motivation: *What is an effective way to capture the user intent so as to synthesize functionally valid API endpoints?*

In order for a functional program to be synthesized, the system that synthesizes the program needs to know the intent for which the program is being synthesized [23]. This intent is usually presented to the system in the form of formal specifications or constraints. SPAPI Coder's ultimate intent is to synthesize API endpoints. But the complex workflow behind the SPAPI development process makes it so that the system needs to capture the intents of the lower level, like the mapping of CAN signals to their respective API properties. For SPAPI Coder to capture this user intent, the CAN signal specifications should be presented to the system in such a manner that it can use the CAN signal specifications and map them to their corresponding API properties.

As CAN signal communication plays a fundamental role in the functioning of SPAPIs that the system synthesizes, the problem arises during automation, where you have to solve these two problems.

- **1:** Semantically map the CAN signal to its API properties while synthesizing the API property.
- **2:** The need for a system component that the synthesized API property interacts with to facilitate CAN signal communication.

Solution: While presenting the CAN signal information as text from the CANdb to the system can solve the first problem by being able to semantically map CAN signals to their corresponding API properties based on the information provided. But, there

is still a need to develop the system component that will enable the synthesized API property to communicate with the CAN network. To solve this, instead of constructing a search space of textual but structured CAN signal information during synthesis at Level 2, we instead generate and populate the search space with a library containing CAN signal interface functions, which are functional modular components that enable the API property functions to perform CAN signal transmission. As these library functions are generated using the same CAN signal information, it is still possible to use these functions to make semantic connections with the API properties.

6.2.2 RQ2: Search Space and Search Technique

Motivation: *How can an effective search space be constructed to enable both efficient and accurate synthesis of API endpoints? What technique can be used to perform a computationally efficient search within this space?*

Now that the way to effectively convey the user intent to the system is identified as creating a program search space with signal interface functions, a search should be conducted over the program space to find the candidate CAN signal functions. Building an effective search space by program synthesis definitions requires overcoming these two obstacles:

- **1:** The need for the search space to be "expressive".
- **2:** The need for a restrictive enough search space for the search to be efficient.

Solution: An "expressive" search space is one where the candidate programs in the search space, using which the final solution will be synthesized, incorporate enough information to be semantically relevant to the intent behind the search. This way, the system can use it to arrive at the correct solution. For example, mapping the CAN signals to their corresponding API properties entails that the LLM, which is the synthesizer, can establish a semantic connection between an API property and its relevant programs in the search space. In the case of synthesizing an API property for ***acMode***, while performing a search for the signals, it is necessary that the CAN signals are well represented in such a way that the search technique can find the relevant signals. The search space can contain functions for signals like ***"ACMode"***, ***"ACFanLevel"***, which are relevant to the API property for which the synthesizer is performing the search. But the search can also contain signals like ***"ACLevel"***, similar to the aforementioned signals, but could relate to something completely different, like Alternating Current Level. To mitigate this, while generating the CAN signal library, we attain extra information about each signal from proprietary documentation sources. We make the functions in the search space more expressive by prompting the LLM to write docstrings with the natural language signal description. Doing so enables the search technique to be more efficient as it restricts the search space of candidate programs based on semantic similarity.

While constructing an effective and expressive search space by itself enhances the system's ability to perform an efficient search, the ability to further restrict the program search space was possible because of the vector-search technique that we employed in our compound AI system. Since the search space was populated as embeddings in a vector database compared to the use of DSL, which is the most common technique to construct an effective search space in program synthesis, performing a search enables efficient finding of semantically similar data points. This is possible because vector databases perform approximate nearest neighbour search, which can be less exhaustive than using traditional search techniques. So, utilizing a vector database as storage for the search space and employing RAG that performs a vector search enables SPAPI Coder to build an effective search space and operate an efficient search to enhance program synthesis.

6.2.3 RQ3: Evaluating Synthesized Artifacts

Motivation: *How can the quality and utility of the generated API endpoints be evaluated?*

Now that the system is successfully able to synthesize API artifacts, any synthesized program is evaluated to determine whether it adheres to the specification by which it was synthesized. As SPAPI Coder generates modular API components across three different levels of abstraction, each level with a different specification, it becomes necessary to evaluate whether the synthesized components are functional.

Solution: To evaluate the synthesized components, we employ dynamic program analysis by executing the API components using test scripts for the synthesized programs at Levels 3 and 1 of the SPAPI Coder pipeline. We employ LLMs as judges to evaluate the LLM-synthesized API properties at Level 2 of the SPAPI Coder pipeline. Doing so helps evaluate the utility of the LLM-synthesized API components across all three abstraction levels. Evaluating the artifacts at each level ensures robustness and correctness at the module level. Having functionally solid individual modules means that the interactions between them are also more reliable. For example, an error that occurs at the artifact synthesized at Level 2 can propagate to the next level. Skipping the evaluation of an LLM-synthesized artifact at a certain level means that all the potential issues are inherited by the next. Issues, as mentioned earlier, can only be determined during the evaluation of the actual endpoint implementation.

The results for using language models as judges suggest that they cannot be blindly trusted at this stage. On the upside, they show great potential for being used as initial screening agents to filter out obvious candidates. Future advancements in GenAI will likely make them a more viable choice for autonomous assessments of this sort.

6.3 Threats to Validity

6.3.1 Conclusion Validity

As the experimental setup of the study focused on a relatively modest sample size of APIs, properties, signals, and test cases, the statistical power of our evaluations is inherently constrained. This makes it challenging to come to definitive and generalizable conclusions about the superiority of one technique or language model over the other.

The study found that LLMs as judges had varying accuracy but relatively high recall, which suggested that they could be used as initial screening tools to reduce human effort. This conclusion is based on the specific models used and the tasks provided. Further research would be needed to confirm whether this claim holds in broader contexts.

Several confounding variables may have influenced both the modules' output quality and the judges' evaluations. Such variables could be prompt phrasing, the structure of intermediate artifacts, or the ordering of retrieved context. Without randomized control experiments or ablation studies, isolating the mechanisms by which different factors affect system performance is difficult.

6.3.2 Internal Validity

One of the most prominent internal threats is related to the manual labeling process used to generate ground-truth data for evaluating property mappings' correctness and alignment to the specification. While we adopted a systematic approach for annotation, the process remains naturally subjective. Subjectivity could lead to inconsistencies in labels, especially when determining whether a generated mapping or code snippet aligns with the intended behavior.

The decision to limit the number of APIs available for our experiments to 10 APIs out of 50 and 500 CAN signals out of 3500 could potentially introduce selection bias. Even though the selected APIs and signals were specifically curated to include variety and cover edge cases, the final subset might not fully represent the complexity and diversity of all SPAPIs and all signals available in the entire CANdb.

The use of proprietary information and tools could mean that any biases or limitations in them will likely propagate into the results of this study. For example, the test cases generated by the SPAPI tester tool are assumed to be a gold standard for testing the APIs generated by the proposed system. Still, the possibility of some test cases being wrong cannot be completely ruled out.

Another identified internal validity threat is the greedy approach to evaluation, in which the best configuration from one level is used in subsequent levels. This approach assumes that local optima approximate a global optimum. However, it is possible that this is a false assumption, and the optimal configuration for an earlier

stage might not be optimal for the entire pipeline.

6.3.3 External Validity

An example of an external validity threat is that the system is designed and evaluated within a specific context involving Volvo's SPAPI development. Although signal information in CANdb and OpenAPI specifications are standardized to a degree, other proprietary data sources and the very nature of the signals most likely remain unique to this domain. Unfortunately, this means that the performance of SPAPI Coder and the effectiveness of the techniques used might not be directly transferable to other domains in the software engineering world. In fact, out-of-the-box usability might not be possible in other automotive manufacturers.

Another significant limitation is related not only to the specific language models chosen for generation and evaluation tasks, namely GPT-4o, LLaMA 3.3 70B, Qwen2.5 Coder 32B, and DeepSeek R1 Distill Llama 70B, but also to the specific set of embedding models chosen for RAG. While these represent state-of-the-art models at the time of writing, the rapid pace of LLM development implies that their relative performance and availability may change in the near future. As such, the conclusions drawn regarding model accuracy, judgment reliability, or code synthesis quality may not hold for future iterations of these models or for different models not evaluated in this study. For instance, the choice of GPT-4o was partly due to its availability at Volvo Trucks, and it was accessed through a company-specific endpoint. As a result, the model might be partially adapted to Volvo's needs and might not represent the exact model available through OpenAI's API.

6.4 Future Work

While this thesis presents a functional and modular system for automatically generating RESTful APIs, several directions remain open for further research and refinement. These future efforts may enhance the adaptability and generality of the proposed approach and address existing limitations.

One significant area for improvement involves the regeneration process. The current pipeline treats signal-to-API translation as a static task. Any modification to the signal or API specification, such as a renamed, deleted, or newly added signal, triggers regeneration of the entire set of mappings and endpoint implementations. This strategy might seemingly be robust, but is considered inefficient in dynamic engineering environments where specifications evolve incrementally. Future work on this topic could explore an incremental approach, where the system modifies only the relevant parts of an existing implementation based on detected changes. Investigating whether this targeted modification yields better efficiency or quality compared to complete regeneration would be valuable.

An important practical extension of this work would be transitioning from high-level scripting environments to real-world embedded deployment targets. The code gen-

erated in this thesis is intended to be used in Python-based environments. Python was chosen deliberately to focus on a higher level of abstraction and not worry about low-level issues such as manual memory management and writing insecure code. However, production automotive systems often require C or C++ code to run on resource-constrained embedded control units. Adapting the LLM-based synthesis pipeline to generate correct, optimized, and secure embedded code remains a significant challenge.

Further refinement of the prompt engineering process is also needed. The prompts used in this study were not subjected to systematic evaluation. Conducting ablation studies on the prompts used at various stages could reveal the extent to which prompt design constrains current system performance. Alongside prompt optimization, experimenting with different temperature settings for LLM outputs could be explored. While a zero temperature policy was used for reproducibility in this study, investigating higher temperatures might reveal benefits in generating more diverse and accurate solutions. A more direct application of reasoning language models for the primary code generation tasks could be examined to see if they lead to more precise code. Combining all of these would allow one to gain deeper insights as to whether limitations are related to prompt design or model capabilities.

Last but not least, expanding the core library learning capabilities offers another intriguing and promising direction. An ambitious extension would be to enable the system to produce APIs based solely on signal information, without explicit API property definitions from an OAS. Aligning with the vision of library learning, this would represent a more advanced level of abstraction and learning, where the system discovers potential, meaningful services directly from low-level data.

7 Conclusion

This thesis explored the feasibility of generating RESTful APIs through library learning using language models. The study focused on the specific challenge of automating SPAPI development in the automotive sector, as the current manual process is labor-intensive and involves multiple teams and iterative coordination. Employing the design science research methodology, the primary aim of the study was to develop and evaluate an artifact. Inspired by the manual development process, the study proposed SPAPI Coder, a three-level compound AI system that is capable of synthesizing functional API endpoint implementations using low-level CAN signals and high-level OpenAPI specifications.

Through a series of controlled experiments, we evaluated the effectiveness of several open and proprietary LLMs and embedding models across the three abstraction levels. Additionally, a secondary contribution was introduced: using LLMs as automated judges to assess the correctness and specification alignment of generated artifacts.

Our results demonstrate that, under well-structured few-shot prompting and with access to domain-specific context, state-of-the-art LLMs are capable of producing signal-to-property mappings and endpoint code that align closely with human-authored specifications. Furthermore, employing a strategy involving RAG seems to help with efficient program search and reduced program space. Using LLMs as judges for evaluations revealed their potential but also indicated that current setups or state-of-the-art models are not ready to completely replace human evaluation.

All in all, our work was a successful demonstration of how compound AI systems that follow the characteristics of library learning and program synthesis can automate complex and iterative software development workflows in industrial settings. However, the total potential of what LLMs can do as part of compound AI systems, and how we orchestrate different AI tools and frameworks to automate complex tasks, remains a vast area to venture into. This leaves many windows of opportunity open for the use of such systems, not only for similar cases that fall under the scope of software workflow automation, but also AI in Software Engineering.

Bibliography

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Stefan Aust. Vehicle api and service catalog for next generation mobility. In *2022 25th International Symposium on Wireless Personal Multimedia Communications (WPMC)*, pages 418–423, 2022.

[3] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs, 2017.

[5] David R. Barstow. A perspective on automatic programming. *AI Magazine*, 5(1):5, Mar. 1984.

[6] Nicole Beaulieu, Sergiu M Dascalu, and Emily Hand. Api-first design: A survey of the state of academia and industry. In *ITNG 2022 19th International Conference on Information Technology-New Generations*, pages 73–79. Springer, 2022.

[7] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 610–623, 2021.

[8] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022.

[9] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), January 2023.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] Saurabh Chauhan, Zeeshan Rasheed, Abdul Malik Sami, Zheying Zhang, Jussi Rasku, Kai-Kristian Kemell, and Pekka Abrahamsson. Llm-generated microservice implementations from restful api definitions, 2025.

[12] Nachum Dershowitz and Zohar Manna. On automating structured programming. *Proc. Colloques IRIA on Proving and Improving Programs*, pages 167–193, 1975.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[14] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[15] Mario Dudjak and Goran Martinović. An api-first methodology for designing a microservice-based backend as a service platform. *Information Technology and Control*, 49(2):206–223, 2020.

[16] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[17] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020.

[18] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. Publication, University of California, Irvine, 2000.

[19] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR,

2023.

[20] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2, 2023.

[21] Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code, 2024.

[22] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[23] S. Gulwani, O. Polozov, and R. Singh. *Program Synthesis*. Foundations and Trends® in Programming Languages Series. Now Publishers, 2017.

[24] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, page 13–24, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):90–99, October 2015.

[26] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[27] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

[28] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.

[29] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. A survey on large language models: Applications, challenges, limitations, and practical usage. *Authorea Preprints*, 3, 2023.

[30] Florian Haupt, Frank Leymann, Anton Scherer, and Karolina Vukojevic-

Haupt. A framework for the structural analysis of rest apis. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 55–58, 2017.

[31] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[32] Geoffrey E. Hinton, Peter Dayan, Brendan J. Frey, and Radford M. Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

[33] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), December 2024.

[34] Yaojie Hu, Qiang Zhou, Qihong Chen, Xiaopeng Li, Linbo Liu, Dejiao Zhang, Amit Kachroo, Talha Oz, and Omer Tripp. Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks, 2025.

[35] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, 43(2), January 2025.

[36] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[37] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[38] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[39] Shugang Jiang. Vehicle e/e architecture and its adaptation to new technical trends. Technical report, SAE Technical Paper, 2019.

[40] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[41] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

[42] Karl Henrik Johansson, Martin Törngren, and Lars Nielsen. *Vehicle Applications of Controller Area Network*, pages 741–765. Birkhäuser Boston, Boston, MA, 2005.

[43] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[44] Gordon S. Novak Jr. Cs 394p: Automatic programming. Accessed: 25-03-2025.

[45] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *EMNLP (1)*, pages 6769–6781, 2020.

[46] Christoph Kreitz. Program synthesis. In *Automated Deduction—A Basis for Applications: Volume III Applications*, pages 105–134. Springer, 1998.

[47] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[48] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[49] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*, 2023.

[50] Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. Preference leakage: A contamination problem in llm-as-a-judge. *arXiv preprint arXiv:2502.01534*, 2025.

[51] Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*, 2024.

[52] Jiatong Li, Rui Li, and Qi Liu. Beyond static datasets: A deep interaction approach to llm evaluation. *arXiv preprint arXiv:2309.04369*, 2023.

[53] Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models. *arXiv preprint arXiv:2407.02883*, 2024.

[54] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 280–301, Cham, 2024.

Springer Nature Switzerland.

[55] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[56] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, 55(9):1–35, 2023.

[57] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[58] Anders Magnusson, Leo Laine, and Johan Lindberg. Rethink ee architecture in automotive to facilitate automation, connectivity, and electro mobility. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 65–74, New York, NY, USA, 2018. Association for Computing Machinery.

[59] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[60] Blanca Martínez de Aragón, Jesus Alonso-Zarate, and Andres Laya. How connectivity is transforming the automotive ecosystem. *Internet Technology Letters*, 1(1):e14, 2018.

[61] Multi-Linguality Multi-Functionality Multi-Granularity. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. 2024.

[62] Rodrigo Nogueira and Kyunghyun Cho. Passage re-ranking with bert. *arXiv preprint arXiv:1901.04085*, 2019.

[63] Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy Lin. Multi-stage document ranking with bert. *arXiv preprint arXiv:1910.14424*, 2019.

[64] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration, 2021.

[65] OpenAI. New and improved embedding model, December 2022. `https://openai.com/index/new-and-improved-embedding-model`.

[66] OpenAI. New embedding models and api updates, January 2024. https://openai.com/index/new-embedding-models-and-api-updates.

[67] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[68] Dhasarathy Parthasarathy, Yinan Yu, and Earl T. Barr. Polymer: Development workflows as software, 2025.

[69] Max Peeperkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. Is temperature the creativity parameter of large language models? *arXiv preprint arXiv:2405.00492*, 2024.

[70] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[71] Matthew Renze. The effect of sampling temperature on problem solving in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7346–7356, 2024.

[72] Milos Rusic. Beyond peak data: The rise of compound ai systems. *Forbes Tech Council*, February 2025.

[73] Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Levis, and Matei Zaharia. Alto: An efficient network orchestrator for compound ai systems. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pages 117–125, 2024.

[74] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

[75] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[76] Shreya Shankar, JD Zamfirescu-Pereira, Björn Hartmann, Aditya Parameswaran, and Ian Arawjo. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–14, 2024.

[77] Douglas R Smith. A problem reduction approach to program synthesis. In *IJCAI*, pages 32–36, 1983.

[78] Armando Solar-Lezama. *Program synthesis by sketching.* PhD thesis, USA, 2008. AAI3353225.

[79] Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A Smith, Luke Zettlemoyer, and Tao Yu. One embedder, any task: Instruction-finetuned text embeddings. *arXiv preprint arXiv:2212.09741*, 2022.

[80] Harihara Subramanian and Pethuru Raj. *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs.* Packt Publishing Ltd, 2019.

[81] Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2023.

[82] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[83] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[84] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[85] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[86] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[87] Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *arXiv preprint arXiv:2502.06193*, 2025.

[88] Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. Automating a complete software test process using llms: An automotive case study, 2025.

[89] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[90] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*, 2024.

[91] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[92] Martin Weyssow, Aton Kamanda, Xin Zhou, and Houari Sahraoui. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *arXiv preprint arXiv:2403.09032*, 2024.

[93] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[94] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[95] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[96] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound ai systems. `https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/`, 2024.

[97] Janis Zenkner, Tobias Sesterhenn, and Christian Bartelt. Shedding light in task decomposition in program synthesis: The driving force of the synthesizer model, 2025.

[98] Janis Zenkner, Tobias Sesterhenn, and Christian Bartelt. Shedding light in task decomposition in program synthesis: The driving force of the synthesizer model. *arXiv preprint arXiv:2503.08738*, 2025.

[99] Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. Codejudge-eval: Can large language models be good judges in code understanding? *arXiv preprint arXiv:2408.10718*, 2024.

[100] Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

A

Appendix

A.1 Example OpenAPI specification

```
swagger: "2.0"
info:
  description: Set and Get cabin climate settings
schemes:
- http

paths:
  /climate:
    get:
      description: Retrieves cabin climate settings
      produces:
      - application/CabinClimate+json
      responses:
        200:
          description: Successful response
          schema:
              $ref: "#/definitions/ClimateObject"
        400:
          description: Bad Request
          schema:
            $ref: "#/definitions/ErrorInfoObject"
        500:
          description: Internal Server Error

    put:
      description: Update cabin climate settings
      consumes:
      - application/CabinClimate+json
      produces:
      - application/CabinClimate+json
      parameters:
```

```yaml
      - name: climate
        in: body
        required: true
        schema:
          $ref: "#/definitions/CabinClimateObject"
      responses:
        200:
          description: OK
          schema:
            $ref: "#/definitions/CabinClimateObject"
        400:
          description: Bad Request
          schema:
            $ref: "#/definitions/ErrorInfoObject"
        500:
          description: Internal Server Error

definitions:
  ErrorInfoObject:
    type: object
    description: Representation of an error
    properties:
      statuscode:
        type: integer
        description: repetition of HTTP status code
        example: 429
      message:
        type: string
        description: error message parsable by human
        example: Too many requests

  CabinClimateObject:
    type: object
    description: Settings object
    properties:
      acMode:
        type: string
        enum: ["ON", "OFF"]
        description: Defines which mode the AC is set to.
      fanLevel:
        type: string
        enum: ["LOW", "HIGH"]
        description: Defines which level the AC fan is set to
```

A.2 Example CAN signal information

```
{
  "name": "ACMode",
  "description": "This signal contains the information of the
  ↪  user's choice of AC mode",
  "min_val": null,
  "max_val": null,
  "unit": null,
  "resolution": 1,
  "enum_values": {
    "ACMode_Off": 1,
    "ACMode_On": 0
  }
}
```

A.3 Example Level 3 code

```
from app.utils import CANBus
from enum import Enum

canbus = CANBus()

class ACMode(Enum):
    ACMode_On = 1
    ACMode_Off = 0

def read_air_conditioning_mode_status():
    """
    Reads the ACMode signal from the CAN bus and returns the
    ↪  corresponding enum value.

    Returns:
        ACMode: The current air conditioning mode as an enum.
    """
    ac_state = canbus.read("ACMode")
    if enum_value not in [e.value for e in ACMode]:
        raise ValueError(f"Invalid enum value {enum_value} for
        ↪  {can_signal}")
    return ACMode(ac_state)
```

A.4 Example Level 2 code

```
from app.signals.read_ac_mode import
↪ read_air_conditioning_mode_status, ACMode

def get_acMode():
    """
    Retrieves the current air conditioning mode.

    Returns:
        str: The current AC mode, either "ON" or "OFF".

    Raises:
        ValueError: If the mode status is not valid or cannot be
        ↪ retrieved.
    """
    try:
        ac_mode_status = read_air_conditioning_mode_status()
        if ac_mode_status == ACMode.ACMode_On:
            return "ON"
        elif ac_mode_status == ACMode.ACMode_Off:
            return "OFF"
        else:
            raise ValueError(f"Unexpected AC mode status:
            ↪ {mode_status}")
    except ValueError as e:
        raise ValueError(f"Failed to retrieve AC mode: {str(e)}")
```

A.5 Example Level 1 code

```
from typing import Literal
from app.properties import get_acMode, put_acMode
from app.objects import CabinClimateObject, ErrorInfoObject
from fastapi import APIRouter, HTTPException

router = APIRouter(tags=["CabinClimate"])

@router.get("/climate", tags=["CabinClimate"])
def get_cabinclimate():
    try:
        ac_mode = get_acMode()
        fan_level = get_fanLevel()
        return CabinClimateObject(
            acMode=ac_mode,
            fanLevel=fan_level
        )
    except ValueError as e:
        raise HTTPException(
            status_code=400,
            detail=ErrorInfoObject(message=str(e)).model_dump()
        )
    except Exception:
        raise HTTPException(status_code=500)

@router.put("/climate", tags=["CabinClimate"])
def put_cabinclimate(body: CabinClimateObject):
    try:
        if body.acMode is not None:
            put_acMode(body.acMode)

        if body.fanLevel is not None:
            put_fanLevel(body.fanLevel)

        return body
    except ValueError as e:
        raise HTTPException(
            status_code=400,
            detail=ErrorInfoObject(message=str(e)).model_dump()
        )
    except Exception:
        raise HTTPException(status_code=500)
```

A.6 Prompt for signal interface synthesis

```
class CANSignalFunctionWriter(dspy.Signature):
    """
    The goal is to write a Python module that reads/writes the
    ↪  given CAN Signal from/to the CAN Bus.

    Some CAN Signals have CAN values which are of the type enum.
    ↪  The enum classes should always and strictly be the CAN
    ↪  Signal's name unmodified. Do not add any suffix or prefix.
    If no enum values are given, just read/write the signal. Do
    ↪  not write the enum classes if the enum values are null.

    The function names:
    - should be in such a way that it describes the CAN Signal.
    - should be specific to the CAN Signal's name but it should
    ↪  not contain the signal's name itself.
    - can be long so that it encapsulates all granular and
    ↪  necessary details from the signal name like the
    ↪  abbreviations, numbers and days if present.
    - must always start with 'read_' or 'write_' and be lower
    ↪  case.

    For 'read', the function should read the CAN Signal and return
    ↪  its corresponding enum or CAN Value.
    For 'write', the function gets the CAN enum or CAN value as an
    ↪  argument and should write the corresponding enum value or
    ↪  any other value for the signal.

    Write the functions only for the given signal type. All
    ↪  imports should be on top.
    Do not change or miss any of the CAN values or the CAN
    ↪  Signals' name. The CAN values in the Enum should be the
    ↪  same as they are given.
    You must write docstrings. The written functions should always
    ↪  check the min, max of each CAN Signal if given.
    """

    CAN_Signal: dict = dspy.InputField(
        desc="A dictionary with the CAN Signal information."
    )
    code: str = dspy.OutputField(
        desc="Python modules written with the given CAN Signal
         ↪  information."
    )
```

A.7 Prompt for property synthesis

```
class APIPropertyWriter(dspy.Signature):
    """
    The goal is to write a Python module for an API property using
    ↪ one or more of the given functions.
    Use only the functions most relevant to fulfilling the
    ↪ specification of the API property.

    First, import the relevant functions:
        - Import the signal functions from app.signals
        - Import the Enums from the same file if applicable.
        - The CAN Signal name is the key in the given dictionary.

    Make sure that the written code for the API property adheres
    ↪ to the type, description and any other of the information
    ↪ provided.
    The function can use any of the given functions with the "AND"
    ↪ or "OR" logical operators as long as the functions are
    ↪ related to the API Property.
    The function can also use different values from the same
    ↪ function with the "AND" or "OR" logical operators.

    Do not make up additional properties! If the property has
    ↪ subproperties (when it's referencing an object), make sure
    ↪ all subproperties are included and returned.
    Do not include the given functions' code in your final output,
    ↪ they should always be imported as stated.

    Handle errors raised by the signal function(s) by re-raising
    ↪ the errors with more appropriate API property related
    ↪ messages.
    """

    api_property: dict = dspy.InputField(
        desc="The API Property and its description for which the
         ↪ python module should be written."
    )
    functions: list[dict] = dspy.InputField(
        desc="Signal functions and enums that can be used by the API
         ↪ Property. Keys are the corresponding CAN signal names"
    )
    api_property_code: str = dspy.OutputField(desc="Implemented
    ↪ Python module for the API property.")
```

A.8 Prompt for mapping correctness judge

```
class MappingCorrectnessJudge(dspy.Signature):
    """
    Your task is to evaluate the correctness of the mappings
    ↪ between an API property (or combination of subproperties)
    ↪ and signal functions in Python code.

    You are able to check the signal function
    ↪ implementation/documentation from the RAG results.

    Evaluation steps:
    1. Identify the API property (or combination of subproperties)
    ↪ from the the specification and/or the code provided.
    2. Determine which signal function(s) each API property is
    ↪ mapped to.
    3. In the RAG results check what the mapped signal functions
    ↪ actually do.
    4. Confirm that the signal mappings accurately reflect the
    ↪ intended functionality of the API properties
    """

    api_property_specification: dict = dspy.InputField(
        desc="The specification of the API property"
    )
    rag_results: list[RAGResult] = dspy.InputField(
        desc="Results from the RAG containing signal function
         ↪ candidates"
    )
    code: str = dspy.InputField(desc="The implementation code of the
    ↪ API property")
    verdict: Literal["YES", "NO"] = dspy.OutputField(
        desc="The final judgement whether all mappings are correct
         ↪ or not"
    )
```

A.9 Prompt for specification alignment judge

```
class SpecificationAlignmentJudge(dspy.Signature):
    """
    Your task is to evaluate the specification alignment of an API
    ↪  property (or combination of subproperties) implementation.

    Evaluation steps:
    1. Identify the API property (or combination of subproperties)
    ↪  from the the specification and/or the code provided.
    2. Ensure that all API properties or subproperties (excluding
    ↪  "type") are mapped to AT LEAST one or more signal
    ↪  functions
        - If a property is left out, it means that the
        ↪  specification has not been fully followed
    3. Confirm that the API property function returns the correct
    ↪  type at all branches (excluding any exception raises) and
    ↪  all cases are covered
        - For instance, if the type is string and it's enum, the
        ↪  function must only return the viable enum strings and
        ↪  it should handle each scenario
    """

    api_property_specification: dict = dspy.InputField(
        desc="The specification of the API property"
    )
    code: str = dspy.InputField(desc="The implementation code of the
    ↪  API property")
    verdict: Literal["YES", "NO"] = dspy.OutputField(
        desc="The final judgement whether the implementation meets
        ↪  the specification"
    )
```

A.10 Prompt for API router synthesis

```
class APIRouterWriter(dspy.Signature):
    """
    The goal is to complete the Python implementation of a FastAPI
    ↪  router using information about API object(s) and the
    ↪  overall code structure (boilerplate).

    First, start with imports:
    - Import the given Pydantic models using the API name - from
    ↪  app.models.api_name import ExampleObject, ErrorInfoObject
    - Additionally, include ALL imports, given in the boilerplate
    ↪  code, in your final output
    - Do NOT use aliases

    Remove existing object related class definitions (Pydantic
    ↪  models) from your final output, they must be imported as
    ↪  previously stated.
    Only fill in the given router functions.
    Do NOT write additional functions (endpoints).
    Do NOT write/add the API properties functions and enums code
    ↪  in your final output.

    If method is PUT and the property is "readOnly", you do not
    ↪  have access to the put function of the property. Do NOT
    ↪  make it up.

    For PUT method, the response model should be the body itself.

    Handle all specific errors (e.g., ValueError) raised by the
    ↪  API property functions by raising HTTPException with the
    ↪  correct status code detail as populated ErrorInfoObject.
    """

    api_name: str = dspy.InputField(desc="The name of the API.")
    api_object_info: dict = dspy.InputField(
        desc="Information about the API Object and its properties."
    )
    api_code_structure: str = dspy.InputField(
        desc="The Pydantic models and code structure for the API
        ↪  router."
    )
    api_router_code: str = dspy.OutputField(
        desc="The code for the API router implementation"
    )
```

A.11 Example Level 3 test case

```
import pytest
from app.signals.read_ac_mode import
↪   read_air_conditioning_mode_status


@pytest.fixture
def signal_name():
    return "ACMode"


@pytest.fixture
def valid_enums():
    valid_enums = {
        'ACMode_Off': 1,
        'ACMode_On': 0
    }
    return valid_enums


@pytest.fixture
def valid_enum_values(valid_enums):
    valid_enum_values = [
        test_enum_values for test_enum_name,
        test_enum_values in valid_enums.items()]
    return valid_enum_values


def test_correct_signal_name(signal_name):
    can_return = read_air_conditioning_mode_status()
    assert can_return.__class__.__name__ == signal_name


def test_valid_enum(valid_enum_values):
    can_return = read_air_conditioning_mode_status()
    can_return_enum_value = can_return.value
    assert can_return_enum_value in valid_enum_values
```

A.12 Example Level 1 test case

```
from app.utils.vcan import CANBus
import requests
import pytest
import time

def test_get_climates():
    vcan = CANBus(keep_state=True)
    vcan.set_signal(signal_name='ACMode', signal_value=1)
    vcan.set_signal(signal_name='AutoFanLevel', signal_value=0)

    response = requests.get(url='http://localhost:9999/climate')

    response_body = response.json()
    assert response.status_code == 200
    assert response_body['acMode'] == 'OFF'
    assert response_body['autoFanLevel'] == 'HIGH'
```