

Development Workflow Generation Methodology Applied to a Propulsion Supervisory Controller for
Battery Electric Vehicles

by

Henri Rofiq

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Applied Science

in

Mechanical and Mechatronics Engineering

Waterloo, Ontario, Canada, 2025

© Henri Rofiq 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The increasing integration of software in modern vehicles has transformed the automotive industry, enabling advanced functionalities across the domains of safety, performance and user experience. However, the design and development of vehicle control systems is a complex process that requires familiarity with specialized tools and validation practices. These skills are typically not taught during university and thus, this thesis presents a comprehensive methodology for generating and implementing a control logic development workflow. The application of this methodology is demonstrated through its successful application to the design of a Propulsion Supervisory Controller (PSC) for deployment to a Cadillac LYRIQ, developed as part of the EcoCAR EV Challenge (EVC).

The proposed workflow provides a structured approach to tackle software tool and hardware selection, requirements generation, software design principles, testing strategies and codebase maintenance considerations. Application of this workflow results in the generation of the UWAFt controls development methodology which uses the MathWorks (MATLAB/Simulink) toolchain and Speedgoat hardware, where the team developed software that was a “pipes and filters”, layered and component-based control architecture. UWAFt employed Agile-hybrid principles for the comprehensive development of requirements which originate from supplier documentation, team goals as well as safety analyses. Finally, software was integrated using version control via Git and emphasized comprehensive verification which includes extensive “-in-the-loop” (XIL) testing.

Application of this methodology enabled UWAFt to achieve consistent and high-quality software development under resource constraints, leading to successful deployment and validation of vehicle control features such as torque management and directional control. Furthermore, the generated software also led to success at year-end competitions where the PCM team was able to successfully achieve a 3rd place finish. Beyond technical outcomes, the workflow improved collaboration, documentation and onboarding within the student team, bridging the gap between academic learning and industry-standard experience.

An assessment of limitations and areas for future improvement is presented, including enhanced CI/CD automation, cross-project integration and adaptation of the workflow for internal combustion architectures. Overall, this research contributes a modular and educationally valuable framework that can be adopted by student design teams and research groups to produce reliable automotive control software.

Acknowledgements

I'd like to start by acknowledging the dedication of Dr. Roydon Fraser to the success of UWAFI and his tireless commitment to enhancing the academic experience of students at the University of Waterloo through experiential learning. His 30-year tenure with UWAFI has not only directly sustained the team through all the highs and lows, but also been a guiding light that has inspired myself and others to pursue challenges within the automotive industry and beyond. I am eternally grateful for his guidance and support throughout my time as a master's student and I hope that others like me will be able to experience this unique opportunity for many years to come.

I'd like to also acknowledge all the team members that I've worked with on UWAFI over the last few years. Ranging from graduate leads to undergraduate volunteers, your enthusiasm for collaboration and working on a unique project, which became stressful at times, has encouraged me to push my own boundaries and provided relief that the team will be left in good hands.

I'd like to also express my gratitude to all the EcoCAR organizers and mentors from GM, Argonne National Laboratory, MathWorks and the US Department of Energy for their continued support in managing this exceptionally unique opportunity as well as allowing me to participate within it. Their commitment to the next generation of automotive engineers has been truly inspirational.

Finally, I'd like to thank John Francis Marrone for informing me of UWAFI and the EcoCAR competition and encouraging me to pursue this opportunity; it has truly been one of my best career decisions thus far.

Dedication

This thesis is dedicated to my siblings and my family, specifically my mother, Lucky, who has always pushed me to be the best possible version of myself and aim for success in whatever endeavour I choose to embark on.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	x
List of Tables.....	xii
List of Abbreviations.....	xiii
Chapter 1 Introduction.....	1
1.1 Background	1
1.2 Problem Identification	3
1.3 Objective and Thesis Outline	4
Chapter 2 Literature Review	6
2.1 Software Architectures	6
2.1.1 Architectural Views.....	6
2.1.2 Architectural Styles	7
2.1.3 Control Strategy Safety	10
2.2 Production Vehicle Control Systems.....	12
2.3 Student Design Team Control Methodologies	14
Chapter 3 Development Workflow Generation Process for the Creation of a Propulsion Supervisory Controller.....	17
3.1 Tool / Hardware Selection and Project Management.....	17
3.1.1 Software Tool Chain Selection.....	18
3.1.2 Compute-Hardware Selection / Communication Protocols.....	22

3.1.3 Version Control Methodology	26
3.1.4 Working and Planning Methodology	28
3.1.5 Cost.....	30
3.2 Requirements Generation	31
3.2.1 Safety, Reliability and Standards Compliance	31
3.2.2 Supplier and Supporting Documentation	32
3.2.3 High Level Product Goals	33
3.2.4 Competitor Analysis.....	33
3.2.5 Diagnostics and Fault Mitigation	34
3.2.6 Requirements Tracking	35
3.2.7 Requirement Format.....	35
3.3 Development Methodology	36
3.3.1 System Architecting and Partitioning.....	36
3.3.2 Task Generation.....	38
3.3.3 Deployment and Integration Environment	39
3.3.4 Communication and Networking Considerations.....	40
3.3.5 Inspiration Via Industry Standards	40
3.4 Testing and Validation Framework	41
3.4.1 Types of Testing	41
3.4.2 Testing Environments.....	43
3.4.3 Test Tracking.....	45
3.4.4 CI/CD	45
3.4.5 Additional Considerations and Best Practices	46
3.5 Codebase Maintenance and Monitoring / Debugging	47

3.5.1 Data Logging, Diagnostics and Visualization	47
3.5.2 Refactoring and Environment Updates.....	48
3.5.3 Standard Operating Procedures and Safety Guidelines	49
Chapter 4 UWAFI Propulsion Supervisory Controller	50
4.1 Selected Toolchains.....	50
4.1.1 Software Tool Selection	50
4.1.2 Hardware and Communication Selection	51
4.1.3 Version Control Methodology Selection.....	53
4.1.4 Working and Planning Methodology Selection.....	54
4.2 Generated Requirements and Justification	55
4.3 Controller Recap and Development Principles	58
4.3.1 UWAFI PCM Controller Layout and Structure	58
4.3.2 Task Generation.....	66
4.3.3 Integration Methodology and Deployment Schedule	68
4.4 Testing Framework.....	70
4.4.1 Virtual Testing / Simulation	70
4.4.2 Non-Virtual Testing.....	81
4.4.3 Planning and Tracking Tests	83
4.4.4 CI/CD and Test Automation.....	85
4.5 Maintenance Methodology	85
4.5.1 Visualization Dashboards.....	85
4.5.2 Tool and Code Maintenance.....	88
4.5.3 Documentation	89
Chapter 5 Methodology Application Walkthrough, Validation and Outcomes	91

5.1 Feature Example.....	91
5.1.1 Feature Explanation.....	91
5.1.2 Requirement Sources, Generation and Task Creation.....	94
5.1.3 Logic Overview	97
5.1.4 Software Integration and Version Control.....	99
5.1.5 Testing and Validation	100
5.1.6 Maintenance, Debugging and Documentation	107
5.2 Validation, Advantages and Outcomes	108
5.2.1 Competition Outcomes	108
5.2.2 Team Outcomes	109
5.3 Future Endeavors.....	110
Chapter 6 Conclusion	112
6.1 Summary of Work	112
6.2 Current Limitations and Future Work	113
References	115

List of Figures

Figure 1 : Camera-Based Active Safety Feature Flow Chart [4].....	2
Figure 2 : Responses to whether involvement with UWAFt has led to better co-op opportunities (N = 31) [6].....	3
Figure 3 : Functional View Example [9].....	7
Figure 4 : "Pipes and Filters" Architectural Style Example [9].....	8
Figure 5 : Component-based Architectural Style Example [9]	9
Figure 6 : Layered Architectural Style Example [9]	9
Figure 7 : Powertrain of a Hybrid Electric Vehicle [12]	13
Figure 8 : State machine Example Applied to Autonomous Vehicle Decision Making [17].....	18
Figure 9 : Jira Ticket ID	54
Figure 10 : UWAFt PCM Controller High Level Overview.....	58
Figure 11 : Subsystem Separation and Functionality Scheme	59
Figure 12 : State Estimation and Fault Detection Subsystems.....	60
Figure 13 : State Estimation Subsystem Functionality.....	61
Figure 14 : Vehicle State Controller Functionality	62
Figure 15 : Torque Split Subsystem	63
Figure 16 : Torque Split Logic	63
Figure 17 : Final Component Execution Subsystem	64
Figure 18 : Torque Limiting Logic within the Component Execution Subsystem.....	64
Figure 19 : Communications Layer Subsystem.....	65
Figure 20 : Example Sprint Tasks in Progress; with related Epic, Current Status, Deadline, Priority and Assignee.....	67
Figure 21 : UWAFt Gantt Chart.....	67
Figure 22 : Battery Power Limit Sample Requirements.....	68
Figure 23 : RTM Requirement Implementation and Validation Status.....	69
Figure 24 : Simulink Test Harness Visualization.....	71
Figure 25 : UWAFt Control Logic Test Harness	71
Figure 26 : Simulink Test Manager Home View	72
Figure 27 : Example Temporal Assessments for Pedal Map Test Case	73
Figure 28 : MATLAB CAN Data Import Script	74

Figure 29 : Testing Data Source / Generation Within Simulink Test Harness.....	74
Figure 30 : Sample Accelerator Pedal Data Trace Used Within Testing	75
Figure 31 : MIL Testing Report Generation.....	76
Figure 32 : Speedgoat Baseline M Controller Running Control Logic for HIL Testing.....	76
Figure 33 : Speedgoat Performance Unit Injecting Open Loop Testing Data for HIL Testing	77
Figure 34 : UWAFTHIL Setup with Speedgoat Unit Communication	77
Figure 35 : HIL Testing Data Injection Model for Speedgoat Performance Unit	78
Figure 36 : Speedgoat Baseline M Controller Modified HIL Testing Harness	79
Figure 37 : Modified HIL Test Case and Callback Script.....	80
Figure 38 : Waterloo Region Emergency Services Training and Research Centre Track	82
Figure 39 : University of Waterloo Parking Lot C Testing Area	82
Figure 40 : Test Plan Management.....	84
Figure 41 : Data Log Info Tracker Table	84
Figure 42 : UWAFTHIL Visualization Dashboard.....	86
Figure 43 : CANalyzer Configuration for Full Vehicle Visualization (With an Accompanied Depiction of a Brake Percentage Trace).....	88
Figure 44 : Initial Pedal Map Subsystems.....	93
Figure 45 : Final Torque Request Saturation.....	93
Figure 46 : Torque Request Generation Map	94
Figure 47 : Sample Tasks for Requirement PSC-PWR-7.1.....	96
Figure 48 : Detailed Task Description.....	97
Figure 49 : Torque Command Directional Control Logic.....	99
Figure 50 : Directional Control Temporal Assessments	101
Figure 51 : Passing Visualization of the Temporal Assessment.....	101
Figure 52 : Failing Visualization of the Temporal Assessment	102
Figure 53 : Relevant Injected CAN Test Data.....	103
Figure 54 : Drive Gear Logic Response	104
Figure 55 : Reverse Gear Logic Response	105
Figure 56 : Preliminary VIL Testing Setup.....	106

List of Tables

Table 1 : Feature Adoption Rate Through Time by # of Automakers [1].....	2
Table 2 : Examples of SCADA Weaknesses and Recommendations [10].....	11
Table 3 : FSAE Control Design Comparison with EcoCAR Considerations.....	16
Table 4 : Software Tool Selection Recap	21
Table 5 : Hardware Tool Selection Recap.....	24
Table 6 : Communication Protocol Selection Recap.....	26
Table 7 : Version Control Selection Recap	28
Table 8 : Project Management and Planning Selection Methodology	30
Table 9 : UWAFT Vehicle Technical Specifications	56
Table 10 : Sample Requirements for UWAFT PCM Controller	57

List of Abbreviations

ADAS	Advanced Driver Assistance Systems
ADC	Analog to Digital Converter
AUTOSAR	Automotive Open System Architecture
AWD	All-Wheel Drive
CAN	Controller Area Network
CAPL	Communication Access Programming Language
CAV	Connected and automated vehicles
CI/CD	Continuous Integration and Development
DAC	Digital to Analog Converter
DIL	Driver in the Loop
DOF	Degrees of Freedom
ECU	Electronic Control Unit
EMI	Electromagnetic Interference
ETRS	Electronic Transmission and Range Selection
EV	Electric Vehicle
EVC	Electric Vehicle Challenge
FD	Flexible Data-Rate
FIA	Federation Internationale de l'Automobile
FSAE	Formula Society of Automotive Engineers
GM	General Motors
GUI	Graphical User Interface
HAZOP	Hazard and Operability
HIL	Hardware in the Loop
HV	High Voltage
I/O	Input/Output
I2C	Inter-Integrated Circuit
ICE	Internal Combustion Engine
IDE	Integrated Development Environment

IED	Intelligent Electronic Device
ISO	International Organization for Standardization
LIN	Local Interconnect Network
LPSO	LYRIQ Propulsion System Overview
MBD	Model-Based Design
MBSE	Model-Based Systems Engineering
MIL	Model in the Loop
NVH	Noise, Vibration and Harshness
OBD	On Board Diagnostics
OEM	Original Equipment Manufacturer
ORSE	On Road Safety Event
PCB	Printed Circuit Board
PCM	Propulsion Controls and Modelling
PHA	Preliminary Hazard Analysis
PIL	Processor in the Loop
PLC	Programmable Logic Controller
PRINCE2	PROjects IN Controlled Environments
PSC	Propulsion Supervisory Controller
ROS	Robot Operating System
RTM	Requirements Traceability Matrix
RTU	Remote Terminal Unit
RWD	Rear-Wheel Drive
SAE	Society of Automotive Engineers
SCADA	Supervisory Control and Data Acquisition
SDC	Sedra Student Design Centre
SDI	Systems Design and Integration
SDV	Software Defined Vehicle
SIL	Software in the Loop
SOP	Standard Operating Procedure
SOTIF	Safety of the Intended Function

SPI	Serial Peripheral Interface
STPA	Systems Theoretic Process Analysis
SVN	Subversion
TC	Traction Control
TSN	Time Sensitive Networking
TV	Torque Vectoring
UWAFT	University of Waterloo Alternative Fuels Team
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
VCS	Version Control System
VIL	Vehicle in the Loop
VTI	Vehicle Technical Inspection
WOT	Wide Open Throttle
WSC	Wheel Slip Control
XIL	“-In-The-Loop”

Chapter 1

Introduction

1.1 Background

Software development and testing has become an increasingly integral part of vehicle development ever since the inception of the 1957 “Bendix Electronic Fuel Injection System”, the world's first automotive electronic control system. This Electronic Control Unit (ECU) was responsible for managing the fuel delivery to the 1958 Chrysler 300D and 1958 Chrysler Imperial with the goal of managing environmental emissions. Since then, the automotive industry has rapidly scaled up development efforts towards goals such as fleet electrification and improved safety systems with the aid of several ECUs. These computers are responsible for consolidating various vehicle functionalities, ranging from propulsion system control to occupant comfort and have become a fundamental aspect of the vehicle design lifecycle.

Having a holistic understanding of how these systems interact has become essential to the proper development of automobiles and industry-sweeping initiatives have been undertaken to ensure the formalization of the development process. This has been done with the goal of reducing iteration time and the ultimate goal of creating safer systems that minimize occupant hazards in regard to feature sets such as propulsion control or advance driver assistance systems (ADAS). Adoption of these “state-of-the-art” methodologies and functionalities has further been shown to rapidly be adopted by automotive companies through the years with developmental times for features decreasing from more than 50 years of development time to approximately 10 years; with safety features (typically ECU based and mission critical software) being adopted twice as fast as other vehicle features [1]. Table 1 recaps the exponential growth in feature adoption for several features by large automakers within an 8-year time span.

Table 1 : Feature Adoption Rate Through Time by # of Automakers [1]

Technology	'17	'18	'19	'20	'21	'22	'23	'24	'25
High Eff. Gearbox (Auto or DCT)	8	25	37	50	56	60	5	56	56
CEGR - Level 2 (27 bar BMEP) - Med Disp.	0	0	0	0	3	4	6	7	6
Aero Drag Reduction – Level 2	41	51	65	73	84	88	88	89	89
12V Micro-Hybrid (Stop-Start)	7	10	16	26	33	39	40	41	37

Furthermore, several safety measures implemented into vehicles with the increasing rollout of connected technologies, whether they be passive features such as fault detection or even active measures, are deployed entirely onto sophisticated software and hardware stacks. For example, vehicle diagnostics functionality, which are federally regulated and legally mandated within all vehicles sold in North America since 1996 [2]; with adherence to various Society of Automotive Engineers (SAE) standards such as the J1978 standard for the “Universal Interface for OBD II Scan” [3], are entirely software-based feature sets. Additionally, vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication allows for the advent of novel algorithms such as in Figure 1, where the use of visual data and detected objects originating from cameras installed within vehicles as well as roadside infrastructure to pre-emptively detect hazards, is presented [4].

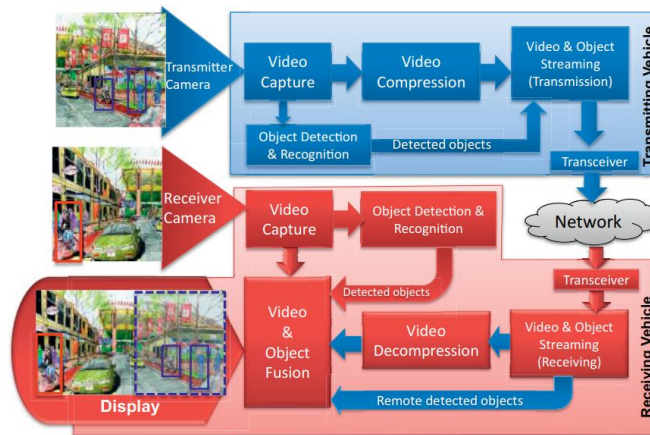


Figure 1 : Camera-Based Active Safety Feature Flow Chart [4]

At the University of Waterloo, co-operative education through experiential learning gained via internships or design team experience is integral to the engineering experience and culture of the university. The structure ensures that students graduate with up to two years of real-world experience, making them highly sought after by employers, with almost 12% percent of students at the university being primarily either from Software or Mechanical Engineering, as well as Computer Science [5]. Moreover, many students pursue or find opportunities in the automotive sector, whether through projects offered by the University of Waterloo Alternative Fuels Team (UWAF) or directly with industry partners. Roughly 58% of students that have joined UWAF have spoken to the experiences that they've gained and skillset they've learned to secure internship and job opportunities at prestigious companies in several different industries. This demonstrates a clear desire to enter into transportation and vehicle technologies [6]



Figure 2 : Responses to whether involvement with UWAF has led to better co-op opportunities (N = 31) [6].

1.2 Problem Identification

When embarking on a design challenge, either via the one of the design teams that are common on university campuses (for example, the University of Waterloo has 40 formally recognized design teams under the umbrella of the Sedra Student Design Centre (SDC) with approximately 5 being automotive focused [7]) or maneuvering responsibilities within their first career role, students regularly lack the skills to effectively tackle the work they are undertaking. They either need to

become proficient with the required skillset through co-op experiences (which are typically competitive to obtain and not necessarily facilitated by university administration) or need to learn on their own through available resources. Furthermore, software and controller design for student teams typically are behind the cutting edge and formal publications available online typically do not go into the intricate detail. This makes it such that it would be difficult for a student to be readily equipped to solve some of the developmental challenges that large scale Original Equipment Manufacturers (OEM) are tackling or even solving smaller scale problems, in accordance current industry methodologies and technologies.

To illustrate, engineering curriculums at the University of Waterloo, while they are globally renowned and cover a variety of topics to provide students with a wide breadth of knowledge, they typically might miss the mark on providing more applied experience. ME360 is the only controls course that is mandated by the Mechanical Engineering department and furthermore, course work typically revolves around theoretical understanding rather than practical application. [8]

Therefore, there exists a need to outline the process and considerations that go into the design and development of a controller and the associated workflow that will be employed to generate the software, detailing the development life cycle of the software stack starting with tool selection all the way to software maintenance. This proposed approach should also aim to be at a level of detail and complexity that allows university-level students to be able to apply the strategy to both academic and industrial challenges.

1.3 Objective and Thesis Outline

The work presented in this thesis aims to address the above problem statement through the presentation of the methodology that has been successfully applied to the conception of a development workflow for a PSC that has been deployed for UWAF's Cadillac LYRIQ (which has been donated to the team through their participation in the EVC). Moreover, the research presented in this thesis also will serve as a resource that student teams and industrial stakeholders alike can refer to when laying the groundwork for controller development.

Chapter 1 serves as an introduction to the design problem as well as providing context to the challenges that the following thesis aims to address.

Chapter 2 will then provide a review of the current state of controller software design, ranging from software architecture selection, as well as a highlight of both industry and academic trends with their associated benefits and drawbacks.

Chapter 3 will present a workflow for architecting the development of a vehicle controller as well as a comparative analysis of the various considerations to be made when tackling controller design, ranging from software tool chain selection, hardware selection and development/testing methodology.

Chapter 4 outlines the application of the aforementioned methodology in the generation of the UWAFt controls workflow, with an emphasis on highlighting competition considerations and constraints.

Chapter 5 recaps the results and outcomes of UWAFt's chosen methodology for controller design, running through a concrete example of the software development lifecycle from requirements generation all the way through to testing and vehicle deployment. Additionally, hurdles encountered by the team as well as competition success shall be provided with Chapter 6 serving as a recap of the work presented in the thesis and a resume of potential future work.

Chapter 2

Literature Review

This chapter aims to provide a comprehensive review of the current state of research surrounding automotive controller design with respect to several concepts relevant to the work of the thesis. It is grounded within the context of both production-level engineering and student design teams and serves to provide a critical examination of the strengths as well as the shortcomings of several bodies of work that a new engineer might potentially refer to when embarking on the challenge of developing an ECU.

2.1 Software Architectures

Staron performs a comprehensive overview of the various types of architecture views and styles that can be selected and employed within the use of control software development. An architectural view can be considered as a methodology for characterizing how an engineer should approach software design by identifying the most critical aspect of the software design process, including the “functional view”, “physical system view”, or “software view” [9]. Furthermore, architectural styles “[seek to] form principles of software design in the same way as building architecture shapes the style of a building”. In other terms, the methodology that drives the functionality of the software stack.

2.1.1 Architectural Views

The “functional view” captures what the system is supposed to do from a customer or end-user perspective. It describes the functions the vehicle must perform like braking, steering assistance, or infotainment without consideration of software or hardware selection. This view is critical during the initial requirements phase, serving as a bridge between customer needs and system capabilities. Functions are typically modeled in a hierarchical structure, enabling developers to decompose high-level vehicle features into manageable sub-functions, which later guide software and hardware selection and actualization.

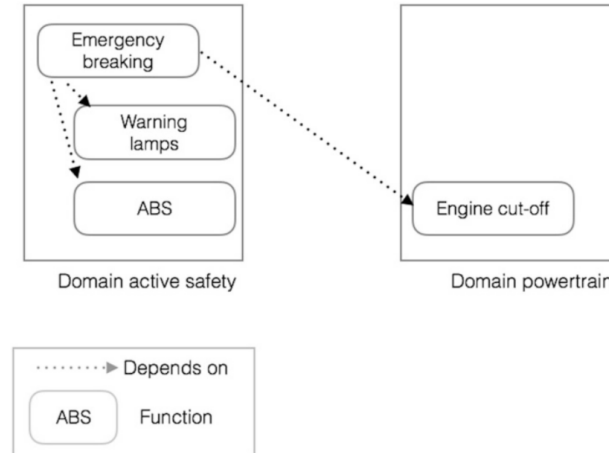


Figure 3 : Functional View Example [9]

The “logical view” maps abstract functions from the functional view into structured software architecture. The focus shifts to how functions are grouped into software components and how they communicate. Logical components are defined to include detail on the interfaces and interactions that will occur between these components. This view supports scalability and reusability, as logical components can be independent of a vehicle platform and reused across different projects (a truck or a sedan). It is also the main abstraction used in the Automotive Open System Architecture (AUTOSAR), forming the foundation for code generation and integration planning.

The “physical system view” reflects the actual deployment of software components onto hardware elements like ECUs, with the added consideration of communication buses and sensor measurements (as needed). It finalizes the architecture by resolving timing, bandwidth and safety constraints, mapping logical components to physical resources. This view is crucial for real-world integration, as it includes hardware-specific limitations, redundancy plans for safety and domain-specific constraints (e.g., powertrain vs. infotainment). Design choices here affect cost, robustness and compliance with safety standards, making this view central to production readiness

2.1.2 Architectural Styles

10 different styles are recapped by Staron, including “Event-driven” and “Service-oriented” styles; several of which originate from traditional software development principles and conventions however, extra care is taken to describe “Pipes and filters”, “Component-Based” and “Layered

architecture”, where each style offers specific benefits depending on the functional requirements, performance constraints and system modularity needs.

The “Pipes and filters” architectural style breaks the software into discrete steps, called *filters*, which are connected by *pipes* that pass data between them. Each filter performs a well-defined transformation on its input and sends its output downstream. This approach is ideal for processing streams of data where each stage is independent of the next, such as during sensor fusion for ADAS. The key advantages include simplicity, reusability of filters and the ability to parallelize stages. However, care should be taken to ensure that issues with latency are addressed prior to deployment in time-critical automotive applications due to the sequential nature of the data flow.

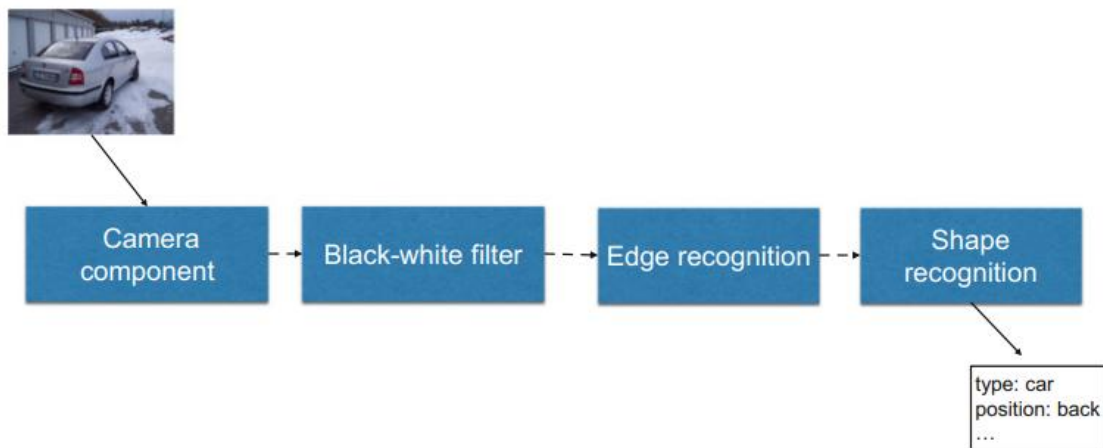


Figure 4 : "Pipes and Filters" Architectural Style Example [9]

In “Component-Based” architecture, the system is built from isolated, interchangeable components that use well-defined public interfaces and interact through well-specified contracts (i.e.: what the component CAN and CANNOT do in relation to adjacent components). This style promotes reuse and separation of concerns, enabling OEMs and suppliers to independently develop, verify and integrate components which are foundational for standards like AUTOSAR, which defines software components as deployable and reusable units. The challenge lies in maintaining interface stability and managing interdependencies, especially across teams, suppliers and vehicle programs.

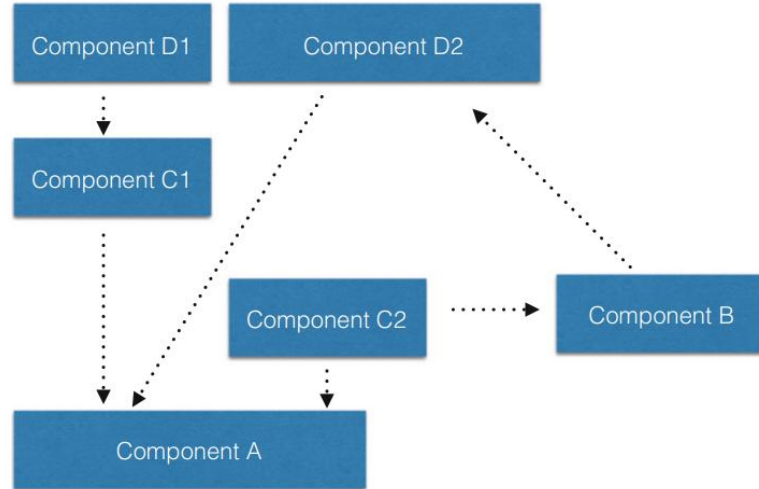


Figure 5 : Component-based Architectural Style Example [9]

“Layered architectures” organizes the system into horizontal layers, where each layer has a specific role (e.g., hardware abstraction/communications layer, logic etc...) and interacts only with adjacent layers. It enforces strict separation of concerns and aids in maintainability, testing and abstraction. As a consequence of the previous, the layered architecture aligns well with the AUTOSAR standard. While it brings clarity and modularity, it can introduce overhead and limit flexibility, especially when performance optimizations require breaking the layer boundaries.

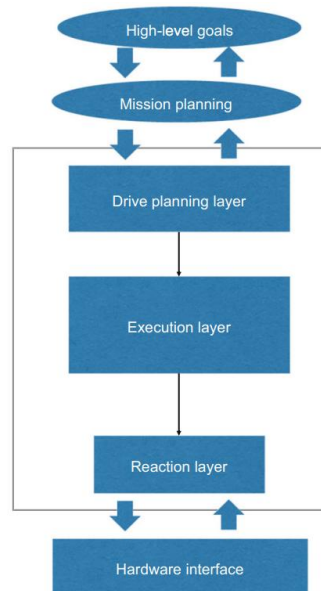


Figure 6 : Layered Architectural Style Example [9]

Staron clearly outlines the different views and architectures that a control's developer may employ during the development of their control logic; touching on "[how] there is no unified formalism describing a software architecture and that software architecture is a means of communication. It allows architects to describe the principles guiding the design of their system and discuss the implications of the principles on the components." [9] However, the lack of a guiding example or a tangible application of these abstract software frameworks has its downsides, namely the inability to ground a new developer into how these frameworks interplay with secondary considerations effecting a group of engineers. For example, how would the selection of an architecture scheme manifest itself in the form of working conventions that a set of engineers need to abide by? How would they synergize with project management principles such as the "Agile" methodology or "Predictive" vs. "Adaptive" planning strategies? The absence of the benefits and trade-offs of each of these strategies as well as a comparison to industry standards (with a contrast to teams of various headcount/size) fundamentally makes it so deciding amongst these frameworks is difficult.

2.1.3 Control Strategy Safety

Similarly to traditional software applications such as web development, cybersecurity and intellectual property management are critical factors that should be considered when architecting a development workflow for control software development, namely due to the mission criticality of the processes that are being controlled. Upadhyay and Sampalli performed an analysis of the main vulnerabilities that are in Supervisory Control and Data Acquisition (SCADA) systems and provides detailed security recommendations for each aspect of their architecture. SCADA is an industrial framework for control engineering where a "network typically includes a control server placed at the control center, communication links and one or more topographically dispersed field sites containing field devices. The SCADA sensors and actuators at field sites monitor the different attributes of electromechanical equipment continuously and send signals to field control devices like Programmable Logic Controller (PLC), Remote Terminal Unit (RTU) or Intelligent Electronic Device (IED)" [10] Upadhyay and Sampalli discuss how SCADA systems are now increasingly interconnected for remote access and operational efficiency and has introduced several weaknesses in the framework. They categorize SCADA risks into Product, Configuration, Network and Protocol risks where key vulnerabilities are attributed to things such as improper input validation, improper coding practices, weak authentication or improper communication protocols [10].

Table 2 : Examples of SCADA Weaknesses and Recommendations [10]

Vulnerability	Description	Recommendation / Fix
Poor Code Quality	Invalid function calls	The custom application should be implemented using security features; Code review should be perform during each iteration of testing; SCADA protocols should integrate integrity check and authentication.
Audit and accountabilities	Poor Logging practices	The network administrator should review logging files regularly to identify security incidents; The proper methodology should be implemented to trace all console related activities; Proper policies should be implemented for log storage, protection and review
Cryptographic Issues	Broken encryption techniques for sensitive information	SCADA control engineers and developers should properly research before incorporating encryption solution in the product; Security experts should update the team regularly on published vulnerabilities to keep patches up to date.

While the focus of the paper is on the control of industrial processes such as chemical or utilities control, the learnings can be carried over into automotive controller design where they can manifest themselves similarly to the recommendations compiled in Table 2 above. For example, establishing proper development conventions, implementing fault detection or data validation algorithms, or even traditional cybersecurity protocols may potentially address some of the concerns of the researchers. These considerations have become increasingly crucial since similarly to SCADA networks, vehicles are also becoming ever-presently connected to each other and the infrastructure around them.

2.2 Production Vehicle Control Systems

The shift to software defined vehicles (SDV) has always been driven by industry trends, the desire for OEMs to provide a refined experience for their customers and to effectively extract all the desired performance in the technology that is being integrated into the vehicles of today. Electrification has been the biggest trend within the automotive industry in recent years however, hybrid vehicles and namely parallel-hybrid systems, which predate electric vehicles (EV), also have had a resurgence within the 1st quarter of 2025 [11]. Researchers at Toyota have thus proposed a novel control system for a parallel hybrid vehicle which includes a turbocharged engine, All-wheel drive (AWD) topology with 2 clutches and a 6-speed automatic transmission, which were developed for installation into 2023 Toyota Crown and Lexus RX vehicle programs. Some of the challenges being tackled revolve around maintaining user experience and vehicle acceleration when switching between pure electric and hybrid modes, ensuring optimal acceleration is maintained throughout the most efficient operational bands and finally, maintaining vehicle traction during high acceleration events [12].

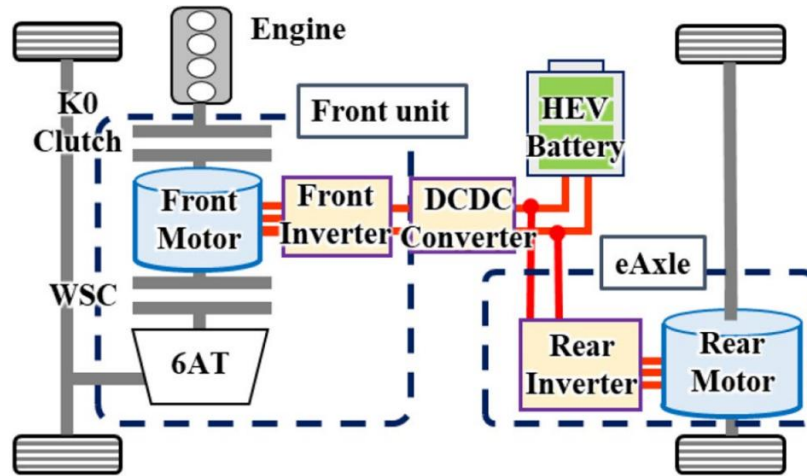


Figure 7 : Powertrain of a Hybrid Electric Vehicle [12]

Wheel slip control (WSC) is a feature of interest to the research presented in this thesis as it is a feature that is directly related to the development required within the context of the EVC. Takeichi et al. [12] recap the methodology for WSC by presenting the use of a feedforward controller that employs the use of motion equations to predict the driveline response as well as a feedback controller to improve robustness to disturbances, designed for dead-time in engine torque response. While flow charts have been presented to outline the controller layout for both controllers respectively, major considerations including controller performance targets such as settling time or percentage overshoot are omitted. Furthermore, tuning methodologies for selecting controller gains as well as toolchain selection criteria are excluded entirely; something that is typically acquired and improved with experience, which most student design teams lack entirely. Admittedly, the paper lacking in technical depth is likely due to intellectual property restrictions and the authors' need to protect proprietary control strategies that provide a competitive edge in the market.

Furthermore, the above paper additionally fails to address several key considerations and design challenges as detailed in Hasan and Irgen's "Electronic Control Unit Hardware Design Challenges for Software Defined Vehicle". [13] These researchers from General Motors (GM) explore how the shift toward SDVs is reshaping ECU hardware requirements. It highlights the need for scalable and modular ECU architectures that can support a broad range of vehicle models, trim levels and model years without excessive cost or complexity. The paper outlines challenges in designing generalized input/output interfaces, flexible power distribution, adaptable compute resources and robust networking while balancing manufacturability, packaging, environmental constraints and functional

safety. Examples such as analog input biasing strategies show the trade-offs between flexibility, cost and overdesign, ultimately advocating for careful partitioning of hardware capability and modular component population strategies to future-proofed designs. These are a subset of major considerations that OEMs such as GM typically call attention to during their software development processes, used within the very same LYRIQ that was donated to UWAF and ignored entirely during the presentation of the WSC feature by Takeichi et al.

2.3 Student Design Team Control Methodologies

Students at various universities across the world have also released technical writing recapping the work that has gone into various control design projects. Work presented in both a thesis from Politecnico di Torino [14] and an Eindhoven University of Technology paper [15] represent focused attempts by Formula Student teams to tackle advanced vehicle control problems, specifically torque vectoring and traction control, within the constraints of small, student run motorsport projects. The work of Fricano is a comprehensive academic design of a torque vectoring (TV) controller for a 4-in wheel motor Formula Society of Automotive Engineers (FSAE) electric car. It starts with a recap of foundational vehicle dynamics knowledge, before moving to controller architecture. The TV controller is built in MATLAB/Simulink and co-simulated with CarSim, comprising modules for yaw rate reference calculation (via a 2 degree of freedom (DOF) bicycle model), yaw moment computation, torque allocation, slip control and maximum torque limitation. Inputs come from realistic on car sensors (accelerometers, encoders, GPS) and outputs adjust each motor's torque in real time. The system is validated virtually through steady state and transient manoeuvres (constant steer, double lane change), showing improved yaw rate tracking and lateral acceleration over equal torque distribution. The thesis also openly lists the “weak points” of the design, such as dependence on accurate tyre models and sensor fidelity. Comparatively, the work of Loof et al. focuses on traction control (TC) for a rear-wheel-drive (RWD) electric FSAE car with a single central motor and high peak torque capability. The design begins with a 10-parameter nonlinear tyre model, a 3 DOF driveline model (including shaft flexibility, damping and wheel inertias) and a 7 DOF vehicle body model. The TC system has two layers comprising of a feedforward controller and a feedback controller which are used to estimate vehicle state to pre-emptively limit slip and then to subsequently correct any actual errors respectively.

Both papers demonstrate a notably high level of technical depth for FSAE teams, with each team applying physics-based modelling that is sufficiently accurate for control system design while remaining simplified enough for real-time execution. Their development processes follow a complete validation loop, progressing from simulation to on-car testing and ultimately achieving full hardware implementation in the case of Loof et al. Both teams also place importance on drivability, ensuring that peak performance improvements do not come at the expense of predictable and controllable vehicle behaviour. More importantly, the authors are transparent about the limitations of their approaches, openly acknowledging issues such as dependency on accurate tire models, sensor resolution limits and environmental assumptions that may not hold outside competition settings. For the narrow, high-performance context of FSAE competitions, these projects are well-scoped and highly relevant, serving as strong examples of targeted control system design within the resource and time constraints of student motorsport however, there exists a disconnect between the work that was presented and with the challenges faced with the production vehicle industry.

Namely, FSAE controllers are tuned for maximum transient performance and driver feedback in an open loop competition environment. EVC vehicles are production based, heavy, full system EVs that must deliver efficiency, safety and reliability over years of real world driving in all climates and road conditions. Furthermore, FSAE control logic does not typically abide by ISO 26262-style functional safety layers, extensive diagnostics or fault management strategies whereas this is a non-negotiable within the automotive industry. Additionally, the goals of production vehicles and more specifically, of EVC, emphasize sustained performance and system health over years, with the ultimate goal of deploying software to vehicles that can accommodate hardware that changes during development (such as different inverters, different Controller Area Network (CAN) protocols, or even different propulsion architectures). Therefore, the restricted scope of these FSAE-centered student publications undermines the holistic and well-rounded development workflows required of robust, industrial control design.

Table 3 : FSAE Control Design Comparison with EcoCAR Considerations

Aspect	FSAE Control Design	EcoCAR Control Design
Vehicle Platform	Lightweight, short-wheelbase, high power-to-weight race car (slick tyres, low mass).	Full-scale production EV/hybrid, heavy, road-legal, multi-domain vehicle.
Competition Objectives	Maximise transient performance, driver feel, lap	Optimise efficiency, ensure safety, meet emissions/energy targets, robust performance.
Control Scope	Single-function controllers (torque vectoring or traction control).	Multi-function supervisory control (torque split, regen blending, traction, energy management, thermal).
Integration	Low – fixed architecture, few systems to coordinate.	High – integration across propulsion, chassis, thermal, ADAS and diagnostics.
Testing	Short, closed-course competition events with limited variability.	Real-world driving cycles, varied climates, mixed road conditions.

Chapter 3

Development Workflow Generation Process for the Creation of a Propulsion Supervisory Controller

The following section of this thesis presents an overview of a workflow implemented by UWAFI for designing the methodology for use in the development of the PSC that was deployed onto the donated Cadillac LYRIQ. The proposed methodology seeks to address the shortcomings outlined in Chapter 2 by introducing a structured approach for developing a custom controls development workflow that is both scalable and adaptable to academic and industrial applications. It emphasizes alignment with current industry standards and trends, ensuring that the resulting workflow not only meets the needs of student design teams but also reflects the expectations of professional automotive development environments. A recap of the 5 major sequential considerations that were made is provided as well as a summary of their relevancy and importance when ensuring robust control design for automotive applications. Within several of these considerations exist diverse sub-options that each provide their respective positives and negatives and a selection must be made amongst these options, evaluating against factors such as cost, ease of use, support availability, etc... In this case, a comparative study will be performed including a summary of the advantages and disadvantages of selecting one option over another.

3.1 Tool / Hardware Selection and Project Management

Prior to undertaking the development of control software for automotive applications, planning out the various software tool chains, hardware platforms and project management approaches to be used is central to the success of the overall project. This is especially critical in the context of student design team control development where limited budgets and diverse technical backgrounds impose difficulties on the selection of an appropriate and effective working environment. By carefully balancing cost, flexibility and scalability, the team ensures that the project not only produces functioning control algorithms but also establishes a workflow that is sustainable for future work. The section below highlights the 4 main elements that must be settled upon prior to performing any technical work as well as the various options that an engineering team can choose between.

Furthermore, Simulink's code generation capability, through the Embedded Coder tool, is a powerful tool that allows for the transformation of block diagrams into traditional code. Automatic code generation reduces human error during model translation, accelerates prototyping and enables the same algorithm to be tested in simulation and on the hardware (which typically requires C/C++ code to function). This aligns well with industry trends, where model-based design (MBD) has become a popular method of robust control development, for instance, researchers have used this methodology in generating a platform for rapid power train control [18] and as reported by a survey of software engineers, 79% of respondents have looked positively towards Model-Based Systems Engineering (MBSE) and its utility [19]. Using the previously mentioned software tools with traditional scripting (in MATLAB or Python) also facilitates rapid data analysis, logging and automated test case execution, enhancing reproducibility.

For seamless integration of the Simulink environment with customizable and scalable hardware, 3rd part tools such as dSPACE TargetLink or the Speedgoat Real-Time Blockset allow for users to generate code for ease of deployment to rapid-prototyping controllers and hardware-in-the-loop (HIL) benches. Other MBD based tools such as LabVIEW from National Instruments provides an all-in-one solution where a custom graphical based language known as "G" (and in some ways, very similar to Simulink), can let a developer deploy software easily to National Instruments controllers without the need to worry about code generation or real-time considerations due to the high-level of integration of software and hardware [20]. In this case, you are constrained to performing all work within the LabVIEW environment as hardware cannot be modified or swapped if system requirements have changed.

3.1.1.2 Embedded C/C++ and Traditional Programming

If a user or engineering team decides to move away from the MBD philosophy and instead go down the path of traditional software development tools, there exist several languages that are readily available and commonly used within multiple engineering sectors.

C and C++ (more specifically known as Embedded C when employed in the domain of embedded systems), are popular choices that are selected both for performance, scalability and ease of deployment to a variety of off-the-shelf (and budget conscious) hardware systems. It is an entirely traditional software language where engineers and developers can write precise, deterministic code that typically executes in a predictable, repeatable and in a line-by-line fashion [21]. MBD tools

handle high-level design, can abstract away certain functionality and allow for easier high-level system architecting, whereas low-level embedded code development for controller design allows for easier manual optimization, memory management and interface handling.

In recent years, to move away from tools that require costly licenses, open-source programming tools such as Python have rapidly gained traction within the world of controls engineering. Through the “Python Control Systems Library”, plant models for a variety of both linear and non-linear systems can be modelled via a “State-Space” representation and analysis tools such as Bode and Nyquist plots can be generated for closed loop transfer functions corresponding to a controller/plant interaction [22]. While powerful for data processing, Python does not provide deterministic real-time performance and lacks embedded deployment maturity, instead, being extremely useful for simulating theory-heavy scenarios [23].

3.1.1.3 External Simulation Tools

Simulation of control logic is a critical step within any sort of controller design as it validates the functionality of logic prior to deployment to hardware. This is due to the fact that bugs in control logic intended for automotive applications can have repercussions to human life as well as property damage.

Several commercial software simulation suites exist, with a popular option being CarSim, which is a widely used commercial simulation tool for vehicle dynamics analysis. It offers high-fidelity models of full vehicles, powertrains, tires and driver inputs. It excels at simulating complex maneuvers such as lane changes, braking and cornering, providing accurate outputs for yaw rate, lateral acceleration, tire forces and suspension behavior. One of its key strengths is integration with MATLAB/Simulink, enabling control designers to co-simulate plant and control algorithms in a singular environment across two distinct software packages. Its graphical user interface and large library of predefined maneuvers also reduces development time, while its use in both industry and academia means that results can often be compared across projects with confidence. However, CarSim also has important limitations. While it captures high-level vehicle dynamics, it does not model every subsystem in extensive detail, for example, it abstracts away detailed thermal effects, motor/inverter dynamics and ECU behavior, all which can be modelled in some capacity within Simulink or Simscape. This makes it less suitable for system-level studies where power electronics, battery degradation, or thermal management play major roles. Additionally, licensing costs can also

be significant for smaller teams and its closed-source nature prevents full customization of underlying models [24].

For ADAS and autonomous driving applications, IPG CarMaker is a powerful tool that provides a full virtual environment that includes traffic participants, road networks, sensor models and driver behaviors. This makes it highly suitable for validating system level interactions for holistic ADAS testing, where data transfer and manipulation between perception, sensor fusion, decision-making and control steps must be validated under a wide variety of driving scenarios. Its ability to connect with real ECUs and HIL test benches allows engineers to progress seamlessly between various stages of testing fidelity. CarMaker also integrates with MATLAB/Simulink and Robot Operating System (ROS), enabling co-simulation with custom controllers similar to CarSim. However, the fidelity of vehicle dynamics simulations in comparison to software such as Simulink or CarSim are considerably less deep when used to analyze cornering or ride-handling scenarios. Furthermore, the ever presence of a license cost as well as a steep learning curve (when beginning to use simulated sensors and traffic) can be barriers to entry [25].

Table 4 : Software Tool Selection Recap

Category	Pros	Cons
MathWorks Toolsets and Model-Based Design Tools	Industry standard, easy co-simulation, automatic code generation, extensive libraries, scalable hardware integration.	High licensing cost, dependency on proprietary tools, limited flexibility for non-MBD users.
Embedded C/C++ and Traditional Programming	High control, efficient memory management, open-source options, good for theoretical validation.	Steep learning curve for beginners, lacks abstraction, Python unsuitable for real-time control.
External Simulation Tools (CarSim)	Accurate vehicle dynamics, strong co-simulation support, industry and academic credibility, predefined maneuvers.	Limited subsystem fidelity (no detailed thermal or ECU models), closed-source, expensive licensing.
External Simulation Tools (IPG CarMaker)	Comprehensive ADAS testing environment, real ECU/HIL support, ROS and Simulink integration.	Lower dynamics fidelity, high cost, complex setup for sensor and scenario modeling.

3.1.2 Compute-Hardware Selection / Communication Protocols

Following the selection of the software toolchains to be used, appropriate evaluation of the various hardware platforms as well as communication protocols to be utilized represents an additional critical consideration to be made by an engineering team. Fundamentally, this is because the main bottleneck in limiting software functionality is hardware capacity and requiring that the compute units are able to execute control logic “faster-than-real-time”, to guarantee safe and predictable functionality. The phenomenon of software stacks becoming larger, slower and more resource-intensive with each new version is known as “software-bloat”[26] and therefore, it is of the utmost precedence that hardware should be both powerful and scalable in nature. Additionally, since information between sensors and the various ECUs is transferred in discrete packets of encoded data, the method for communicating that data as well as ensuring the reliability of the data transfer across varying use-cases is also an important factor to consider.

3.1.2.1 Hardware Selection

Low-cost microcontrollers (Arduino, STM32, TI C2000) serve as the entry point for many student and prototype projects. Their main advantages are affordability, wide availability and ease of use and they typically include development environments that are generally well documented, with abundant community support and open-source libraries. For basic control tasks, such as reading sensors and driving actuators, these platforms are more than sufficient and boards like the TI C2000 are designed with real-time motor control in mind. However, their limitations quickly appear in the context of advanced automotive use cases. Namely, their limited processing power, lackluster floating-point performance and constrained memory that can restrict algorithm complexity. Furthermore, they often lack robust safety features and may not meet automotive environmental requirements for noise, vibration, or harshness (NVH) as well as weather or temperature resistance. As such, they are excellent for prototyping work but unsuitable for safety-critical or production-level controller deployment [27].

Automotive-grade development boards (e.g., dSPACE MicroAutoBox, NI CompactRIO, Speedgoat) are one tier up in terms of hardware quality, used for both academic research purposes as well as industrial applications in the automotive industry (and beyond). These platforms are purpose-built as rapid-prototyping controllers, possessing high computational performance, real-time operating systems, extensive input/output (I/O) options and direct support for automotive

communication protocols such as CAN, Local Interconnect Network (LIN) and FlexRay. They integrate seamlessly with MATLAB/Simulink and other model-based design toolchains (as mentioned in 3.1.1.1), allowing for validating through several testing environments without requiring custom low-level coding. Their major downside is cost, as these systems are proprietary and much more expensive than low-cost microcontrollers, making them inaccessible to smaller student teams unless sponsorship is available. Additionally, they are often physically bulky and not intended for long-term in-vehicle operation, meaning they serve more as development tools than as final production controllers [28].

Custom ECUs constructed using automotive-grade processors (Infineon Aurix, NXP, Renesas) represent the most production-level of quality available. These devices are designed for OEM applications, with high computational performance, advanced safety and full compliance with automotive standards like ISO 26262. Developing such platforms offers the greatest realism and scalability, as many OEM supervisory controllers use these exact microprocessors. The disadvantage lies in the steep upfront development cost as teams must often design custom Printed Circuit Boards (PCB), develop or port low-level drivers and manage more complex toolchains. This requires deep expertise in embedded systems engineering that many student teams may always lack and the development cycle is longer compared to prototyping boards [29]. Nonetheless, for efforts that prioritize adherence to industry standards and practices and necessitate long-term serviceability, custom ECU development provides unmatched robustness at the cost of higher complexity and slower iteration speed.

Table 5 : Hardware Tool Selection Recap

Category	Pros	Cons
Low-Cost Microcontrollers (Arduino, STM32, TI C2000)	Low cost, easy to use, extensive documentation, strong community support, suitable for basic prototyping and testing.	Limited processing power, poor floating-point performance, small memory, no automotive-grade safety or environmental compliance.
Automotive-Grade Development Boards (dSPACE MicroAutoBox, NI CompactRIO, Speedgoat)	High computational power, real-time OS, native automotive protocol support, seamless MBD integration, ideal for validation and research.	High cost, proprietary systems, physically bulky, not intended for long-term vehicle deployment.
Custom ECUs (Infineon Aurix, NXP, Renesas)	High realism, compliance with ISO 26262, robust safety and reliability, scalable for production applications, long-term serviceability.	High development cost, long iteration cycles, complex toolchains, requires deep embedded systems expertise.

3.1.2.2 Communication Protocol Selection

CAN, CAN Flexible Data-Rate (FD) and LIN networks serve as the traditional solution for designing in-vehicle communication networks. CAN offers deterministic arbitration, robustness to electromagnetic interference and widespread industry support, making it the default choice for safety-relevant control loops such as engine management, braking, or chassis systems. CAN-FD is an extension of the traditional CAN protocol that includes larger data payload sizes and higher bit rates, supporting large amounts of data transfer between modern ECUs. LIN buses are a cost-effective supplement, intended for simple, low-speed functions such as window actuators, climate controls and seat motors. Physically, CAN buses are created by helically twisting two sets of wires to ensure protection against Electromagnetic Interference (EMI) while LIN buses are single wired connections. The combination of both CAN/CAN-FD and LIN buses for critical data transfer and peripheral system management respectively, allows OEMs to optimize cost without compromising reliability. Their main limitations are bandwidth constraints compared to newer protocols and relatively limited scalability for the data requirements of ADAS feature within connected vehicles [30]. Additionally, these protocols typically offer larger scalability since a CAN or LIN bus can be compared to a highway, with each ECU serving as an exit. Therefore, ECUs can be swapped out and modified to consolidate functionality as long as the physical integrity of the bus is maintained. CAN messages are

transmitted over the wires without the need for any response or confirmation for the communication to be valid [31].

FlexRay and Ethernet serve as higher-performance networking protocols being rapidly adopted to satisfy modern software-defined vehicles and associated ADAS architectures. Automotive Ethernet, particularly with Time Sensitive Networking (TSN), enables high-bandwidth communication (ranging from 10 megabits per second up to 10 gigabits) with deterministic timing, making it suitable for camera feeds, RADAR and LIDAR communication [32]. FlexRay, while offering deterministic and redundant communication with fault tolerance, has lost traction in the automotive space as Ethernet gains momentum. Its structured time-division multiplexing is ideal for safety-critical systems but comes at a higher cost and with less flexibility compared to Ethernet. The trade-off here is between FlexRay's robustness and Ethernet's scalability as Ethernet supports the massive data rates needed in modern vehicles but requires more thoughtful design to ensure real-time reliability, physical serviceability and scalability [33].

Serial, Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C) are elementary communication protocols primarily used at the microcontroller and sensor/board level rather than for full in-vehicle networks. They are ideal for local, short-range communication between chips, sensor modules, or expansion boards due to their simplicity and minimal overhead. For example, SPI is common for high-speed communication with Analog to Digital Converters (ADCs), Digital to Analog Converters (DACs), or motor drivers and I2C is often used for low-speed sensor data exchange while the Serial communication protocol remains a staple for debugging and interfacing with simpler devices. Their advantages are ease of use, low cost and ubiquity in embedded systems, but they lack the scalability, error handling and robustness required for distributed automotive systems. Thus, while critical at the ECU subsystem level, they typically cannot replace bus architectures like CAN or Ethernet for system-wide communication [34].

Finally, a note should be made as to why wireless communication protocols are absent from the options presented earlier for handling vehicle-wide data transfer. This is because safety-critical systems demand deterministic, low-latency and interference-free communication. Wireless communication is subject to packet loss, electromagnetic interference and potential latency spikes, which are unacceptable for tasks like brake actuation, torque distribution, or airbag deployment where milliseconds matter. Security is another limiting factor as wireless communication protocols increase

the potential for cybersecurity intrusions, making safety-critical ECUs vulnerable to signal manipulation. That being said, wireless technologies are used in non-critical feature sets such as infotainment, telematics and vehicle-to-everything communication [35].

Table 6 : Communication Protocol Selection Recap

Protocol	Pros	Cons
CAN, CAN-FD, and LIN	Cost-effective, reliable, EMI resistant, scalable, and widely supported across OEMs.	Limited bandwidth, scalability issues for high-data ADAS, slower than modern protocols.
FlexRay	Deterministic and redundant, ideal for time-critical applications, robust fault tolerance.	Expensive, less flexible than Ethernet, declining adoption, complex implementation.
Automotive Ethernet (with TSN)	Extremely high bandwidth, scalable, and supports advanced sensing and perception data exchange.	Requires careful network design for determinism, potential EMI issues, costly infrastructure.
Serial, SPI, and I2C	Simple, low-cost, and ubiquitous in embedded systems, ideal for chip-to-chip communication.	Not robust or scalable for full vehicle networks, limited error handling.

3.1.3 Version Control Methodology

Since control software development projects can rapidly grow in size and scope and often times reference various files and rely on several dependencies, version control of the software being deployed and worked on is an important step within the development lifecycle of vehicle controllers. Furthermore, the accompanying tool to use when working within the selected methodology typically drives the overarching version control approach. There exist two main version control systems (VCS) that are typically employed within software development projects: centralized and distributed systems. [36]

Traditionally, centralized version control systems work around a single repository stored on a central server where developers interact with this server by checking out files, making modifications and committing updates back to the repository [37]. Only one developer at a time may make changes to a file and this architecture provides a singular “source of truth,” making it easier to manage access, enforce permissions and validate changes across a team. Subversion (SVN) is the most widely used

centralized VCS tool, offering branching, tagging and strong repository management capabilities. It is often paired with integrated development environments (IDE) like Eclipse or Visual Studio through plugins, streamlining the commit, update and review process. Their primary advantages are simplicity, straightforward onboarding for new developers and predictable repository behavior. However, drawbacks include reliance on constant server connectivity, limited capacity for offline work and more rigid branching and merging compared to modern distributed systems like Git [37].

The more common VCS approach is the distributed methodology, where every developer owns a full copy of the repository, including its entire history, on their local machine as well as any subsequent branches that have been generated for containing new work. This allows work to continue seamlessly offline, with changes later synchronized through push and pull commands to remote repositories. Git and web-based wrappers such as GitHub or GitLab, are by far the most widely adopted distributed VCS tools. They are known to be fast, flexible and have robust branching and merging capabilities. Mercurial is another notable tool, offering a simpler command structure while still supporting distributed workflows. Furthermore, the hosting wrappers typically provide issue tracking, code review and continuous integration pipelines in addition to repository storage that align seamlessly with major working methodologies such as Agile. However, they can have a steeper learning curve, and large repositories may require careful management to avoid complexity [37].

One working methodology that is worth noting but is strongly recommended against is the local VCS, as this is a methodology where files are typically shared amongst developers with no remote management considerations and in certain cases, master files and working versions are hosted on a singular working PC. In this situation, merging and version control is entirely nonexistent and when starting with a singular file, changes and revisions of the original file can rapidly grow and diverge until it is practically impossible to verify what is the most “up-to-date” version of a file. Thus, extra care should be taken to avoid this VCS and emphasis should be placed on settling on either a distributed or centralized VCS in the early days of a software development project.

Table 7 : Version Control Selection Recap

Version Control Type	Pros	Cons
Centralized Version Control Systems (CVCS)	Simple structure, easier onboarding, single authoritative repository, strong permission control, predictable repository management.	Requires constant server connection, limited offline functionality, rigid branching, potential single point of failure.
Distributed Version Control Systems (DVCS)	Supports offline work, fast and flexible branching/merging, encourages parallel development, integrates well with modern DevOps tools, supports Agile and CI/CD.	Steeper learning curve, complex repository management, potential performance issues with very large repositories.

3.1.4 Working and Planning Methodology

The main working methodology that will govern the development team, but more specifically, will allow the leadership team working on a software development or engineering project to plan and schedule various tasks can drastically determine the overall experience of a project. The effects of committing to a specific methodology can have significant repercussions for a team, mainly from both a time value and productivity perspective. Thus, effectively comparing the pros and cons of various governance strategies is a critical step prior to starting any work on an engineering project. There exist numerous methodologies, but the following section will recap 3 of the most popular strategies within industry that are the most suitable for control design. [38] [39]

Agile [39] is a popular choice that emphasizes iteration, adaptability, collaboration and delivering working solutions in short cycles consisting of (typically) 2-week sprints. Within the sprints, tasks are prioritized and assigned to team members to serve as their major goals within the sprint themselves. Instead of rigid long-term planning, Agile breaks projects into “Epics”, “Stories” and finally “Tasks”, that have descending scope and hierarchy respectively, breaking down an overarching goal into manageable units with a specific theme. This allows teams to gather feedback early and adjust to changing requirements, making it particularly effective for software and product development in uncertain environments such as student design teams. However, this adaptability of Agile makes it less ideal for highly regulated environments requiring exhaustive upfront documentation. The strengths of Agile lies in its flexibility, customer-centric focus and ability to reduce risk through continuous validation. Additionally, Agile can be challenging to implement successfully as it requires

strong communication, disciplined teams and active stakeholder involvement and when lacking the aforementioned items, projects risk devolving into scope creep or disorganization.

Lean Development [39] was born from the world of manufacturing engineering and aims to consolidate lean manufacturing principles developed by Toyota and the Toyota Production System. Its focus is on maximizing customer value while minimizing waste, regardless of if that waste is time, cost, or effort. Lean emphasizes continuous improvement principles (commonly known as Kaizen), respect for team members and empowering production teams to make targeted and informed decisions. In product development, this manifests itself as the elimination of unnecessary steps, shortening development cycles and improving efficiency across workflows. The advantages of Lean include streamlined processes, lower costs and improved responsiveness to customer needs but, its reliance on disciplined execution and cultural buy-in can be a drawback. Without organizational-wide alignment, Lean Development risks being misunderstood as simply cost-cutting and the random elimination of underutilized resources and assets. Furthermore, the emphasis and focus on efficiency sometimes overlooks the need for longer-term strategic innovation and proactivity. Thus, Lean Development works best in organizations committed to ongoing refinement and environments where waste reduction directly translates into measurable and quantifiable value .

Waterfall [39] is a traditional and sequential methodology in which projects flow through the clearly defined phases of “requirements”, “design”, “implementation”, “testing” and “deployment”, where each phase must be completed before moving to the next. This model is particularly effective in environments with stable, well-defined and predictable requirements such as construction. The main advantage of Waterfall is its clarity in timelines, deliverables and responsibilities, making it easy to manage and audit the quality of work as well as the productivity of team members. Its emphasis on documentation also ensures strong requirements traceability for easier tracking and work justification. However, its rigidity makes it less effective for projects where requirements rapidly evolve, as changes late in the process can be costly and disruptive. Additionally, because testing occurs only after implementation, errors may go undetected until late in the development cycle. Therefore, Waterfall is best for projects with low uncertainty and high regulatory or contractual requirements.

Additionally, there also exist different planning strategies that work in conjunction with a specific working methodology, with the two most popular methods being predictive and adaptive planning

schemes. The methodologies differ primarily in how they handle uncertainty and change throughout a project's lifecycle. Predictive planning relies on detailed, high-level, long-term planning that stays relatively consistent as the project moves forward. It is also well suited for establishing overall timelines and identifying major risks to the project through the use of upfront planning where scope, schedule and requirements are defined early. While it may excel in stable environments, it struggles when requirements and project expectations rapidly evolve. Adaptive planning, by contrast, is iterative, flexible and embraces change by favoring incremental and short-term planning that adjusts as the project does and suits dynamic projects that require iteration. Therefore, while predictive planning prioritizes control and predictability, adaptive planning emphasizes responsiveness, learning and incremental delivery throughout development [40].

Table 8 : Project Management and Planning Selection Methodology Recap

Methodology / Strategy	Pros	Cons
Agile	Flexible, encourages collaboration, rapid feedback, reduces risk, and adapts to change easily.	Can lead to disorganization without discipline, requires strong communication and stakeholder involvement.
Lean Development	Streamlines workflow, reduces costs and waste, promotes continuous improvement, and empowers teams.	Difficult to sustain without cultural alignment, risk of being misused as cost-cutting, may hinder long-term innovation.
Waterfall	Predictable timelines, clear deliverables, strong documentation, and easy auditing of progress.	Inflexible, costly to adapt to changes, delays issue detection until late in development.
Predictive Planning	Enables strong control and predictability, simplifies risk identification, and aligns well with contractual obligations.	Struggles with evolving requirements, lacks flexibility in fast-changing environments.
Adaptive Planning	Highly responsive to change, supports learning and experimentation, and improves stakeholder engagement.	Can lack direction without overarching goals, may cause scope drift without strong coordination.

3.1.5 Cost

While it may be evident, cost of each of the tools to be selected and implemented from both a monetary, effort and effectiveness point of view should always be at the forefront of the decision-making process and in certain circumstances (especially for smaller companies and student design

teams) should be the main driver of determining which decision to settle on. Budgeting and comprehension of the financial state of a team, as well as individual preference can allow for quicker and more effective development. Additionally, exploiting partnerships/sponsorship opportunities may also be a possible avenue for small industry teams as well as student design teams; especially since certain companies tend to have existing resources already budgeted to give away free-of-charge.

3.2 Requirements Generation

Following the finalization of the software stack and working principles to be employed during the control logic development lifecycle, it is subsequently vital to scope out the requirements of the software to be developed. This is done to ensure that work completed by developers is targeted and strives to meet the needs of the system being designed. Software requirements are documented descriptions of the functions, features and constraints that a software system must satisfy and serves as the foundation for design, development and validation. They define what the system should do, how it should behave and the conditions under which it must operate. Furthermore, this eliminates ambiguity surrounding the expectations that the software will need to accomplish (especially within industry between stakeholders) as well as providing a glimpse into the level of commitment required of the larger development team. A well-defined requirement should guide all aspects of the engineering lifecycle and can encompass both functional and nonfunctional aspects of the final intended product, guiding both technical and non-technical teams alike. The following section recaps several sources that a development team should source requirements from and the motivation that each source carries along with it, specifically geared towards the context of student design team control logic development. Furthermore, a note related to the format of a requirement is provided but is not the focus of the section.

3.2.1 Safety, Reliability and Standards Compliance

In academic competitions such as EcoCAR or FSAE, there often exists multiple different forms of documentation that help dictate specific design constraints and performance requirements expected of the generated design. These can include restrictions on energy storage systems, communication protocols, interface safety features, or limitations on modifications to donor vehicles [41]. Within the automotive industry or adjacent industries similar to EcoCAR, governing bodies such as the Federation Internationale de l'Automobile (**FIA**) for motorsport or Transport Canada and International Organization for Standardization (**ISO**) for road vehicles, issue regulations that directly

impact requirements. For example, FIA rules may demand specific fail-safe logic during high-voltage shutdown procedures [42] while EcoCAR rules may mandate the use of functional safety processes or standardized connectors for high-voltage components [41]. Capturing these external inputs explicitly as requirements ensures the team avoids disqualification, meets competition scoring objectives and develops systems aligned with safe and tested industry standards.

Beyond competition rules, governmental and regulatory compliance requirements provide another major source of requirements, particularly in areas of safety and reliability. For instance, ISO 26262 [43] dictates functional safety processes for automotive electronics, requiring that ECUs implement diagnostics, fault detection and safe states in the event of critical errors. Similarly, standards like SAE J1939 [44] constrain how communication networks such as CAN must operate to ensure interoperability and robustness. These requirements should be documented alongside system specifications and explicitly tied to safety and reliability goals for example, such as ensuring brake-by-wire systems default to a mechanical fallback in case of failure. By systematically sourcing and generating requirements from competition rules, regulatory standards and government safety mandates, teams create a framework that not only keeps them compliant but also builds confidence in their designs' robustness. To add, adherence to regulatory requirements and various engineering standards/conventions can help eliminate future headaches when related to serviceability or part availability, especially due to the fragile nature of student vehicle prototypes. This makes it so that teams can readily source replacements parts if they are damaged; eliminating unnecessary work when using design conventions that are tried and true.

3.2.2 Supplier and Supporting Documentation

Requirements should also be derived from supplier documentation to account for the inherent limitations of hardware components being integrated into a design [45]. For example, if a third-party drive unit is used, the supplier will typically provide specifications such as maximum torque, current draw, thermal derating behavior and communication interface protocols. Similarly, high-voltage wiring and connectors will have defined current limits, insulation ratings and thermal properties that cannot be exceeded without risking safety or unintended failure. These specifications must be translated into explicit requirements that guide system design, such as maximum allowable continuous current through a harness or torque limits in the controller logic.

Just as equally important are the limitations of hardware designed in-house, which require careful cross-team communication and verification. A custom PCB, for instance, may be designed with limited current-carrying capacity, specific sensor bandwidth or constrained microcontroller resources. If these limitations are not formalized as requirements, the software team may inadvertently design algorithms that exceed or lag behind what the hardware can support, mistakenly leading to unintended functionality and in student teams, where budgets are limited and hardware often serves multiple roles, this integration is even more critical. While industrial companies rely heavily on supplier hardware that are usually custom built, student teams may combine sourced components with custom solutions, making it essential to recognize and respect each component's constraints. This can only be achieved by having constant communication between stakeholders responsible for the integration of both code and hardware into the final vehicle. By treating both supplier specifications and in-house hardware limitations as formal requirement sources, teams ensure reliable integration, avoid costly redesigns and build a system that operates within realistic performance and safety margins.

3.2.3 High Level Product Goals

Arguably the most important set of requirements stem from what are the intended goals of the product / vehicle to be designed and what the software must accomplish to achieve these functional requirements. Product goals form the foundation of requirements and must strike a balance between what the team or client envisions for their vehicle as well as what is realistically feasible with the choice of selected hardware (both computational hardware but also actuators and mechanical structures). In student design teams, product goals often originate from performance targets such as achieving faster lap times, improving efficiency, or optimizing driveability but may also be related to less tangible targets such as ease of serviceability or rapid integration. These goals are commonly decompositions of a higher-level goal that is determined by the team and extends across all facets of the design process, but is usually abstract. These requirements can also stem from external constraints like EcoCAR rulebooks, which define minimum functionality requirements in order to meet competition standards and thus, documenting these goals explicitly ensures that the system is aligned with both stakeholder desires and technical feasibility.

3.2.4 Competitor Analysis

Competitor analysis is another essential tool for generating requirements as it helps establish a benchmark for both performance and innovation. In automotive and competitive contexts, this means

studying what other teams, manufacturers, or industry leaders are doing. For a FSAE or EcoCAR team, this could involve reviewing publicly available technical papers, presentations, or rule-allowed design disclosures from competing teams to get a sense of what other teams are aiming for but also to obtain insight into what different groups deem to be feasible. By identifying industry trends and the current “bar,” teams can generate requirements that are not only competitive but forward-looking, ensuring they are not designing in isolation and establishing goals that are current with industry trends and practices. For instance, if competitors are implementing systems such as cybersecurity protocols and have achieved a solution that is recyclable, then generating requirements based on this solution can save significant time which can be in turn dedicated to more pressing design tasks.

Beyond direct performance benchmarks, competitor analysis also highlights gaps or missed opportunities. If competitors’ systems show weaknesses, such as sluggish torque response or limited diagnostic coverage, requirements can be written to avoid similar shortcomings. Importantly, competitor analysis is not just imitation as it informs requirements that distinguishes your product from the next. For example, learning that other teams have poor real-time fault handling could inspire requirements like “fault detection latency shall be less than 10 milliseconds with immediate torque derating” which subsequently allow for better protection of consumable components.

3.2.5 Diagnostics and Fault Mitigation

Requirements should also be generated with the goal of defining what remedial actions should be taken if the vehicle under development enters a state this undesirable. It should also be defined within the requirement whether that state necessitates recovery to regular functioning or complete termination of operation. Furthermore, formalizing protocols to automatically trigger and track errors in the form of diagnostics, such that the development team can correct them in a targeted fashion, is essential in the development of a robust control system [45]. These requirements can be derived and generated according to industry standards (e.g., ISO 26262 functional safety) or in accordance with competition mandated sequences and safety requirements [41]. This is with the goal of ensuring that the system not only performs under ideal conditions but also remains safe and predictable when failures occur, since safety and reliability would be directly embedded into the design.

It should be noted however that developing diagnostic and mitigation requirements also requires close collaboration between hardware and software teams, since fault responses must be both technically feasible and effective. For example, thermal derating of drive units, an EcoCAR mandated

remedial action, must be designed such that hardware is protected but also regulated to allow for a chance to exit the erroneous state, while also maintaining the overall predictable functionality of the vehicle.

3.2.6 Requirements Tracking

Following the generation of requirements, emphasis should be placed on ensuring that they are properly documented and tracked throughout their implementation lifecycle. Traditionally, documents of this type are typically known as Requirements Traceability Matrices (RTM) and they aim to compile and track important information such as requirement origin, the sub team or developer responsible for implementation, implementation status, testing status as well as priority. Secondary information such as the date or additional notes may also be included.

This exercise is done as it enables clear traceability from high-level objectives down to subsystem-level specifications and corresponding test cases. Tracking also allows teams to monitor requirement changes, ensuring that any modifications are reviewed, justified and communicated across sub teams and stakeholders with the goal of preventing miscommunication, duplication of effort and gaps between software development and testing teams. Furthermore, by compiling all of the requirements for the software into a concise document, teams can also visually review the magnitude of work that is required of them. This ultimately enables accountability and continuous alignment between what was planned, what was built and what has been validated.

3.2.7 Requirement Format

Several considerations should be kept in mind when performing the exercise of requirement generation:

1. Requirements should be written in a format that is suited for direct translation into either one or several test cases.
2. Requirements should be testable and avoid vague language that is subjective such as “good” or “comfortable” and strive to include tangible metrics and number targets.
3. Requirements should follow a standard formatting convention with a repeatable sentence structure and using verbs such as “shall” or “must be” instead of “should” or “will”.

4. Requirements can be separated based on hierarchy or priority but should be labelled and indicated as such.
5. Requirements that are generated must be in accordance with each other, attainable and not conflict with one another.
6. Requirements may evolve and change over time as requirements and requirement tracking systems are considered to be “living” and constantly changing over time, as team priorities and resources shift.

3.3 Development Methodology

Once the set of tools to be used during the development process has been selected and the requirements for which the software must be accomplished has been outlined, developing the actual control logic to be deployed to an onboard ECU is next. Developing an automotive control system requires working to a structured and deliberate methodology that is meticulous but allows for flexibility due to the nature of the environment of a student design team. Teams can adhere to either one of several industry standards or a novel one that aligns with team principles.

Therefore, the following section presents the considerations to think about as well as recommendations for the next stage of the control logic development lifecycle. The development methodology ultimately serves as a road map, formalizing how development teams should go about architecting their logic and writing discrete units of code. These major considerations were analyzed by UWAFI during the generation of the team’s selected working methodology and guided Propulsion Controls and Modelling (PCM) subteam members as they iteratively expanded the code base for deployment for vehicle deployment. Nonetheless, a development methodology should reference the established link between requirements, design and testing, to preserve traceability while crucially outlining how design tasks are transformed into code. Moreover, it should set up a framework for continuous iteration and growth while allowing teams to incorporate lessons learned, to subsequently go back and correct software that was lacking in the face of evolving requirements.

3.3.1 System Architecting and Partitioning

The overarching structure of the control software, similar to most other software development projects such as websites or applications, must be architected and laid out in a logical manner. An architecture should be selected to allow developers to easily traverse the expectedly large code base

while also being able to effectively work on the project itself. A well-thought-out software architecture should also primarily address team needs, and may vary from project to project based on requirements and the specific functionality being developed for that software unit.

Traditionally, as vehicle control software typically encompasses a tangible customer facing feature set or key functionality that is required of the vehicle, such as auto-park or cruise control functionality, control logic is often partitioned with the aim of regrouping several similar sets of functionality, that are required of the feature in question. They also typically encompass any redundant operations required of the feature ranging from sensor data manipulation to actuator control generation. This is done for several reasons, namely due to the ability to assign ownership, simplify debugging and manage dependencies more effectively between several units, models or referenced files. Additionally, modularity and discretization within the software project allows for individual components to be reused, tested and refined independently before full system integration. Each subsystem should have clearly defined inputs, outputs and responsibilities, ensuring that development remains isolated but compatible with adjacent development efforts. Several architecture styles exist as described by Staron and reviewed in Chapter 2.1. As was highlighted in the literature review, an architecture style is simply a method of conveying the principles and priorities of a software feature and simplifies the communication of what a feature is meant to do [10]. Therefore, architecture selection typically becomes a stylistic choice that aligns with team preference. that allows for ease of integration with software toolchains and working principles. Furthermore, since certain architectures are often better suited to certain types of features over others, extra care should be taken to lay out a software feature effectively.

With 10 formal architecture styles being referenced in Chapter 2.1, 3 specific styles were reviewed as these closely align with the format of the UWAFD PCM codebase to be presented in Chapter 4. The overarching recommendation of this thesis though, is that the deployed software be partitioned per feature and similar feature sets be regrouped with others of similar behavior. The formal architectural style to be used can vary from user to user based on what is most appropriate when reviewing what the individual software unit itself, is aiming to do. For example, a “Pipes and Filters” architectural style may be better suited for a feature such as object detection where data is continuously being interpreted and acted upon when the feature is enabled [10].

3.3.2 Task Generation

Task generation bridges the gap between requirements that a software feature must meet and actionable development work to be performed by a developer. Each system requirement, whether functional or nonfunctional, must be decomposed into discrete tasks that engineers can design, implement and verify.

It is recommended that tasks are generated by first identifying which sets of requirements pertain to a singular feature set. Once this has been done, translation and mapping each requirement into a tangible engineering task such as a control algorithm development task, a research task, testing task, etc... should be performed to establish traceability between software requirements and the generated task itself. It should be noted that several requirements may be bundled into a singular concrete task, with the goal of merging similar work that can be done in sequence such that team resources can be used effectively. By breaking large, complex requirements into smaller, measurable tasks, several portions of the software base can be developed in parallel due to their compartmentalization (in conjunction with the recommendation for a software architecture that favors partitioning) and ensures accountability by the individual undertaking the task itself. Furthermore, defining acceptance criteria and measures of success for each task, supports development as it gives the developer insight into what is the end goal. By generating tasks based on the requirement set, each deliverable can be tied back to specific requirements during validation and supports traceability, so everything that the software must perform is addressed in one fashion or another. While not entirely necessary, it should be also considered that effective task generation is complimented by categorizing tasks according to complexity, dependency, subsystem ownership as well as expected time required for completion. This is done with the goal to manage workload distribution and resource planning, as tasks should be scheduled/generated with the goal that development progresses logically and efficiently.

To add, depending on the “Working and Planning Methodology” selected, as was discussed in Chapter 3.1.4, tools such as Jira and methodologies such as Agile necessitate proper task generation to be used effectively as they help to visualize workflow, assign responsibilities and monitor progress. Thus, care should be taken to generate tasks properly and in student environments, where time and resources are limited, structured task generation prevents oversights and ensures that even ambitious projects remain achievable within the available timeframe.

3.3.3 Deployment and Integration Environment

The deployment and integration environment dictates how software evolves, is integrated into the main code base, as well as conventions for being loaded onto computer hardware. It encompasses everything from version control management to hardware deployment.

It is recommended that software integration should be done using a change management and reviewal system where the main developer of a task is responsible for ensuring that the development work is integrated to main branches / repositories. They should also subsequently review the outcomes of the integration work with a reviewer to ensure that the work was done properly. Alternatively, teams can also allow a team lead to manage all of the integration work themselves. Furthermore, each iteration of software must be traceable, with clear links to its source version and following testing phases, it should reference results as well as any additional notes related to required improvements. This is all done with the goal of ensuring that the software can evolve safely and that development work can come together in a cohesive manner that compliments progression. Furthermore, it is recommended that software is updated and integrated often, to ensure that there is no mismatch between final software versions and development branches.

While integration of development efforts is crucial in the control algorithm development lifecycle, deployment processes are also an important step in successful algorithm development. While deployment may be simplified pre-emptively during the software toolchain and ECU hardware selection phase, it should be noted that virtual development does not necessarily account for issues that may arise with compiler compatibility, memory usage and incompatible hardware configurations. For instance, the compiled software image may be loaded via a USB serial connection or Ethernet link depending on the platform, however issues with task execution time that are independent of developed logic but are entirely related to hardware limitations may arise. Therefore, outlining clear steps on how often, when and how someone should go about the activity of hardware deployment for the sake of ensuring easy testing (especially during HIL testing phases) should be clearly laid out.

For student teams, having even a lightweight deployment and integration process helps maintain software reliability across multiple contributors and prevents regression, when new features are introduced.

3.3.4 Communication and Networking Considerations

While not typically the most difficult aspect of control algorithm development, the communication layer forms the backbone of any distributed control system, by linking several ECUs, sensors and actuators together in a unified control network. Therefore, the communication layer of a singular ECU during development should always be treated as a distinct subsystem that is isolated from core control logic. Ideally, this is done for several reasons namely for ease of testing, allowing scalability as well as reducing dependency.

As the communications layer is related to a physical consideration and concerns itself with concepts such as transfer rates or message encoding, it is entirely removed from any sort of mathematical or algorithmic operations that are performed with a software feature. Therefore, isolating it in its own subsystem ensures that when actual logic is tested throughout the various testing environments, code that is unrelated can be ignored. Furthermore, as software is expanded and as more sensors or ECUs are added to a network, teams can modify network configurations or protocols without rewriting core logic to carefully manage and prevent any sort of latency or collision concerns.

3.3.5 Inspiration Via Industry Standards

Industry standards and tested methodologies common in the automotive industry played a vital role in guiding the bulk of the recommendations provided above, namely in several of the recommendations surrounding software architecture planning, task generation and communication layer abstraction.

Industry standards such as ISO 26262, which outline processes for ensuring functional safety in automotive electronics [43], as well as standards such as AUTOSAR which define structured software layering methods, separating application logic from hardware abstraction and communication layers [46], have heavily inspired the considerations to reflect upon when generating a development methodology. Furthermore, principles from the discipline of Systems Engineering such as the V-model, which is a framework that visually represents the relationship between requirements and granular development tasks, which subsequently leads to several verification and validation phases (i.e.: progressing through various XIL environments) have also informed several of the recommendations above. It is further recommended that standards such as ASPICE and ISO 14229 also be reviewed to allow the reader of this thesis to gain insight into standards that surround process assessment as well as conventions for vehicle diagnostics respectively.

The reason these industry standards were not directly recommended for full adoption is due to the nature of student design environments, where teams operate in a prototype-oriented setting where rapid iteration, limited resources and the ability to have hardware evolve rapidly takes precedence over strict compliance or adherence to a formalized workflow. While many principles from several of the standards above are useful, applying them directly is often impractical as these standards assume mature organizational structures and extensive documentation processes, factors that typically exceed the scope of most student projects. Instead, select elements and best practices should be adapted to fit the team's existing habits and workflows.

3.4 Testing and Validation Framework

Once development is complete, testing and validation become the next critical step in ensuring that the implemented system performs as intended, meets defined requirements and can operate predictably when injected with various inputs. In the context of automotive and student design projects, this stage ensures that the system not only functions correctly in isolation but also integrates seamlessly within the full vehicle environment.

The validation process typically progresses through multiple layers known as XIL testing and for example, in a model-based design workflow, typically indicates traversal through MIL, HIL and vehicle-in-the-loop (VIL) testing. Each environment is traversed with the aim of building confidence in the system's robustness as each stages allow incremental verification while minimizing risk to hardware and safety. Validation also supports requirements traceability, confirming that each functional and nonfunctional requirement has been satisfied through measurable testing while helping to uncover issues that may not have been accounted for during development, such as timing delays, signal mismatches, etc... The following section presents several recommendations and considerations to keep in mind when establishing a rigorous testing framework to both accurately and effectively validate control software functionality.

3.4.1 Types of Testing

When approaching testing, the main goal should always be to validate and verify if the work completed by the development team meets the intended functionality, when compared against generated requirements. Comprehension of the logic behind the generated software is typically a secondary consideration. Through development of the UWAFt PCM controller, two major types of

testing schemes have been formulated, based on experience and past efforts surrounding validating the controller itself.

“Open loop” testing is an internal definition for a testing methodology that has been frequently used during the initial phases of controller verification, mainly when validating requirement sets that relate to occupant and hardware safety. In this scheme, data can be injected into the controller to stimulate various portions of the codebase to be able to observe the controller response, while ignoring how the output influences the overall vehicle body or battery. This is strictly done with the goal of ensuring that controller actuation commands (such as a torque command) fall within predefined safety margins and that the controller can default to defined safe states when injected with faults, such as an e-stop event. Furthermore, the lack of feedback also allows for rapid iteration as it allows for the ability to systematically assess how adjustments to the model affect its dynamic response. Features that are state machine based or fault mitigation / diagnostic logic are better suited to this type of testing as these types of control logic are conditional and deterministic (meaning that they operate under specific scenarios and evaluate to discrete / finite states).

“Closed loop” testing has been used once “open loop” testing has been completed with the goal of verifying the performance characteristics as well as the dynamic response to a set of varying inputs. Furthermore, this scheme also focuses on features where the output magnitude is relevant and can drastically influence the vehicle body as well as the subsequent inputs into the testing environment. For example, “closed loop” testing on a feature such as a pedal map would be performed with the goal of benchmarking how the torque response impacts both drivability and acceleration across a drive cycle or even a wide-open throttle (WOT) test. It typically always necessitates some sort of vehicle plant model to simulate the physics of a vehicle and generate outputs such as acceleration or speed metrics which are not values that are calculated by the control logic itself. In this scheme, the vehicle model can range from a singular model of a wheel and tire all the way to a full vehicle model however, it is up to the feature tester to decide what type of feedback is required into the system.

Both types of testing schemes have been fruitful for UWAFt however, it is recommended that the testing types be used effectively. Moreover, the requirement set as well as the criticality of the feature being tested, should drive both the testing scheme selection as well as the type of data being used as an input into the system. For example, data used can range from constant inputs to recorded CAN log

traffic and should be selected based on the level of fidelity that is required and the mission criticality of the control logic.

3.4.2 Testing Environments

Test environment as well as testing fidelity also plays an important role in the testing framework for a student team, as it directly influences the reliability, accuracy and transferability of validated results from simulation to real-world operation. A well-defined test environment allows the team to verify that each subsystem performs as expected under realistic conditions. High testing fidelity also ensures that behaviors observed in simulation accurately reflect those that will occur in the physical system, reducing the likelihood of unexpected failures during integration or competition. For student teams with limited access to full-scale testing facilities, maintaining high fidelity in virtual environments becomes even more critical to build confidence before deploying on to vehicle.

MIL testing is the initial validation phase where both the controller and the plant model are represented as mathematical models within a simulation environment. MIL is performed with the intention of verifying the logical correctness and stability of control algorithms as well as actuator commands before any code is generated. MIL testing allows rapid iteration, parameter tuning and debugging at low cost and provides a safe, flexible means to test high-level functionality early in development without relying on physical hardware [47].

Software-in-the-Loop (SIL) testing evaluates the controller code, typically auto-generated from the MIL model, running on a computer and in certain situations, interacting with a simulated plant. This approach validates that the compiled software behaves consistently with the original model and that code generation has not introduced functional errors, as the process of code generation is typically an automated process using third part tools. SIL typically presents similar utility to MIL with the added benefit of the ability to test the final production code to eventually be deployed to hardware in a controlled environment.

Processor-in-the-Loop (PIL) testing executes the compiled controller software directly on the intended target processor or ECU and verifies timing performance, computational load and memory usage under realistic processing conditions. PIL testing exposes hardware-specific limitations such as latency or precision errors caused by processor architecture and ensures that algorithms are both functionally correct and computationally feasible for the intended embedded hardware [47].

HIL testing connects the physical controller hardware used in PIL to a real-time simulator responsible for simulating the plant and vehicle environment while recreating real-time effects such as sampling time mismatches. The controller interacts with simulated signals exactly as it would in the vehicle, allowing comprehensive testing of logic, communication and fault responses while enabling validation of safety functions and fault mitigation without risking damage to physical systems [47].

VIL testing integrates real vehicle components (and in certain situations, final production vehicles) into the testing environment to validate system interaction and communication under semi-realistic conditions. For example, an ECU may control a physical motor while other vehicle signals, such as sensor feedback or driver inputs, are virtually simulated. In certain situations, simulation may be omitted entirely, with control code instead being validated on production hardware in a test vehicle. VIL testing is particularly valuable in development programs where not all subsystems are available or when testing risky scenarios [48].

Driver-in-the-Loop (DIL) testing introduces a human operator into a simulated environment, often through a driving simulator or virtual test bench and allows the driver to interact with the simulated vehicle in real time, enabling assessment of control system behavior in response to the subtle nuances introduced by driver behaviour. DIL testing bridges the gap between technical validation and subjective evaluation, ensuring that system responses feel natural and predictable to a human operator and is particularly useful for refining driver assistance features and interface design [48].

Several factors drive the testing environment selection, namely fidelity requirements and cost, both financial and temporal, as well as feature criticality and severity of potential unintended consequences in the event that a feature were to operate unexpectedly. For example, logic pertaining to controlling emergency disconnect hardware in the event of a crash or thermal runaway should be rigorously tested across all environments whereas a feature related to contactor control might necessitate less meticulous testing. It is recommended that a student team's testing workflow places emphasis on the MIL phase, as it provides the most efficient environment for validating control logic and tuning algorithms without relying on physical hardware. MIL testing enables rapid iteration and debugging, making it ideal for developing stable and functional control code early in the process. Additionally, HIL testing is equally as important but is better suited for verifying hardware interfaces, signal fidelity and integration between electronic components. Environments such as DIL testing offer

valuable realism, however the significant time, setup and equipment costs outweigh its benefits in student environments, where easy vehicle access allows for more practical and direct testing.

3.4.3 Test Tracking

Just like requirements, test cases should also be systematically tracked to ensure comprehensive coverage and to gain insight into the overall percentage of the code base that has been validated. Each test case should have a unique identification number, a clear description of its purpose and information about the corresponding system or software requirement it validates. Additional metadata such as XIL (e.g., MIL, SIL, HIL) testing status, execution date, responsible tester and the specific software version under test should also be recorded. This structured approach allows teams to identify gaps, monitor testing progress and ensure that all requirements have been properly verified under the appropriate conditions. Properly managed test documentation also facilitates regression testing when updates are made and gives teams insight into the utility of the code being developed. For instance, if all requirements for a feature are met and the testing status demonstrates that certain aspects of the logic have gone unused, this might indicate that the code has become bloated and optimization might need to be performed.

Test case documentation can be integrated directly with requirements tracking to establish a clear one-to-one or one-to-many relationship between requirements, implementation and validation. This traceability ensures that every requirement has a corresponding test case verifying its fulfillment and that test outcomes can be traced back to the exact software version and configuration used. Merging these data sets creates a streamlined verification system, simplifying audits, competition reviews and safety analyses.

3.4.4 CI/CD

A useful tool that is optional but is highly recommend is Continuous Integration and Continuous Deployment (CI/CD), which is a development practice that automates the building, testing and deployment of software each time changes are made to the codebase. In a testing workflow, CI/CD integrates seamlessly by automatically running unit tests, model validations, or even SIL simulations whenever new code is pushed to a repository. This ensures that issues are detected early and that all software versions remain stable and verifiable any time they are pushed to remote repositories. This inherently improves consistency and reduces integration errors, as a CI/CD pipeline is automatically

able to tell you if pushed code meets minimum established criteria without explicit user validation. However, setting up a CI/CD pipeline requires significant upfront effort and infrastructure as dedicated external computers or servers must remain continuously online to host and execute the pipeline and they must be powerful enough to handle frequent builds and tests. Maintaining this infrastructure introduces ongoing hardware, energy and maintenance costs, but for teams that can support it, CI/CD dramatically increases reliability, traceability and efficiency in software validation and deployment [49].

3.4.5 Additional Considerations and Best Practices

Several additional considerations should be kept in mind when performing testing of control logic, namely:

1. That testing also includes edge cases and strives to test against potential failure modes as these typically lead to scenarios that are unintended and can have large impacts to both people and hardware.
2. Code coverage is a software testing metric that measures the extent to which the source code of a program has been executed during testing. Teams should strive to achieve high levels of code coverage, but its primary use should be to highlight untested sections of code that might contain hidden bugs.
3. Tests must be repeatable under identical conditions to produce reliable results and may be accomplished by employing test scripts and test plans.
4. A singular requirement can be validated with either one test case or many and similarly, one test case may validate one requirement or a group of requirements. Testing should also be strived to be reused within reason as software is matured.
5. All tests should log testing results either in the form of a test report or data log files to ensure traceability while following a formalized test plan. This documentation supports debugging and verification audits while allowing for teams to review data to understand why a test case may not be passing. Furthermore, this allows the tester to gain insight into whether the software contains a bug or if the test case has been poorly designed and executed.

3.5 Codebase Maintenance and Monitoring / Debugging

Code base upkeep is an essential part of sustaining long term reliability of the validated software throughout its lifecycle and utility when deployed to a vehicle. Performing the exercise of regular maintenance not only ensures that the software remains functional and adaptable to changing hardware and performance requirements, but also that it abides by student design team competition regulations which can rapidly change from year to year. Proper software monitoring also plays a critical role in identifying and resolving defects that might have slipped through the testing phase through methods such as visualization and diagnostics. Together, these activities form the foundation of software sustainability and allow for the system to continue to operate safely, predictably and efficiently even as it evolves through multiple development phases.

3.5.1 Data Logging, Diagnostics and Visualization

A powerful codebase monitoring method that allows for testers and occupants to be reactive to issues when programmed remedial action might have been lacking or failed altogether, is the use of data visualization. A visualization system serves as a connection between control code and human interpretations, providing developers with real-time insight into the behavior of software and hardware during operation. Such systems can be implemented in several ways with the most accessible solution being a custom graphical user interface (GUI). The GUI can be developed in parallel to the codebase (while being completely standalone) or using built-in visualization tools within development environments like Simulink, LabVIEW, or dSPACE tools [50].

The visualization should provide a comprehensive snapshot of both functional and diagnostic data to support debugging, testing and real-time validation. It should also visualize the primary system inputs such as sensor readings or driver commands and the primary actuator outputs like motor torque commands or state machine requests, to ensure intended responses are achieved. Additionally, critical system health parameters such as coolant flow rate, temperature, current and voltage should be displayed to monitor thermal and electrical performance in real time. Boolean signals, such as existing or custom fault flags and status indicators should also be visualized as they provide immediate insight into system state and fault activity. Having these signals validated and clearly visible enhances safety, speeds up debugging and allows developers to detect abnormal operation before it leads to hardware damage or unsafe conditions.

Diagnostics and fault requirements are essential components of a robust software architecture and should always be defined explicitly during the requirements phase. This recommendation is repeated here to strongly emphasize their importance in maintaining system safety and reliability as robust diagnostic logic can automatically identify abnormal conditions and trigger appropriate mitigation responses without requiring human intervention. These features prevent runaway failures that are not redeemable and risky to the overall longevity of the development vehicle. Furthermore, performing formal safety analyses such as Safety of the Intended Functionality (SOTIF), System-Theoretic Process Analysis (STPA) and Hazard and Operability Study (HAZOP) allows developers to anticipate potential failure modes and assess their severity and likelihood pre-emptively, to better accommodate these extraneous situations directly within their control software [52] [52]. Integrating these analyses into early design stages ensures that faults are not only detected but also handled safely and predictably, establishing a foundation for functional safety compliance.

Another recommendation that was quickly outlined in Chapter 3.4.5 is the necessity for data logging. It serves as one of the most powerful tools in software validation and debugging, as it enables developers to capture, analyze and replay system behavior over time. Logging conventions should also be formalized and as much relevant data as possible, should be recorded with precise timestamps as this data can then be used for regression testing verification. It allows comparison between previous and current software versions to confirm whether changes have led to tangible improvements or regressions. Logging also supports offline analysis and replay testing where data collected from real runs can be injected back during “open-loop” testing simulations to replicate specific conditions without requiring physical re-tests. Moreover, in student environments, logs are invaluable resources that help to validate development and calibration work. Therefore, it is recommended that teams dedicate time to formalize procedures for data logging and review following a drive cycle or hazardous event, with the goal of allowing easy and systematic root cause analysis.

3.5.2 Refactoring and Environment Updates

Refactoring and environment updates are essential practices in maintaining a high-quality codebase since as software grows in complexity, code can become fragmented, inefficient, or difficult to read, especially in environments where multiple developers contribute to a singular project. Refactoring involves reorganizing and cleaning existing code without altering its external functionality, with the goal of improving readability and maintainability. This can include consolidating repeated logic into

reusable functions, renaming variables for clarity, restructuring file hierarchies, ensuring consistent coding standards or even removal of depreciated code entirely. Refactoring regularly also prevents the buildup of technical debt, making the system easier to debug and extend with new features as the project is passed from team to team. Alongside this, maintaining a consistent development environment is equally critical. Tools such as compilers, IDEs and simulation platforms must be periodically updated to ensure compatibility with evolving hardware and software dependencies. However, updates should be performed in a documented and controlled manner to avoid introducing new inconsistencies. For example, confirming that team machines can run updated versions of software ahead of dedicating time to procure new licenses should be completed ahead of time. Therefore, establishing version control for both the code and the environment (e.g., tracking software tool versions via documentation) ensures that the system can always be reproduced and maintained over time.

3.5.3 Standard Operating Procedures and Safety Guidelines

Establishing and enforcing Standard Operating Procedures (SOPs) and safety guidelines, while not directly tied to the development of control software, still remains vital to maintaining both the integrity of the codebase and the longevity of the software project or team. SOPs formalize how code is written, reviewed, tested and deployed with the goal of ensuring that every team member follows consistent practices regardless of experience level. This includes enforcing version control usage, standardized naming conventions, documentation requirements and defined test plan execution before code integration. This subsequently can also extend to developing safety guidelines related to operation of the vehicle, during both normal and abnormal operation. This is done in order to avoid entering a non-redeemable state which necessitates unorthodox methods to reset the vehicle. Regular code reviews, structured test plans and sign-off procedures help catch issues early and maintain accountability across contributors, especially in environments with high turnover such as student design teams.

Chapter 4

UWAFT Propulsion Supervisory Controller

The following section seeks to provide a tangible example of the result of the application of the development workflow generation process presented in Chapter 3. This will be accomplished in the form of a recap of the design choices and conventions that were created (as well as reasoning) which led to the standardization of UWAFT's control development methodology and subsequent deployment of the PCM controller to a 2023 Cadillac LYRIQ. Each previously discussed design consideration is revisited in context, demonstrating how the earlier framework can translate into actionable engineering outcomes. Within each design step, trade-offs as well as influencing factors are discussed, such as cost, complexity and system performance with the goal of providing a realistic representation of the decisions an engineering team may face when developing a controller. Ultimately, this chapter serves to bridge the conceptual methodology with practical execution and showcases how a structured development process yields a robust, reliable and competition-compliant controller, suitable for advanced automotive applications.

4.1 Selected Toolchains

This section provides an overview of the tools, hardware platforms, software environments and management principles utilized by the team in the development of the UWAFT PSC. It outlines the integrated toolchain and organizational methodologies that supported controller design, implementation and validation with the goal of creating the foundation for a structured development process.

4.1.1 Software Tool Selection

For the development of the UWAFT PCM controller, the team selected MATLAB and Simulink as the primary software tools for controller design, code generation and validation. The entire control logic was implemented using Simulink, leveraging both standard and custom cybersecurity library blocks to create modular and segregated subsystems that worked in unison to achieve propulsion functionality on the LYRIQ. Supplementary MATLAB scripts were used to support secondary tasks such as data analysis, automated test setup as well as testing data generation for testing workflows. Validation was primarily performed using Simulink Test, supported by Simulink-based vehicle models, while co-simulation with CarSim allowed for higher-fidelity dynamic evaluation. CarSim

was seldom used and deployed mainly for evaluation of preliminary vehicle dynamics-based functionality such as traction control. The team, however, intends to keep using the tool going forward for rudimentary wheel flare management functionality. The decision to adopt this tool chain was driven by its accessibility and powerful integration capabilities but admittedly, the fact that the MathWorks tools were fully sponsored, came with formal training and support and seamlessly interfaced with the team's embedded hardware was the main driver behind the decision to select these set of tools. Moreover, the MBD workflow is well-aligned with both industry standards and the academic curriculum, making it intuitive for both experienced and new members [18] [19].

There were numerous advantages that were presented with this selection, namely Simulink's integrated ecosystem allowed for rapid prototyping and efficient testing within a unified environment. Additionally, the graphical interface enabled clear visualization of control logic in an easy to comprehend manner that is intuitive, easing collaboration when working with new members or external subteams. However, several limitations were also encountered such, as the high level of abstraction away from low-level C code sometimes introduced obscure compilation issues, particularly related to file paths and build configurations inherent to MathWorks tools. Additionally, when Simulink lacked explicit functionality, unconventional workarounds or unorthodox block configurations were required to achieve desired behavior. While these issues occasionally complicated debugging, they were overcome by the convenience and productivity that the Simulink and MATLAB ecosystem provided, making it an effective and scalable solution for UWAF's controller development workflow.

4.1.2 Hardware and Communication Selection

For hardware, UWAF selected the Speedgoat hardware line as the foundation for controller development and deployment. The Speedgoat Baseline M served as the primary vehicle controller, while the Speedgoat Performance system was used for HIL testing, as well as tertiary off-vehicle development tasks such as real-time execution of Simulink code snippets and compilation verification. This selection was heavily influenced by sponsorship and support as the Speedgoat real time target hardware and accompanying training saved the team an estimated 40,000 US dollars in procurement costs. The hardware's seamless integration with the MathWorks toolchain, particularly Simulink Real-Time, allowed for rapid deployment of models directly from the Simulink interface and the ability to develop custom applications, such as visualization tools, directly in Simulink was

extremely convenient. Both systems arrived preconfigured with the necessary I/O channels required for the EcoCAR competition and thus, significantly reduced setup time. Finally, the robust hardware and industrial-grade construction, including rugged enclosures and built-in heat sinks, made the units reliable for both in-vehicle and lab-based testing environments.

Regarding networking and communication with componentry, as UWAF is integrating a custom controller into a production vehicle, there was a requirement to use the CAN-FD communication protocol as the majority of networking buses on the Cadillac LYRIQ used this format. Additionally, communication with components such as the drive units was also performed using CAN FD, whereas several team added components such as electrical insulation measuring devices or competition specific diagnostic CAN buses employed the traditional CAN networking protocol. Finally, for emergency disconnect systems, which allow for disablement of high voltage (HV) systems during occupant e-stop events or hazardous situations, team-defined digital signals were used to both indicate e-stop status as well as to control team-added contactors via a relay. While the selection of the communication protocol was mainly motivated by ease of integration with existing systems, the CAN and the extended CAN-FD protocol provide robust networking capacity, the ability to communicate rapidly without compromising signal transit time as well as being able to easily debug and troubleshoot both physical and electrical issues. Additionally, the physical convention of requiring CAN to be a set of twisted pair wires allowed System's Design and Integration (SDI) teams to easily route and integrate custom harnesses as needed, without the fear of compromising signal integrity due to EMI.

However, while the Speedgoat ecosystem provided an exceptional development experience, several limitations were identified during actual use. The Baseline M, though powerful, was not originally designed to act as a consolidated full-vehicle controller and is better suited for managing individual subsystems or features. As a result, certain integration tasks required additional software optimization and architectural restructuring to ensure all supervisory functions operated smoothly. Deployment times were occasionally long and, in some cases, models failed to deploy despite compiling successfully within Simulink, likely due to underlying communication or driver issues. Regardless, the computational performance of the Speedgoat units far exceeded UWAF's real-time processing requirements and this ensured that task execution overloads were rarely an occurrence. Alternatively, the decision to utilize a combination of CAN, CAN-FD and digital communication proved to be well-

suited to our application, offering the best balance of performance and compatibility with minimal trade-offs within the context of the EcoCAR competition.

4.1.3 Version Control Methodology Selection

For version control, UWAFt selected Git as the primary version control system, implemented through a private GitLab server hosted by the University of Waterloo. Git's distributed nature made it ideal for collaboration as it allowed multiple team members to work in parallel while maintaining full history and rollback capability. The university-hosted GitLab server also provided secure access, frequent system updates and robust backup functionality. This ensures that all proprietary software and sponsor-provided documentation remained protected within an approved institutional environment, addressing confidentiality requirements from industry sponsors. Additionally, GitLab offered built-in CI/CD tools and merge request management, easing collaboration and rapid preliminary validation. The choice of Git and GitLab was further motivated by their accessibility as it was both are open-source and automatically integrated with student records, allowing for easy access management coupled with widespread familiarity among students, to minimize the learning curve.

A few trivial issues did arise however with this selection, namely due to how Git and GitLab integrate with the decision to use MathWorks tool chains as the primary software development tool. Although Git handles text-based code efficiently, Simulink models are stored in a format that does not allow for easy revision comparison or merging, where auto-merging often worked for smaller changes but occasionally failed for larger mergers, when team integration conventions were not rigorously followed. This led to the necessity to perform manual conflict resolution, which can be quite unintuitive when dealing with graphical software languages and usually required secondary validation following merging.

UWAFt's PCM sub team also has several conventions for managing and creating branches within the team's repository:

- Branches are named using the following naming convention:
 - {Development Category}/{Branch_Type}/{Jira_Ticket_ID}/{Description}
- Development Category:
 - SCRPT: Development of Python and MATLAB scripts, machine learning and optimization algorithm development

- MODL: Longitudinal and lateral vehicle dynamic modelling (creation/building of model)
- CTRL: Control algorithm development
- QA: anything related to code quality improvements, automated testing
- Branch Type:
 - feature / bug, main, hotfix or stable
- Jira Ticket ID
 - Go to the associated task on Jira (further explanation in Chapter 4.4.3 and 5.1.2) that is tied to the branch of interest and use the ID in the top left corner of that task.



Figure 9 : Jira Ticket ID

4.1.4 Working and Planning Methodology Selection

From the start of the competition, UWAF T adopted an Agile-based working methodology to better manage the complexity and dynamic nature of student-led development across all sub teams. The team invests significant effort early in the competition year to define tickets, tasks, user stories and epics, creating a structured yet flexible framework for managing progress throughout the year. Each sprint begins with planning sessions where tasks were prioritized based on competition deadlines and subsystem dependencies and as progress advanced or was delayed, priorities were shifted or postponed to later sprints. Initially, the team relied on a predictive planning structure, where

milestones were rigidly scheduled and dependencies were defined in advance however, as the project matured, this approach evolved into a hybrid model that maintained long-term planning goals while incorporating sprints and sprint planning for short-term adaptability. This hybrid structure aligned well with the nature of the multi-year EcoCAR competition, where priorities and deliverables shift rapidly as vehicle integration, competition updates and testing constraints evolve month to month.

The Agile-hybrid methodology offers several advantages for UWAF's development process, such as allowing for quick realignment when unexpected hardware issues, software bugs or resource changes occurred. The sprint-based approach encouraged the importance of understanding task interdependencies as well as placing emphasis on focusing on a tangible goal achieved by completing a finite set of tasks. This ensures that even incremental improvements were captured and documented. However, the method also presented challenges, particularly in avoiding scope creep. Agile's flexibility can sometimes lead to overextension if priorities are not well defined, especially when new members join mid-cycle, and balancing long-term planning with short-term execution required continuous discipline and coordination among sub teams.

4.2 Generated Requirements and Justification

This section presents a selection of representative requirements from UWAF's RTM, which contains over 150 individual software functionality requirements for the PCM controller. The section will walk through the process used to generate and decompose these requirements from the diverse sources outlined in earlier sections, namely competition documentation, supplier specifications and internal team objectives. Each requirement will be accompanied by a justification explaining its relevance, origin and criticality to the team's control strategy where emphasis will be placed on demonstrating how these requirements were structured to ensure clarity, testability and alignment with both EcoCAR technical goals and safety expectations. By highlighting examples across categories such as diagnostics, communication, safety and system performance, this section aims to illustrate an applied example of the approach taken to formalize system expectations into actionable requirements that guided every stage of UWAF's supervisory controller design.

At the outset of the EcoCAR EV Challenge, competition objectives revolved around providing students with direct experience in advanced powertrain control, connected and automated vehicle technologies and the integration of energy-efficient systems within a production vehicle platform [41]. In alignment with these goals, UWAF prioritized pursuing a technically challenging

architecture to maximize learning outcomes while remaining competitive in end-of-year vehicle events [53]. During the architecture evaluation phase, the team selected an AWD configuration that improved upon several baseline LYRIQ metrics, namely acceleration performance (0–60 mph time) and overall vehicle efficiency. Subsequently, these vehicle-level targets were then translated into measurable requirements that guided controller development and calibration. Each performance metric was formalized into traceable requirements with the goal of ensuring that developments objectives could be quantitatively assessed and directly linked to control code behavior.

Table 9 : UWAF T Vehicle Technical Specifications

Vehicle Technical Specifications (VTS)				
Specification	Unit	Stock LYRIQ	Team VTS Targets	Simulated Performance
Accel. (IVM - 60 mph)	s	5.9	5	4.2
Accel. (50 - 70 mph)	s	3.9	2.9	2.9
Max. Lateral Accel.	g	0.81	0.78	0.78
Energy Consumption	Unadj. AC Wh/mi	243	267	260
Range	Adj. mi	250	225	248
Braking	m	44	47	45

In addition to performance-oriented requirements, several core functionalities were required to be recreated to enable fundamental vehicle operation which included systems such as contactor control and range/gear selector logic. Documentation supplied by GM [54] outlined expected signal behavior and other feature specifics, but functional interpretation and implementation were the responsibility of the team. These features were decomposed into testable functional requirements, defining the conditions, expected outcomes and safety constraints for each operational state.

Furthermore, through the completion of system-level safety analyses, including a Preliminary Hazard Analysis (PHA), the team identified potential failure conditions such as loss of braking and violations of battery current or voltage limits. These hazards were addressed through targeted diagnostic and fault mitigation requirements with the goal of being able to detect and safely manage these conditions during vehicle operation. Collectively, other sources similar to the ones that have just

been presented including, but not limited to, documentation from Magna, UWAF T’s drive unit supplier, helped to form the foundation from which the following example requirements were generated and tracked.

Table 10 : Sample Requirements for UWAF T PCM Controller

Requirement	Relevance	Origin	Criticality
The UWAF T LYRIQ shall accelerate from 0 to 60 mph within 5 seconds	Defines performance metrics that the vehicle must attain	Team architecture / simulations	Medium
The Power Moding Feature shall request the VICM to close the contractors on the RESS by transmitting Propulsion System Request as <ACTIVATE> less than or equal to 9.5ms after receiving System Power Mode <START REQUEST> and Propulsion System Status <OPERATIONAL>	Highlights transactional serial data behaviour for vehicle start up	LYRIQ Propulsion System Overview (LSPO)	High
Torque requested to the drive units shall be set to zero if CAN communication with the VICM is interrupted for 1000 ms or more	Defines remedial / safety action to be taken upon communication loss	PHA	High

It should be noted that the above requirements are only a sample of some of the requirements that were generated and have rapidly evolved throughout time, it should be expected that requirement change as new information is received. For example, initial requirements generated for the controller were strongly influenced by UWAF T’s team ethos and goals, developed during the early stages of the EcoCAR competition where the team prioritized a balanced focus between vehicle performance, energy efficiency and system safety. However, as the architecture matured, the emphasis increasingly shifted toward safety and hardware protection, ensuring that all high-voltage and powertrain systems operated within safe limits under all conditions. This included developing requirements for thermal

management, current limiting and fault detection to safeguard critical components. As a result, many of the team's guiding requirements were generated to meet performance and efficiency objectives while also aiming to ensure reliability and compliance with safety best practice, expected in both competition and industry contexts.

4.3 Controller Recap and Development Principles

This section provides a detailed overview of the PCM controller developed by UWAFt, outlining its architecture, subsystem structure and design rationale, with the goal of being able to replicate and extend the functionality of the stock vehicle controller that was removed. The custom controller enabled integration of a new AWD typology and custom automated control strategies within the competition vehicle. Here, this section will review how the PCM's architecture was organized, explain the purpose and interaction of its key subsystems and discuss how each contributes to overall system functionality. Additionally, this section will describe UWAFt's internal task generation process, integration workflow and deployment philosophy with the goal of highlighting the structured approach used to in the development phase of the controller's development lifecycle.

4.3.1 UWAFt PCM Controller Layout and Structure

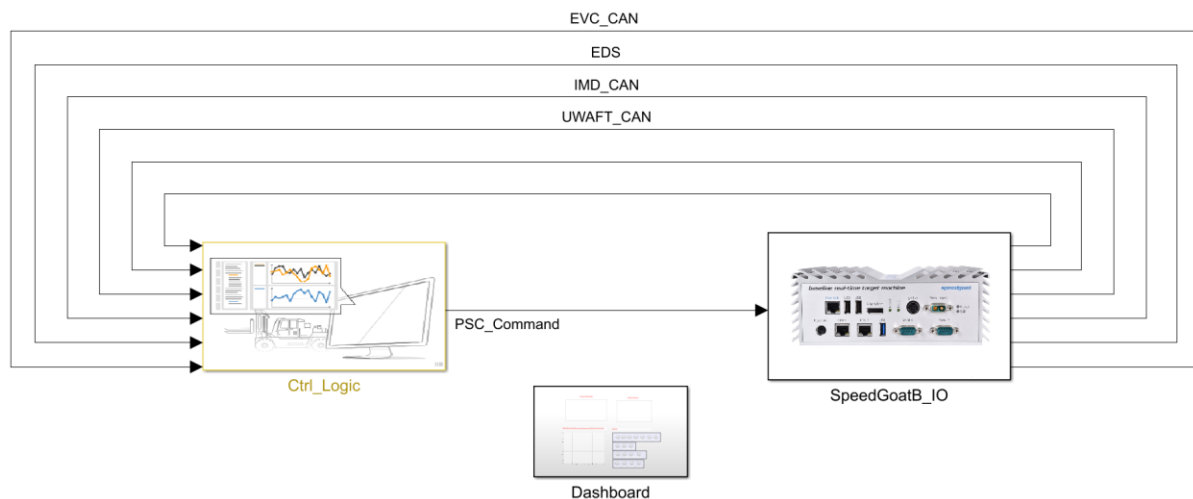


Figure 10 : UWAFt PCM Controller High Level Overview

Figure 10 illustrates the high-level software architecture of the UWAFt PCM controller, highlighting the clear separation between the control logic and communication layer. This structure establishes the foundation for easy deployment to the Speedgoat Baseline M unit while maintaining modularity,

allowing for easy testing and isolation of primary control logic. Each subsystem contains the essential functionality required for operation, yet remains compartmentalized with the goal of enhancing maintainability, parallel development and simplification of debugging. This allows for refinement of the primary “brains” of the controller without disrupting peripheral communication or interface modules.

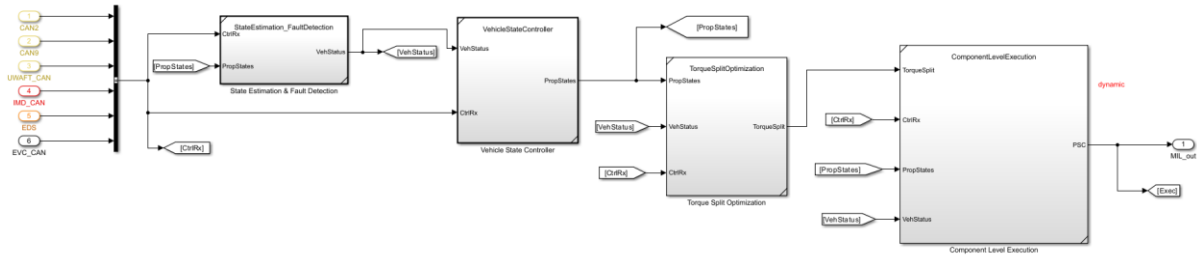


Figure 11 : Subsystem Separation and Functionality Scheme

Figure 11 provides an overview of the four main subsystems within the control logic block, namely the State Estimation and Fault Detection subsystem, the Vehicle State Controller, the Torque Split Subsystem and the Component Execution Block. These subsystems form the core of the controller’s decision-making framework with each block fulfilling a distinct role in succession. Subsequent sections provide a detailed discussion of each subsystem and its purpose within the supervisory control architecture.

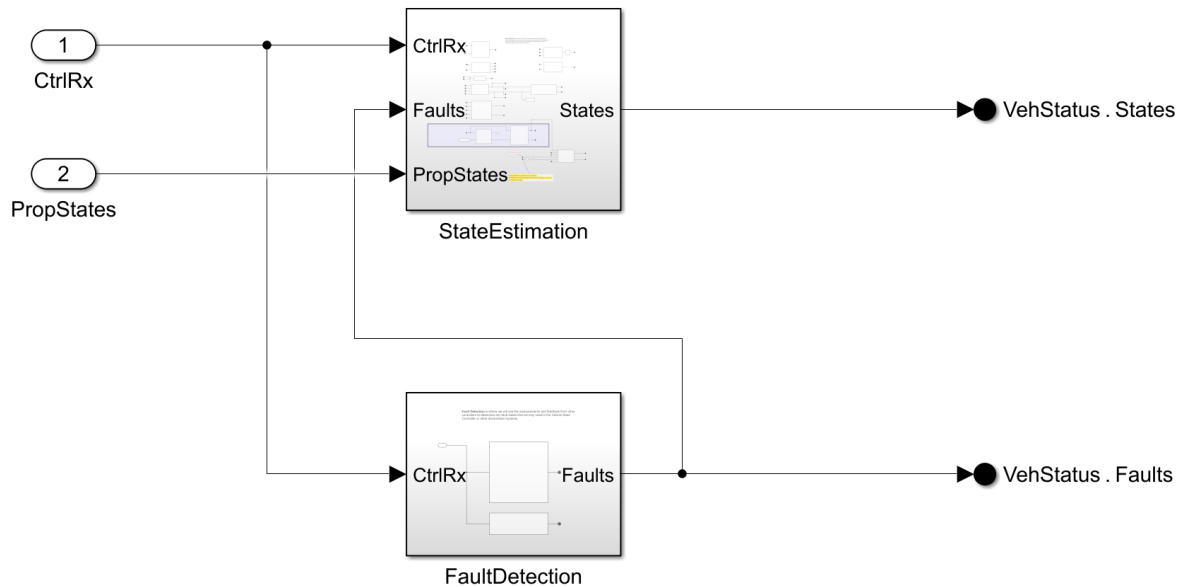


Figure 12 : State Estimation and Fault Detection Subsystems

Figure 12 shows how the State Estimation and Fault Detection block is further divided into two dedicated subsystems, corresponding directly to their respective functions. This separation allows for independent handling of real-time state calculations and diagnostic monitoring while preserving clear interfaces between the two domains. It should be noted that the Fault Detection block is purely for diagnostics and has been designed with the goal of creating the trigger for when remedial action logic must take over during entry into an undesirable vehicle state.

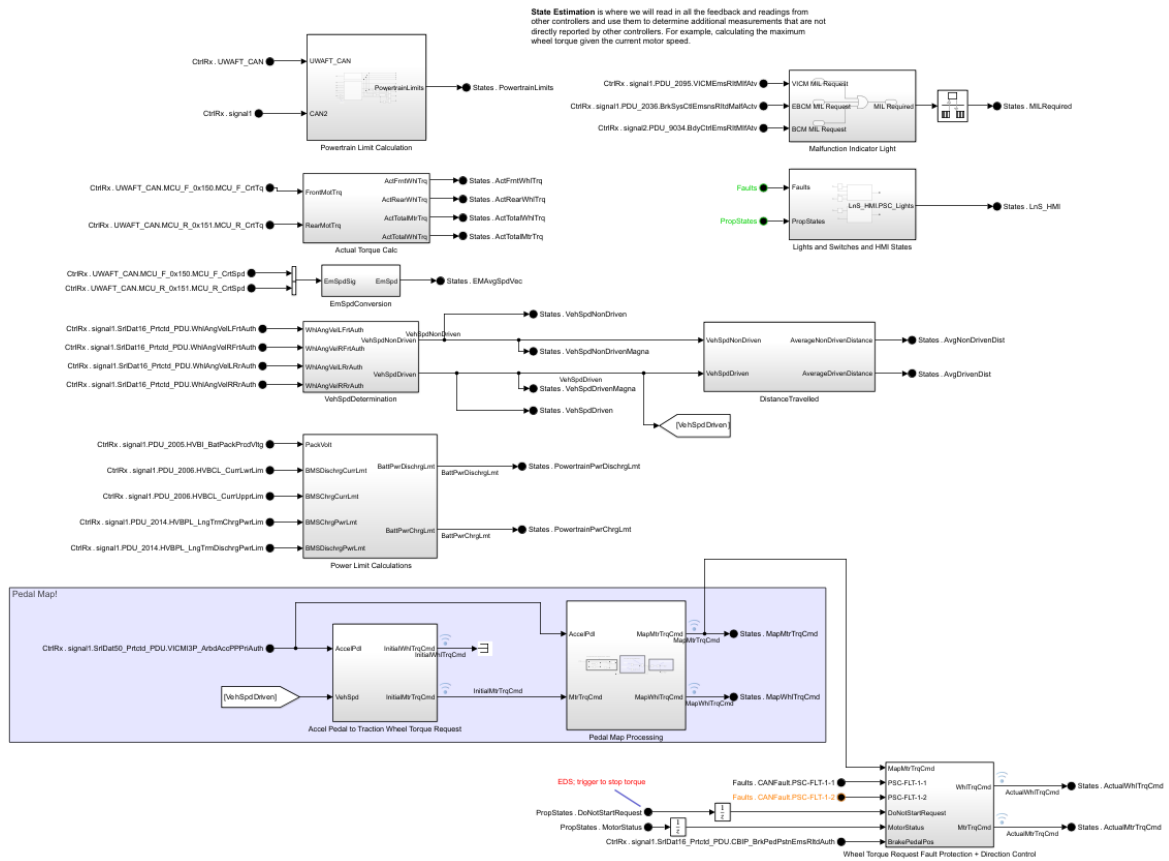


Figure 13 : State Estimation Subsystem Functionality

Figure 13 focuses on the State Estimation Subsystem, which consolidates key functionality related to signal processing and real-time computation. This block primarily executes combinational and sequential logic, where raw data is transformed through mathematical operations for downstream control use. It generates dynamic, time-sensitive outputs intended to support rapid decision-making processes within the rest of the controller. Typical operations include battery limit calculations, preliminary torque command shaping and parameter derivations such as vehicle speed estimation.

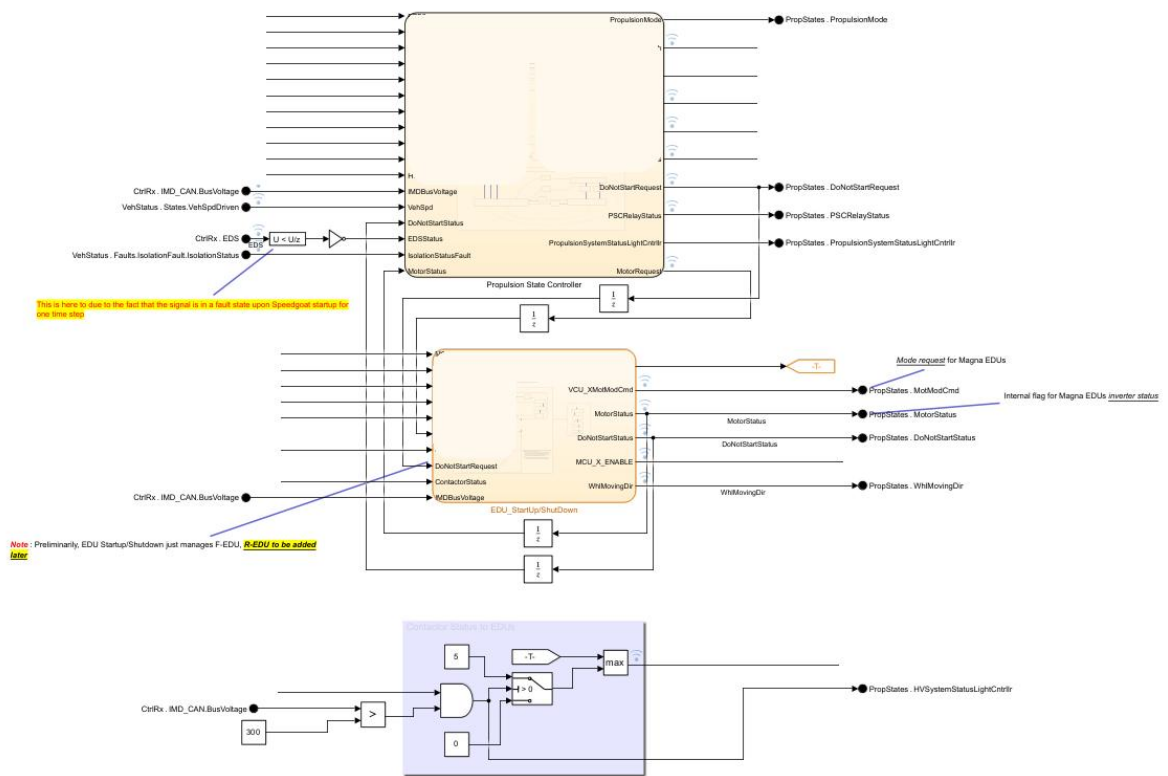


Figure 14 : Vehicle State Controller Functionality

Figure 14 presents the Vehicle State Controller Subsystem, responsible for managing all high-level vehicle state machine logic including propulsion system activation, range selection and direction control functionality. Although the logic here is less time-critical than in the state estimation block, it is essential for ensuring proper mode transitions and vehicle functionality. Additionally, these state machine features, while not directly safety-critical, form the backbone of user-driven functionality and guide the user experience. It should be noted that Figure 14 has been modified for confidentiality reasons.

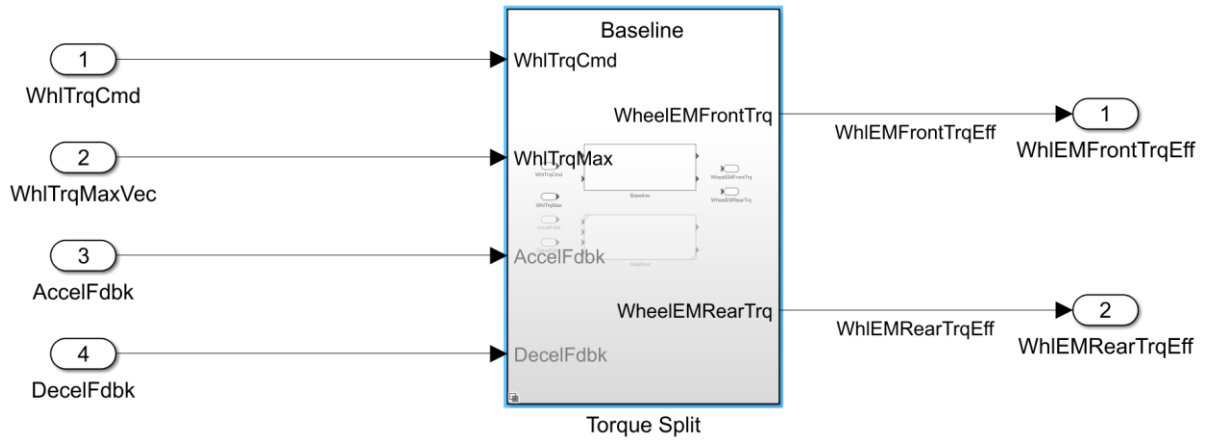


Figure 15 : Torque Split Subsystem

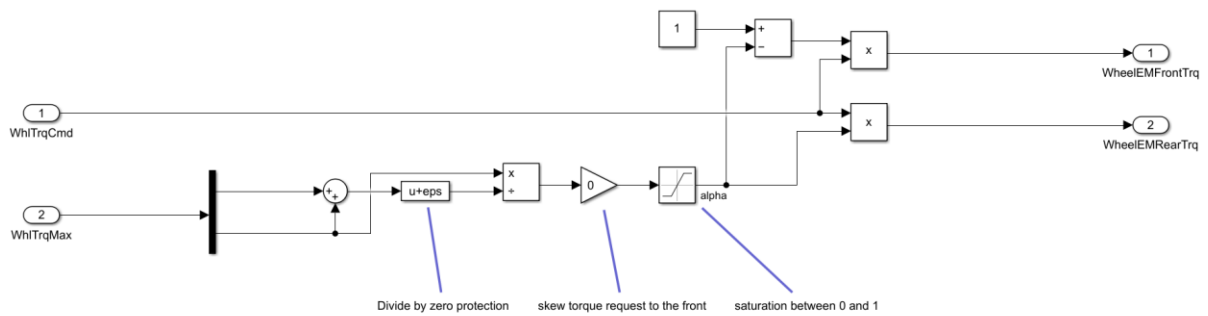


Figure 16 : Torque Split Logic

Figures 15 and 16 depict the Torque Split Subsystem, which divides the overall torque command between the front and rear drive units to support the team's AWD vehicle architecture. This subsystem ensures proper torque allocation based on vehicle torque limits and also allows for preliminary calibration in the event that it is desired that the vehicle operates in a configuration other than AWD.

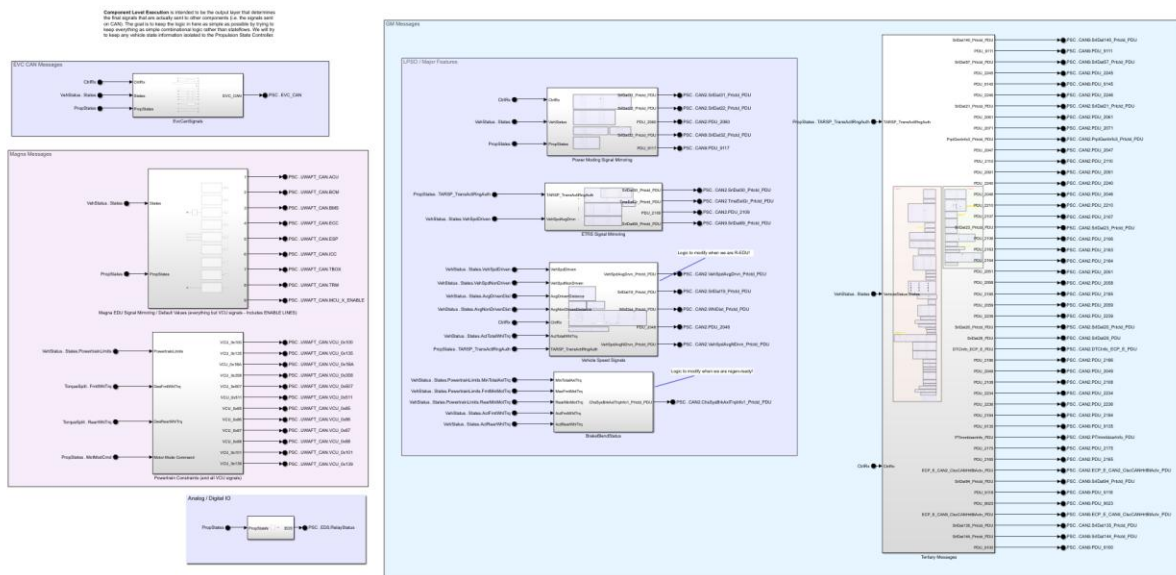


Figure 17 : Final Component Execution Subsystem

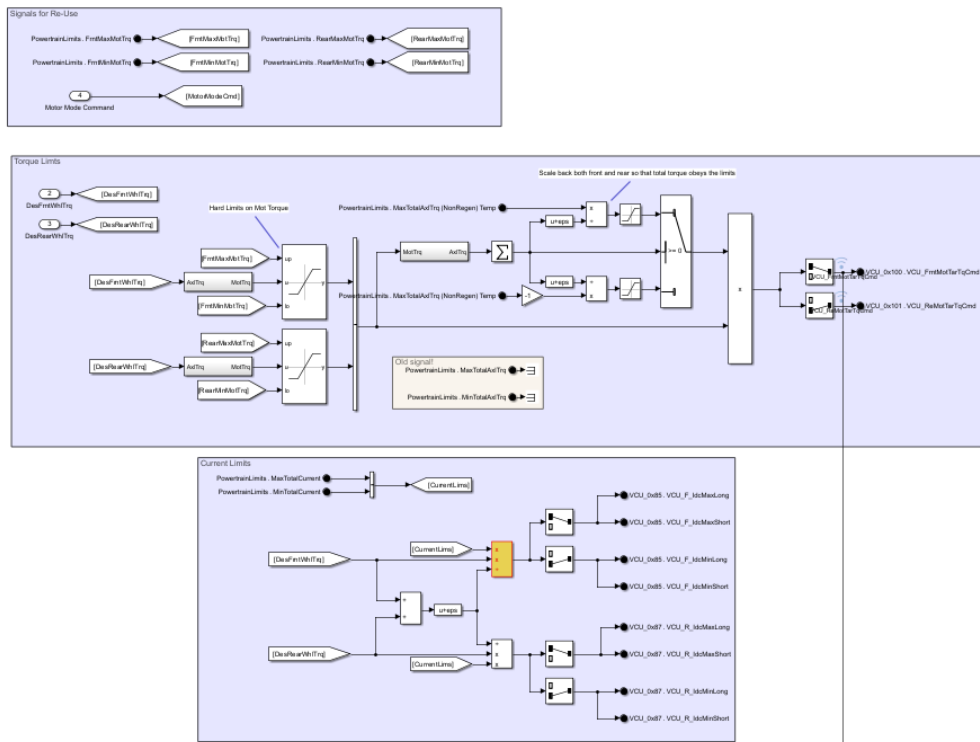


Figure 18 : Torque Limiting Logic within the Component Execution Subsystem

Figures 17 and 18 present the Component Execution Block, the final major subsystem within the control logic layer where this block serves as a gateway, consolidating and mapping all final generated signals to their corresponding CAN messages for transmission through the Speedgoat I/O modules. The logic here is intentionally minimal due to its primary focus being on ensuring signal integrity, applying final safety saturation limits and maintaining consistency with the vehicle's defined signal naming conventions.

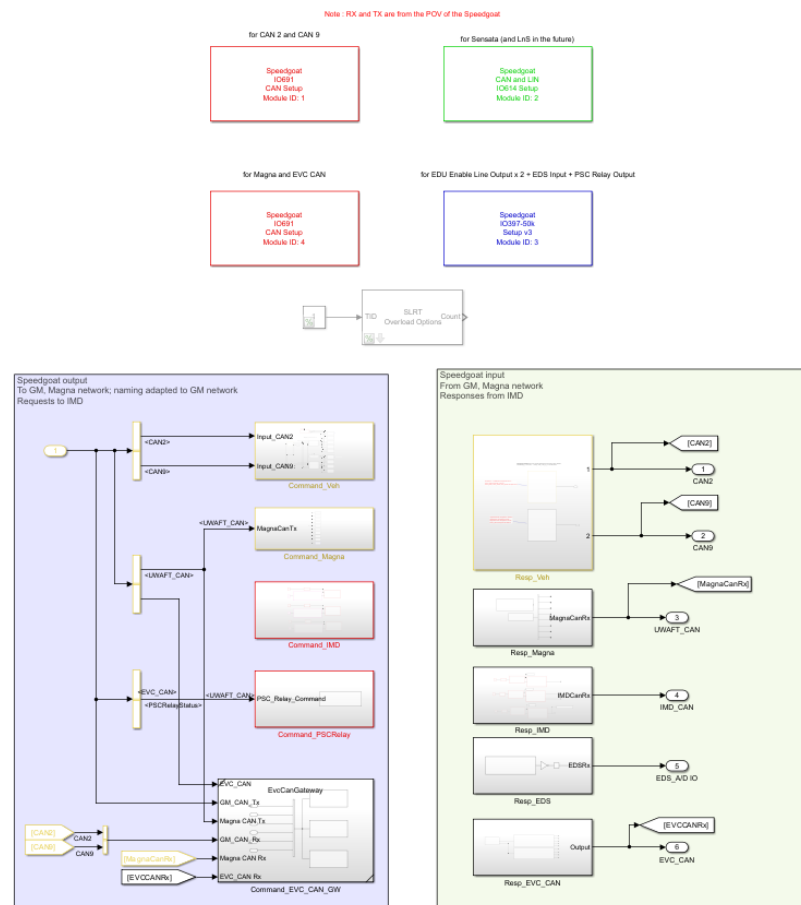


Figure 19 : Communications Layer Subsystem

Finally, Figure 19 illustrates the Communication Layer, which contains no control logic but manages all signal transmission and hardware configuration tasks. This layer defines hardware port parameters (such as CAN bit rates) and performs message and signal packing for transmission through the physical harnesses connected to the controller. It also includes specialized communication handling

logic unique to the GM CAN architecture, namely cybersecurity validation and handshake mechanisms, which are intentionally isolated from any performance-oriented vehicle control logic.

The developed controller architecture draws from a combination of pipes and filters, component-based and layered architectural principles with the goal of mimicking both the team's feature definitions and the nature of the system requirements / features to be developed. Certain requirements necessitated the manipulation and regeneration of signal data through mathematical transformations, such as converting individual wheel speeds into a unified vehicle speed, thus making the pipes and filters approach ideal for sequential data processing and transformation. However, not all functionality follows this linear flow. Many features, particularly those related to decision-making or system coordination, are better suited to a component-based architecture, allowing the team to compartmentalize related logic within isolated modules. By implementing a layered structure, where subsystems are consolidated hierarchically, the architecture supports scalability and maintainability. Finally, leveraging Simulink's model referencing system, each of the four main subsystems were developed as independent files, thus enabling clean separation of functionality and seamless integration with the team's established version control and collaboration conventions.

4.3.2 Task Generation

For UWAF, each task is derived from one or more requirements, transforming abstract system needs into concrete, measurable objectives that guide the engineering team's day-to-day work. In practice, task creation also took into account competition deliverables that may not be directly tied to vehicle functionality, such as technical reports, presentations and documentation submissions. Additionally, project milestones and event deadlines strongly influence task prioritization and scheduling. For example, anticipating the team's participation in the On-Road Safety Event (ORSE) at the year end competition where the vehicle requires dynamic range shifting while the vehicle is in motion, the team ensures that a requirement explicitly addresses this functionality and that an associated task is created, linked to the event and scheduled for completion within an upcoming sprint. At a high level, the expected deliverables define which features must exist, the requirements phase defines what each feature must accomplish, and the team's internal and competition deadlines determine how and when these tasks are executed. Figure 20 and 21 shows a snapshot of UWAF's task-tracking platform, Jira, where requirements, deadlines and sprints are systematically organized to ensure structured progress and accountability.

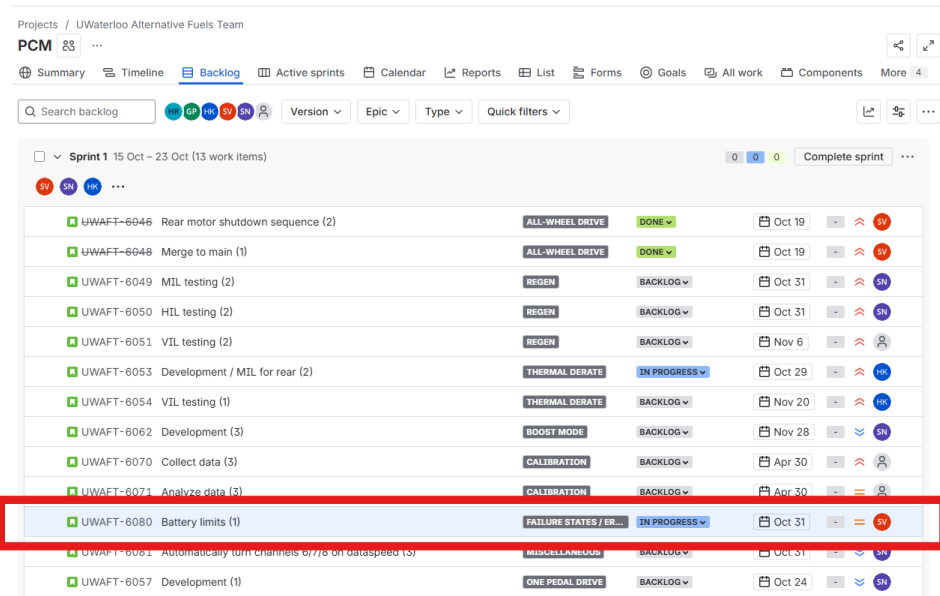


Figure 20 : Example Sprint Tasks in Progress; with related Epic, Current Status, Deadline, Priority and Assignee

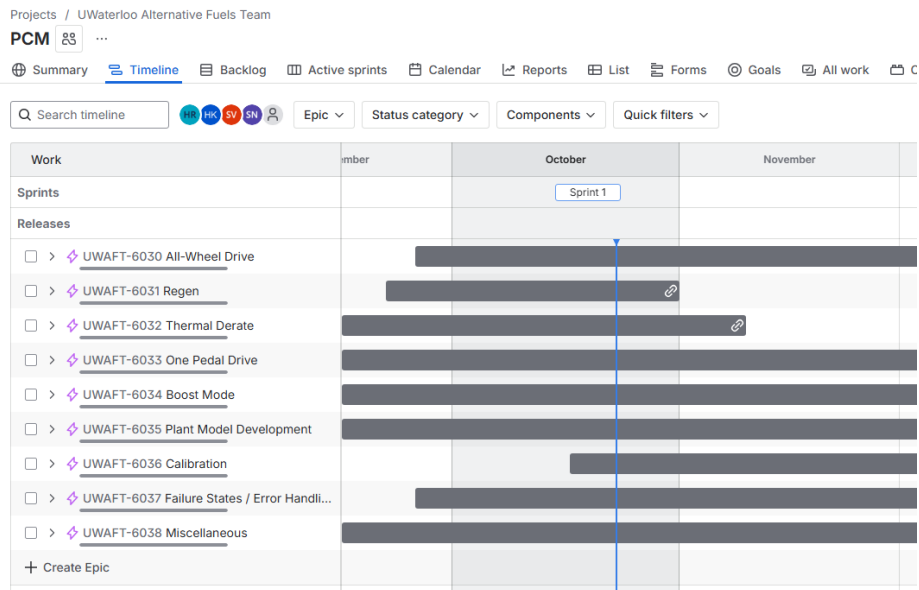


Figure 21 : UWAF6030 Gantt Chart

For example, Figure 20 highlights a representative task related to integrating battery limit violation logic within the vehicle controller. During its development phase, this task is explicitly tied to several

underlying requirements that define the expected system behavior with Figure 22 highlighting several of the generated requirements below.

Requirement ID	Requirement	Tracability
CARB REQUIRED Hardware Limits (Current, Voltage, Power, Derate)		
RESS		
PSC-PWR-4.2	The Powertrain Limit Calculation Feature shall report a total of 50% maximum current calculated from the long term power limits at the current voltage to the Magna Drive units.	LPSO 2.1.3.3.4
PSC-PWR-4.3	The Powertrain Limit Monitors Feature shall set a diagnostic flag and limit requested torque to the front and rear drive units to 0 Nm for at least 10 seconds when the total DC bus current exceeds 55% of the maximum long term discharge current limit.	LPSO 2.1.3.3.4
PSC-PWR-4.4	The Powertrain Limit Monitors Feature shall set a diagnostic flag and limit requested torque to the front and rear drive units to 0 Nm for at least 10 seconds when the total DC bus current exceeds 55% of the maximum long term charge current limit.	LPSO 2.1.3.3.4

Figure 22 : Battery Power Limit Sample Requirements

Recognizing that this feature will be externally reviewed by competition organizers during a scheduled presentation at the end of October, the team established an internal completion deadline of October 24th to allow sufficient time for review and verification. Furthermore, the task description clearly outlines the specific requirements it must satisfy and defines the associated success criteria with the goal of providing the assignee with full context on the work scope, testing expectations and completion urgency. It should be noted though that while not all tasks are directly linked to formal requirements, each is created to be clear and outcome-oriented, ensuring that its end objectives, whether that may be feature research or communication with an external stakeholder, are well understood.

4.3.3 Integration Methodology and Deployment Schedule

When performing integration, the team relies on a combination of resources and clearly defined responsibilities to ensure consistency and reliability throughout the integration process. Firstly, a collection of tutorials and reference videos is maintained internally to guide new and returning members through the Speedgoat deployment procedure, ensuring that knowledge is easily transferable and the learning curve remains minimal. Additionally, the team performs regular branch cleanups within GitLab to prevent repository clutter across development efforts, such that the repository remains lightweight and the PCM sub team knows what work is valid and what needs to be

discarded. To ensure accountability and maintain software integrity, the main integrator or merger is typically a team lead (most often the graduate lead) who has the capacity and flexibility to manage the entire integration process from start to finish, verifying that all new changes align with system requirements and software architecture conventions.

The team also strives to follow a structured process during deployment to ensure consistent and successful integration where software is deployed to hardware at distinct milestones. After HIL testing of a feature set, the final software is always deployed to the Speedgoat hardware prior to transitioning to VIL testing, with the goal of minimizing the risk of bulk software updates and subsequent hardware bugs that could fail during vehicle-level validation. Each feature is only integrated into the main branch once it has been successfully tested in VIL, ensuring that all changes entering the primary codebase are verified on real hardware. The main branch also aims to serve as the most stable and validated version of the software with the goal of acting as a reliable fallback if subsequent development branches fail or render the vehicle unsafe.

The RTM is also used as a central documentation tool for tracking development, integration and validation progress. It includes the testing status of each requirement, indicating whether functionality has been verified in MIL, HIL or VIL environments while also logging the software version and branch associated with each test. This structured documentation allows the team to review at a glance, the current development progress as well as being able to identify untested or partially validated features.

Implementation	Implementation Status	Test Case ID	MIL Test Status	HIL Test Status	VIL Test Status	SW Version / Branch	Notes
Propulsion State Controller -> ETRS Stateflow	Complete	T.1	Pass	Fail	Not Started	1.85 / main	Failure due to issues with real
Propulsion State Controller -> ETRS Stateflow	Not Implemented		Not Started	Not Started	Not Started	1.85 / main	
Propulsion State Controller -> EDU Stateflow	Complete	T.1	Pass	Fail	Not Started	1.85 / main	Failure due to issues with real time set up
Propulsion State Controller -> EDU Stateflow	Complete	T.1	Pass	Fail	Not Started	1.85 / main	Failure due to issues with real time set up
Propulsion State Controller -> EDU Stateflow	Complete	T.1	Pass	Fail	Not Started	1.85 / main	Failure due to issues with real time set up

Figure 23 : RTM Requirement Implementation and Validation Status

4.4 Testing Framework

UWAFIT's testing framework has been carefully developed and continuously refined to enable both efficient simulation-based validation and reliable, safe vehicle-level testing. The framework, which will be presented in the following sections, emphasizes structured test planning and consistent documentation of results to ensure transparency and repeatability across all stages of development. A strong focus is placed on maintaining objective, data-driven evaluation processes while supporting rapid iteration as new features are introduced. By standardizing procedures and reusing well-defined workflows, the team ensures that each test produces reproducible and meaningful outcomes, ultimately enhancing confidence in system performance and safety.

4.4.1 Virtual Testing / Simulation

Virtual testing forms the beginning of UWAFIT's testing framework, serving as the initial stage of software validation where this stage is primarily conducted within two virtual environments. The first stage is MIL testing, where the Simulink Test application, as highlighted in Chapter 4.1.1, acts as the primary validation tool. Using Simulink Test, the team isolates a specific subsystem, primarily the "Ctrl_Logic" subsystem shown in Figure 10, within a dedicated test harness. This harness is unique to the Simulink environment and is simply a file that separates the control logic from the adjacent communication layers with the goal of removing unnecessary code as well as streamlining the testing process. A reference to the actively developed model is maintained, such that any updates made within the development branch are automatically synchronized to the test harness to ensure that testing is performed with the latest code, without the need for manual reconfiguration. All inputs and outputs of the control logic are explicitly exposed and when necessary, organized into signal groups (also known as buses) in accordance with UWAFIT's development conventions. This structure allows testers to easily inject custom input data and observe system responses with full visibility, to effectively evaluate the controller's behavior while treating it as a functional "black box." (commonly known as "block box" testing [55])

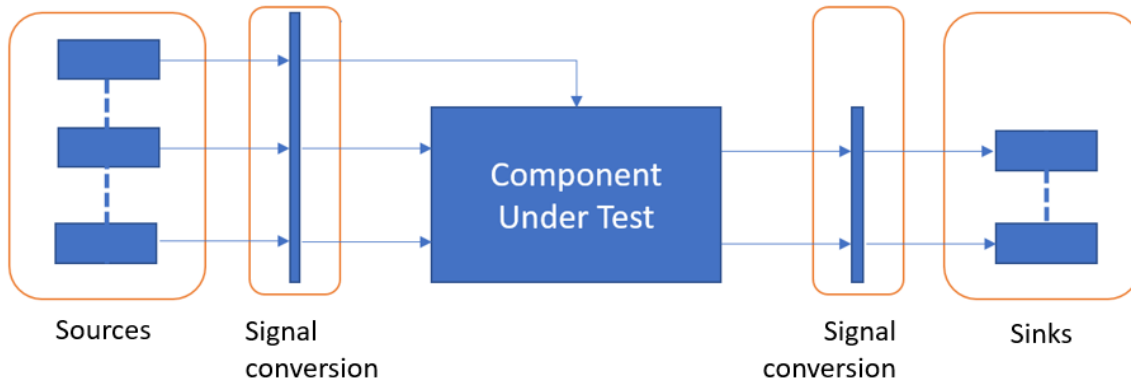


Figure 24 : Simulink Test Harness Visualization

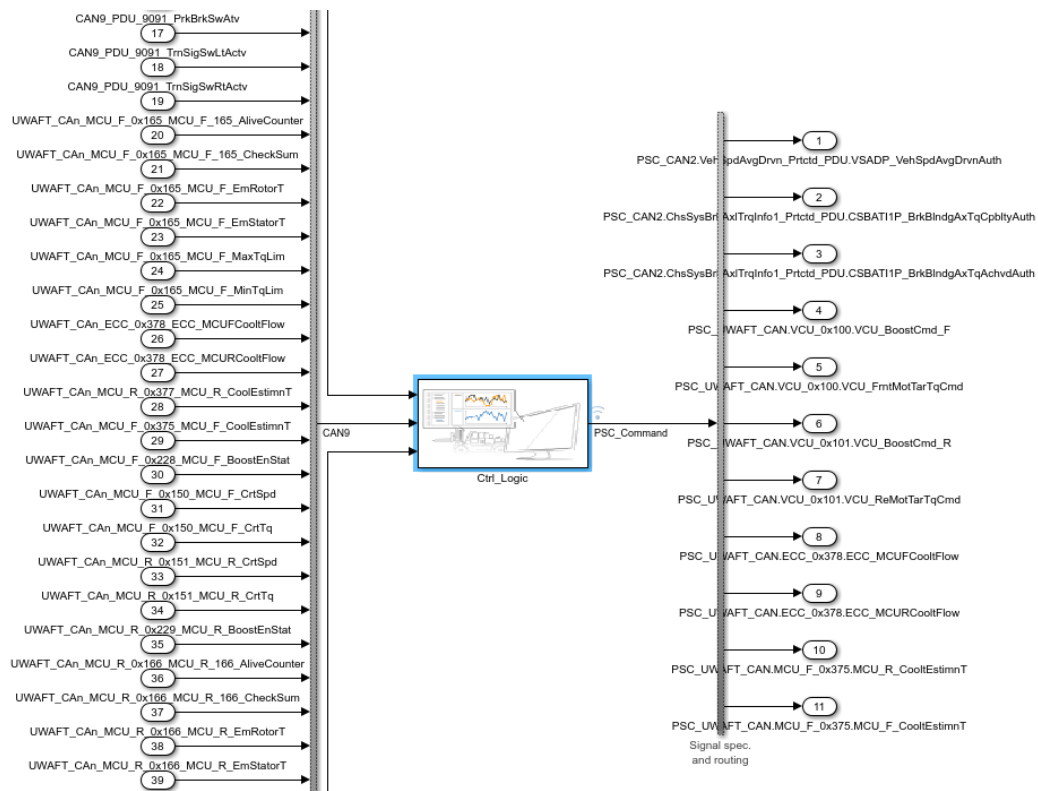


Figure 25 : UWAF T Control Logic Test Harness

Once the test harness is established, test case generation involves translating system requirements into temporal assessments that can be evaluated within Simulink Test during simulation. Unlike traditional software testing, where a test case explicitly defines both the input data and the expected output, the MathWorks toolchain instead focuses on defining the expected behavior of the control

logic under specific operating conditions. Essentially, each test case represents a requirement expressed as a measurable behavioral criterion. As shown in Figure 28, requirements with unique IDs from the UWAFRTM are directly linked to corresponding assessments within Simulink Test where these assessments, written in plain language, are conditional statements referencing variables and signals within the test harness. During simulation, each assessment is continuously evaluated to ensure that the control logic maintains compliance with the defined behaviour.

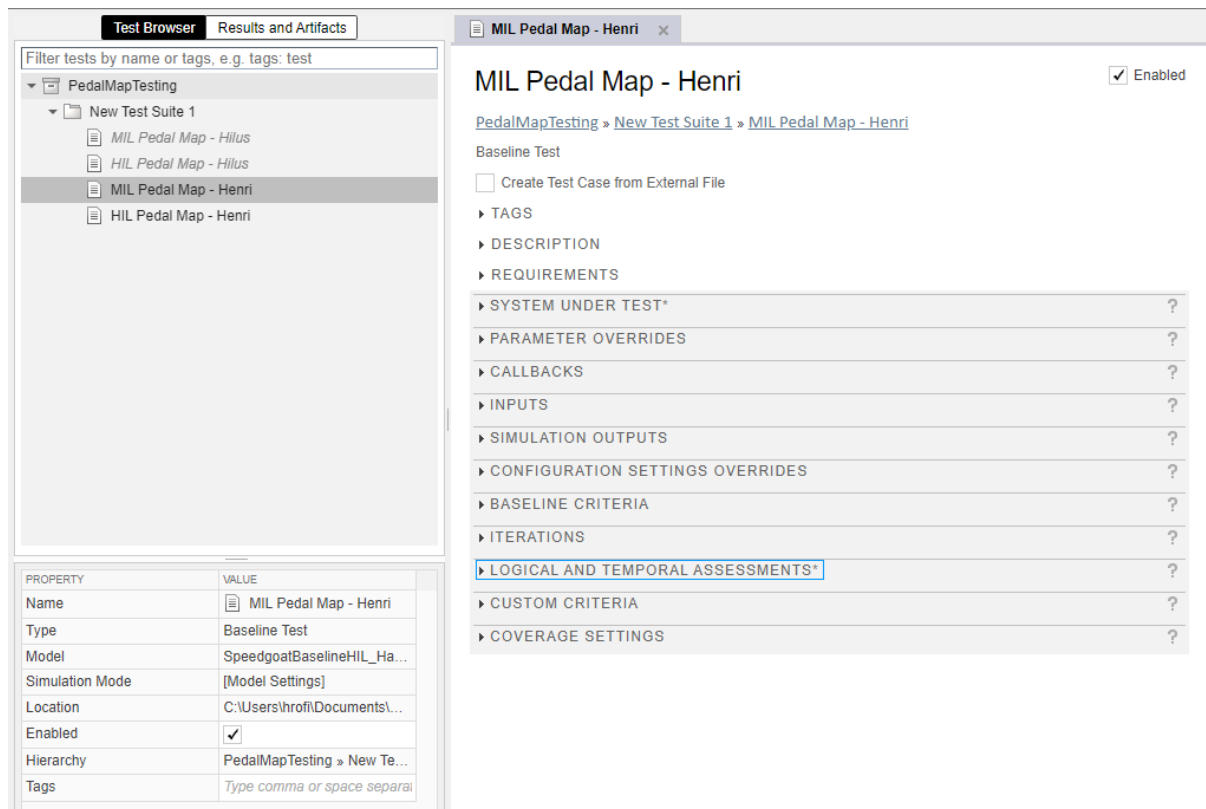


Figure 26 : Simulink Test Manager Home View

ENABLED	NAME	ASSESSMENT
☒	PSC-DYN-1.1	▼ At any point of time ... ▼ bounds-check-pattern: always less than signal: WhlTrqCmd upper-bound: 2500 upper-bound-type: less than or equal to
☒	PSC-DYN-1.2	► At any point of time, AccelPdl must be greater than or equal to 0 and less than or equal to 100
☒	PSC-DYN-1.6	► At any point of time, whenever AccelPdl == 0 & BrakePdl > 0 is true then, with a delay of at most 0.5 seconds, WhlTrqCmd == 0 must be true
☒	PSC-DYN-1.5	► At any point of time, whenever VehSpd ~= 0 & AccelPdl ~= 0 is true then, with a delay of at most 0.5 seconds, InitialWhlTrqCmd ~= 0 must be true

Figure 27 : Example Temporal Assessments for Pedal Map Test Case

The data injected into the control logic during simulation typically originates from recorded vehicle data which is converted into a format compatible with Simulink. In practice, after completing a drive cycle or identifying an undesirable event or condition during on-road testing, a log of all vehicle CAN traffic is recorded and exported from the vehicle's data logging device. This log is then imported into the MATLAB workspace for use within the Simulink test harness, where a custom-developed conversion script processes the raw signal data. The data is formatted it into arrays of type "double", suitable for simulation. This workflow enables the injection of realistic, time-synchronized vehicle data into the controller such that is possible to replicate conditions that cannot be easily recreated using basic Simulink blocks. By replaying these authentic drive scenarios, the team can efficiently diagnose and debug undesired control behavior in an offline environment, with full visibility into all available vehicle signals. However, this should not undermine the power of static data injection (ie : constants of realistic magnitude but missing transient behavior) into software and is used frequently in the absence of relevant data logs for software testing.

```

UWAFT_Torque_Chatter_CANInportScript.m
1  canDB = canDatabase("GM_GB_DWCAN2.dbc");
2  blfData = blfread("UWAFT_CarShakingIssue_Modified.blf",2,"Database",canDB);
3
4  canDB2 = canDatabase("FM29_INVBUS_Matrix_CANFD_V390.8_20230524.dbc");
5  blfData2 = blfread("UWAFT_CarShakingIssue_Modified.blf",4,"Database",canDB2);
6
7  timeOffset = 1200;
8
9  %% GM
10 signalTimetable1 = canSignalTimetable(blfData, "PDU_2005");
11 signalTimetable2 = canSignalTimetable(blfData, "PDU_2006");
12 signalTimetable3 = canSignalTimetable(blfData, "PDU_2014");
13
14 signalTimetable4 = canSignalTimetable(blfData, "SrlDat50_Prtctd_PDU");
15 signalTimetable5 = canSignalTimetable(blfData, "SrlDat16_Prtctd_PDU");
16 signalTimetable10 = canSignalTimetable(blfData, "SrlDat26_Prtctd_PDU");
17
18 %% Magna
19
20 signalTimetable6 = canSignalTimetable(blfData2, "MCU_F_0x150");
21 signalTimetable7 = canSignalTimetable(blfData2, "MCU_R_0x151");
22 %signalTimetable8 = canSignalTimetable(blfData2, "MCU_F_0x355");
23 signalTimetable9 = canSignalTimetable(blfData2, "MCU_F_0x165");
24
25 % HVBatPackPrctdVltg = signalTimetable1.HVBatPackPrctdVltg;
26 % HVBCL_CurrLwrLim = signalTimetable2.HVBCL_CurrLwrLim;
27 % HVBCL_CurrUprrLim = signalTimetable2.HVBCL_CurrUprrLim;
28 % HVBPL_LngTrmChrgPwrLim = signalTimetable3.HVBPL_LngTrmChrgPwrLim;
29 % HVBPL_LngTrmDischrgPwrLim = signalTimetable3.HVBPL_LngTrmDischrgPwrLim;
30
31 %% placeholder signals below
32
33 VICMI3P_ArbdAccPPPrAuth = [seconds(signalTimetable4.Time)-timeOffset, signalTimetable4.VICMI3P_ArbdAccPPPrAuth]; %%
34 WhlAngVelLFrtAuth = [seconds(signalTimetable5.Time)-timeOffset, signalTimetable5.WhlAngVelLFrtAuth];
35 WhlAngVelRFrtAuth = [seconds(signalTimetable5.Time)-timeOffset, signalTimetable5.WhlAngVelRFrtAuth];
36 WhlAngVelLRrAuth = [seconds(signalTimetable5.Time)-timeOffset, signalTimetable5.WhlAngVelLRrAuth];
37 WhlAngVelRRrAuth = [seconds(signalTimetable5.Time)-timeOffset, signalTimetable5.WhlAngVelRRrAuth];
38 CBIP_BrkPedPstnEmsRltdAuth = [seconds(signalTimetable5.Time)-timeOffset, signalTimetable5.CBIP_BrkPedPstnEmsRltdAuth];
39

```

Figure 28 : MATLAB CAN Data Import Script

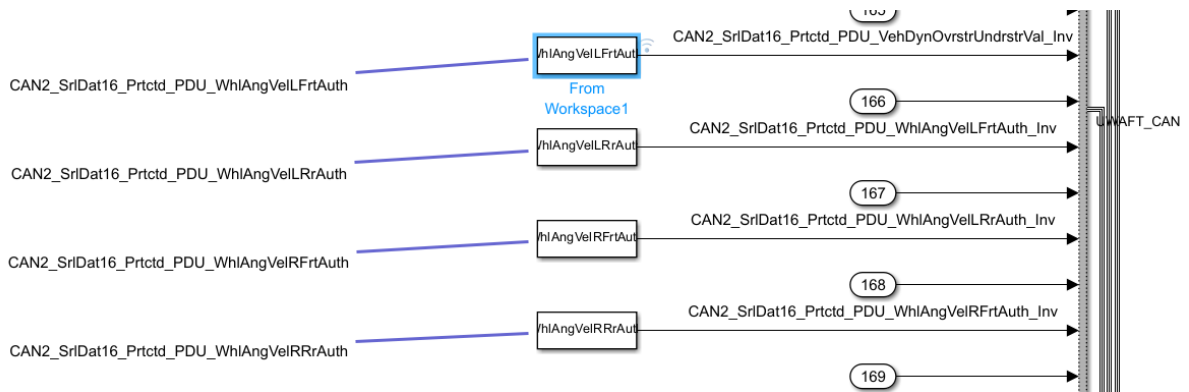
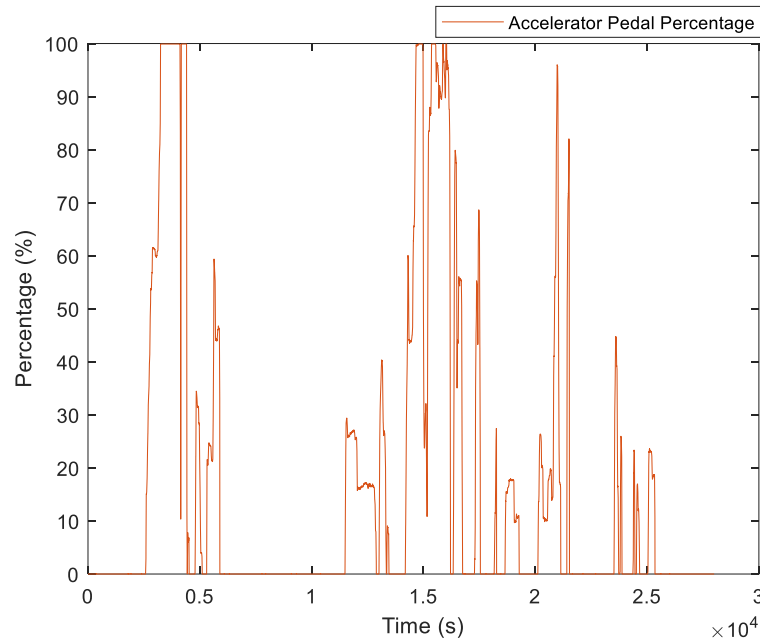


Figure 29 : Testing Data Source / Generation Within Simulink Test Harness



Once the test harness, test cases and input data have been prepared, the simulation is executed through the Simulink Test Manager, which commands Simulink to evaluate all defined signals and verify whether each assessment passes or fails over the duration of the run. A clear and comprehensive test report is automatically generated upon completion and provides a summary of results and detailed visualizations of when assessment conditions are met, how they are evaluated and where failures occur. These reports can also be exported for sharing with the larger engineering team or analysis at a later time. Following review, the developer then proceeds with debugging and rectifying the error, focusing on identifying which assessments failed and the underlying cause. Solving the bug typically involves adjusting the control logic, but in some cases may require re-evaluating the validity or definition of the temporal assessments themselves. This is because Simulink verifies test case compliance rather than the feasibility of the requirement being tested.

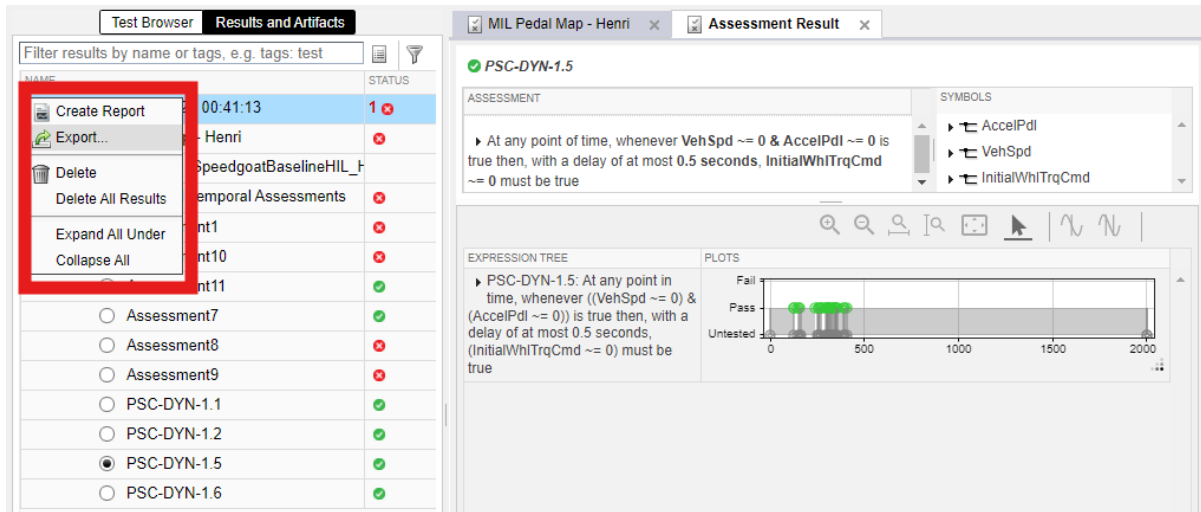


Figure 31 : MIL Testing Report Generation

HIL testing represents the second and final stage of UWAFT’s simulation-based validation process and is used to evaluate the real-time feasibility and performance of the control logic when executed on actual hardware. In the HIL environment, the isolated controller logic, contained within the test harness, is deployed directly onto the Speedgoat Baseline M unit and the input data used to stimulate the controller is generated by the Speedgoat Performance system. The performance unit ideally transmits the same data used within MIL testing, to retain testing consistency, through physical harnesses connected to the Speedgoat Baseline M’s I/O ports. This configuration allows for realistic, open-loop testing under controlled conditions, ensuring that timing behavior, signal integrity and computational performance meet operational requirements prior to VIL testing.



Figure 32 : Speedgoat Baseline M Controller Running Control Logic for HIL Testing



Figure 33 : Speedgoat Performance Unit Injecting Open Loop Testing Data for HIL Testing



Figure 34 : UWAFTHIL Setup with Speedgoat Unit Communication

For HIL testing, two distinct models are required, one for each Speedgoat unit, in comparison to the single test harness workflow used in MIL testing. This separation allows the Speedgoat Performance system to emulate vehicle behavior while the Speedgoat Baseline M executes the control

logic in real time. Similarly to MIL testing, the input data originates from recorded CAN logs, which are imported into the MATLAB workspace for use within Simulink however, in this setup, the data input must be isolated from the control logic. Furthermore, Speedgoat I/O blocks are incorporated into both models to individually define hardware configurations and communication parameters, ensuring that each real-time target machine can effectively exchange signals through their physical I/O interfaces. This is done with the goal of accurately replicating real-world communication considerations such as the Nyquist stability criterion which dictates that the sample time of a sensor should be at least double the highest frequency component of that signal [56].

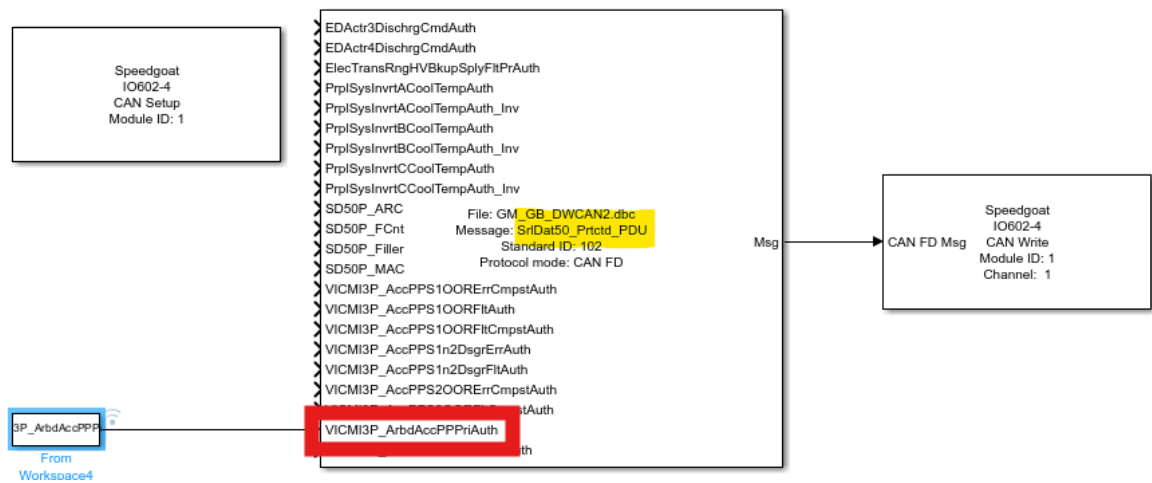


Figure 35 : HIL Testing Data Injection Model for Speedgoat Performance Unit

HIL Pedal Map - Henri

[PedalMapTesting](#) » [New Test Suite 1](#) » [HIL Pedal Map - Henri](#)

Real-Time Test (Baseline Test)

► TAGS

► DESCRIPTION

► REQUIREMENTS

▼ SYSTEM UNDER TEST*

Load Application From

Model

Model:

SpeedgoatBaselineHIL_Harness_PedalMapHIL

Target Computer:

UWAFT_Baseline

► TEST HARNESS

► SIMULATION SETTINGS OVERRIDES

► PARAMETER OVERRIDES

▼ CALLBACKS*

▼ PRE-LOAD*



► Runs before the model loads and any model callbacks

```
1 model = 'SpeedgoatPerformance_PedalMapHIL';
2 open_system(model);
3 slbuild(model);
4 tg = slrealtime('UWAFT_Performance');
5 load(tg,model);
6
```

► POST-LOAD



► PRE-START

REAL-TIME APPLICATION*



Figure 37 : Modified HIL Test Case and Callback Script

In the future, expanding the testing workflow’s functionality and versatility will primarily involve broadening the range of input data sources used during validation, which can be achieved by incorporating additional I/O blocksets or virtual data feeds from the MATLAB Workspace. This ensures that all relevant inputs to the control logic, beyond just the primary signals, are fully populated. Once this infrastructure is in place, modifying test scripts to reference different data logs will enable the evaluation of a wider variety of operational scenarios. Furthermore, to enhance HIL testing accessibility, particularly during off-site testing, data inputs could be sourced directly from real-time CAN playback, physically being performed using in-vehicle loggers. This approach would eliminate reliance on pre-recorded data files, allowing developers to reproduce and diagnose issues immediately in the field when access to the Speedgoat Performance unit is limited.

A distinguishing aspect of UWAFT’s testing framework is the absence of SIL and PIL testing which are two stages typically emphasized in control system validation, as outlined in Chapter 3.4.2. The omission of these environments is intentional and stems from the team’s adoption of a MBD

philosophy, which inherently reduces the need for these intermediate testing stages. By leveraging the MathWorks toolchain, particularly Simulink's robust and highly reliable automatic code generation capabilities, the team effectively eliminates the need to retest controller logic once converted to C code, which makes SIL testing redundant. Similarly, PIL testing is unnecessary within UWAF's workflow due to the integration of Speedgoat hardware in the HIL workflow, which has been purposefully built by Speedgoat with the express intent of being computationally faster than what is required for the EVC, with the goal of futureproofing against hardware performance bottlenecks. As a result, any validation tasks typically associated with PIL, namely identifying timing discrepancies or latency issues during hardware deployment, are inherently addressed during HIL testing.

4.4.2 Non-Virtual Testing

Following the completion of virtual testing, VIL testing is conducted as the final validation stage, requiring authorized team members to operate the vehicle that has been equipped with the validated software on a closed-course testing facility. Prior to testing on a closed course, UWAF typically tests for code functionality and performs preliminary safety checks on the LYRIQ while it is immobilized, which is accomplished by raising it on a vehicle lift. UWAF has access to several such test sites, with their layouts summarized in Figures 38 and 39. Each VIL session is guided by a comprehensive test plan that details specific maneuvers, procedures and safety protocols to ensure efficient and controlled testing. These plans include equipment checklists (e.g., radios, cones, lighting), as well as defined personnel responsibilities to guarantee structured and safe operation. Most importantly though, is the collection of data and the prioritization of logging and highlighting various signals of importance based on the test plan being executed. Additional information on test plan formatting and preparation is provided in Section 4.4.3.



Figure 38 : Waterloo Region Emergency Services Training and Research Centre Track

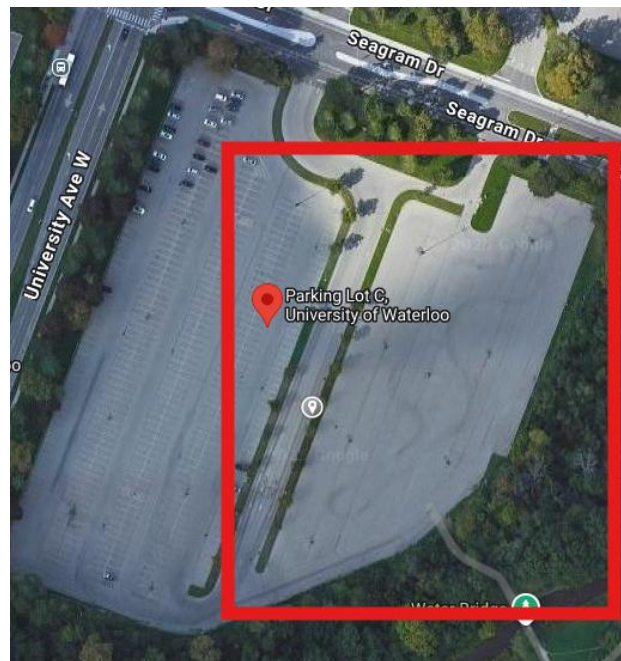


Figure 39 : University of Waterloo Parking Lot C Testing Area

The primary purpose of VIL testing within UWAFt's workflow is to validate software performance under real-world conditions, confirming correct interaction between the developed controller and various vehicle subsystems that are difficult to replicate in simulation. Equally

important is the fact that VIL testing serves as a validation mechanism for the team's MIL and HIL workflows since the team can use real, logged vehicle data to confirm simulation accuracy and refine earlier testing assumptions. This cyclical validation process ensures that simulation-based testing remains representative with the goal of increasing confidence within UWAF's overall control validation methodology.

4.4.3 Planning and Tracking Tests

Test plans play a crucial role in UWAF's VIL testing framework, not only guiding on-vehicle validation but also helping define the types of scenarios that must be considered and replicated during simulation. Each UWAF test plan follows a standardized template requiring several key sections to be completed, ensuring consistency and completeness across all test activities. These sections collectively address every stage of the testing process including test preparation, execution and post-test data analysis:

1. Prerequisites / Item List
2. Test Description
3. Required Signals of Interest for Review / Visualization
4. Procedure
5. Post Processing Instructions
6. Measures of Success
7. Revision Table

All finalized test plans are stored within a dedicated SharePoint directory, accessible to all technical leads via Microsoft Teams, with filenames following a clear and standardized naming convention for quick identification of purpose and scope. Collected data from each test is archived on the team's secure server, accompanied by a tracking table summarizing the relevance of each file and any notable observations made during data collection. For simulation testing, the activity of MIL and HIL testing is typically completed shortly after feature development to maintain rapid iteration whereas for VIL testing, this typically requires more long-term planning as UWAF shares access to test facilities with other stakeholders. To ensure availability, testing days must be scheduled at least

two weeks in advance, with coordination and approval from the appropriate facility partners (e.g., Branca Da Costa at the University of Waterloo).

PCM > Test Plans > VTS Tests

Name	Modified	Modified By	Created
GM MATT Test Plan Revisions	November 14, 2023	Henri Rofiq	November 14, 2023
VTS - 50-70 Acceleration Test Plan - UWAFT.docx	November 14, 2023	Henri Rofiq	October 4, 2023
VTS - 60 MPH Acceleration Test Plan - UWAFT.docx	July 1	Henri Rofiq	September 21, 2023
VTS - Braking Distance Test Plan - UWAFT.docx	November 14, 2023	Henri Rofiq	September 21, 2023
VTS - Lateral Acceleration Test - UWAFT.docx	November 14, 2023	Henri Rofiq	October 4, 2023

Figure 40 : Test Plan Management

Track Log Tracker Table

By Henri Rofiq 4 min 7 Add a reaction

Below are tables organizing data logs by track testing day. Headers for each table must be identical to server folder names containing the actual logs for that specific day. Please include ANY and ALL notes for each measurement so as to allow ease of viewing and reference in the future. Any questions? Please reach out to the PCM leads!

Logs are located here : Z:\PCM\Data Collection

- Baseline Vehicle Evaluation
 - Oct 28 - Waterloo Test Track
 - Oct 29 - Waterloo Test Track
 - Nov 2 - GM Test Track
 - Nov 3 - GM Test Track
 - Nov 5 - Waterloo Test Track

Baseline Vehicle Evaluation

(Z:\PCM\Data Collection\BaselineVehicleEvaluation\Track Testing)

Server Path of Parent Folder	File Name	What was collected / what did we try to collect / what was done?	Any special notes? (pass / fail, difficulties, etc...)
Z:\PCM\Data Collection\BaselineVehicleEvaluation\Track Testing\Oct 28 - Waterloo Test Track\BrakingDistance Logs_2023-10-28\2023-10-28_13-55-22	Measurement_1 → Measurement_13	Braking Distance (60 - 0 mph; full braking test)	N/A
Z:\PCM\Data Collection\BaselineVehicleEvaluation\Track Testing\Oct 28 - Waterloo Test Track\CruiseControl_2 023-10-28\2023-10-28_14-08-25	Measurement_1	Cruise Control Testing	N/A

Figure 41 : Data Log Info Tracker Table

4.4.4 CI/CD and Test Automation

Unfortunately, UWAF's CI/CD pipeline and testing automation strategies have not yet reached the level of maturity originally envisioned in previous competition years, primarily due to resource limitations, both in terms of computational availability as well as personnel for setup and maintenance. Currently, the team has a functioning GitLab runner configured to trigger a CI/CD process whenever code is pushed to the remote repository, including both to main and development branches. However, only two core processes have been successfully implemented to date.

The first process involves an initialization script that opens the code base and loads all necessary dependencies, preparing the system for automated testing. The second process executes a test script linked to a predefined set of test cases intended to validate controller functionality however, this stage currently experiences failures due to custom library compilation issues and conflicting file paths within the codebase. However, addressing these problems remains a priority for improving pipeline reliability and achieving a refined and effective automated simulation testing workflow.

4.5 Maintenance Methodology

This section outlines the codebase maintenance methodology adopted by UWAF to ensure the continued stability and longevity of the developed controller and the vehicle on which it is deployed. It summarizes the guiding principles the team follows to preserve software integrity through consistent upkeep, documentation and monitoring systems. More specifically, this section highlights the visualization and validation tools that are used, including both internally developed tools and third-party software, that facilitate real-time interfacing with the vehicle. Finally, the team's documentation practices are discussed, emphasizing the importance of clear and concise communication to support seamless knowledge transfer across project years and team cohorts.

4.5.1 Visualization Dashboards

4.5.1.1 Custom Tools

Internally, UWAF employs a custom-built dashboard (as seen in Figure 42) developed within the Simulink environment to monitor real-time vehicle behavior and visualize key serial data not displayed on the vehicle's instrument cluster, namely signals such as torque requests and battery voltages. This dashboard provides a clear view of the vehicle's status during runtime, focusing on normal operation rather than diagnostics or fault detection and has been constructed using Simulink

GUI elements. With the GUI, Simulink is able to stream signal data to visual components such as indicators, gauges and switches for interaction with the model while it runs. However, to maintain controller stability during operation, modification of control parameters during runtime is intentionally limited to ensure that validated software operates as designed without user interference.

Currently, the dashboard is integrated with the main control logic deployed to the Speedgoat Baseline M unit and this setup allows the PCM team to simultaneously observe system behavior and interpret key vehicle metrics during testing. Looking ahead, UWAFt aims to enhance this visualization system by developing a Simulink Real-Time application that functions as a standalone wrapper around the control logic [50]. This approach would create a dedicated and customizable monitoring interface that is decoupled from the controller itself (similar to a SCADA-style system), ensuring a smoother and more user-focused visualization experience.

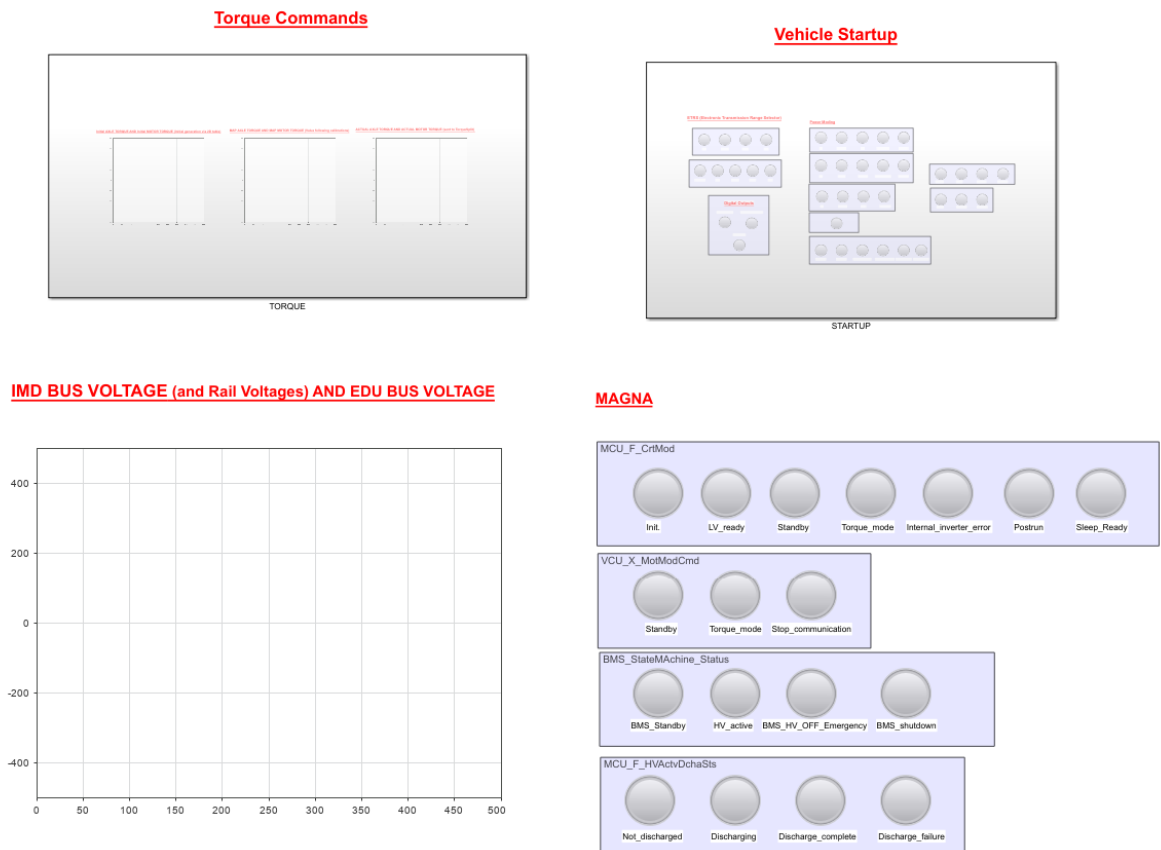


Figure 42 : UWAFt Visualization Dashboard

4.5.1.2 Third Party Tools

Externally to the dashboard, UWAFI has been fortunate enough to receive access to a wide variety of industrial diagnostic tools, courtesy of sponsorships related to EcoCAR. The primary tool UWAFI uses to monitor and analyze CAN traffic is Vector CANalyzer, which has both powerful visualization and diagnostic communication functionality [57]. CANalyzer is particularly valuable for inspecting signals that may not be directly tied to control logic, but are intermittently relevant to system monitoring and validation. When paired with the team's sponsored Vector VN1630 logger, the software enables full diagnostic communication with the vehicle using federally mandated protocols via the On Board Diagnostics (OBD) port [3]. Additionally, through custom connections to the vehicle's CAN harnesses, UWAFI can also observe live CAN traffic as it is transmitted, allowing the PCM and Connected and Automated Vehicle (CAV) team to analyze signals in real time.

Besides data monitoring, CANalyzer also provides extensive functionality for data post-processing, including cropping, comparing and replaying CAN logs. While not explicitly used by UWAFI, CANalyzer also supports Communication Access Programming Language (CAPL) scripting, an event-driven programming language to automate communication and ECU interaction [58]. CANalyzer configurations, which retain data such as workspace layouts and network parameters, can be saved and quickly reloaded, allowing seamless switching between communication with ECUs or monitoring CAN networks when they typically have different physical and network settings. Looking forward, UWAFI plans to integrate Vector CANape, an industry-standard calibration tool, to support controller calibration through third-party software, rather than custom solutions. Overall, the Vector tool suite forms the team's main initial tool for diagnostics and troubleshooting, offering exceptional stability and strong technical support.

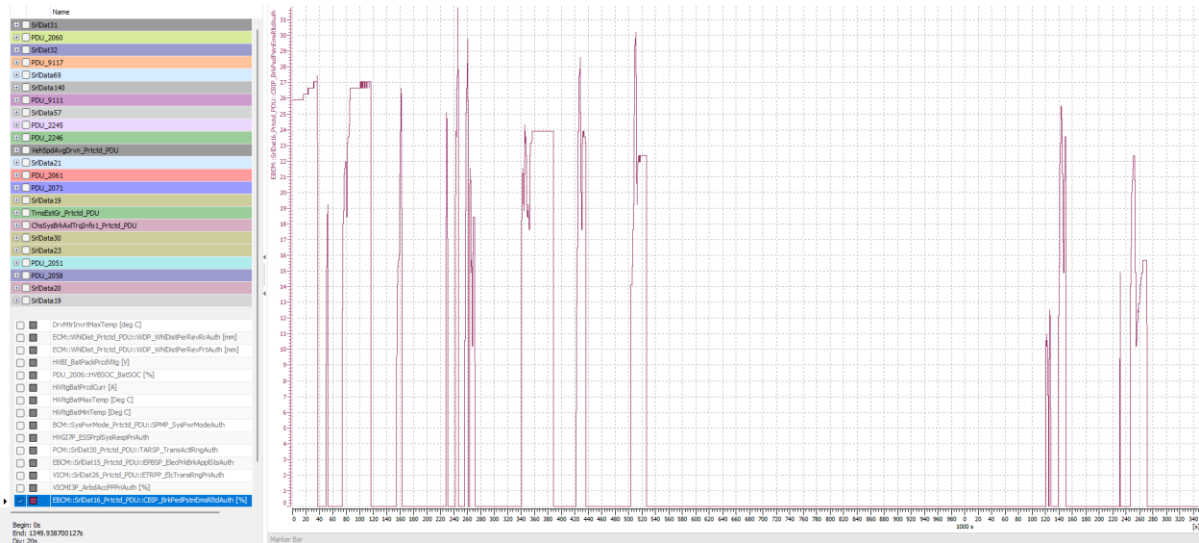


Figure 43 : CANalyzer Configuration for Full Vehicle Visualization (With an Accompanied Depiction of a Brake Percentage Trace)

4.5.2 Tool and Code Maintenance

In terms of tool maintenance, UWAFT prioritizes functionality and stability when updating software and development environments. All major third-party tools, such as Vector CANalyzer and other sponsored software packages that are external to the codebase, are updated regularly, with licenses renewed before expiration to avoid any disruptions to development or testing. When sponsors such as Magna and MathWorks provide new update packages or libraries, the team takes care to review the associated documentation and files to understand how these changes affect the existing workflow, ensuring that only relevant updates are implemented. One notable exception to this practice are updates to MATLAB and Simulink, which the team deliberately avoids updating frequently. Although MathWorks releases biannual updates, newer versions often introduce stability and compatibility issues, especially within EcoCAR's unconventional and prototype-oriented use case. Additionally, sponsors have sometimes cautioned teams to avoid making updates to preserve reliability within existing frameworks.

With regard to code refactoring, the team performed a major reorganization of the controller software following the transition from Year 1 (simulation-heavy development) to Years 2 and beyond, which focused primarily on real-time control software implementation. The refactored structure was aligned with the architecture used during the EcoCAR Mobility Challenge, where

similar software was deployed onto a Chevrolet Blazer. Since then, additional subsystems have been integrated, and development conventions have been formalized and documented (as described in Chapter 4.3.1) to streamline future development. Within the version control system, quarterly repository cleanups are performed to maintain organization by merging outstanding branches, resolving or closing old merge requests and deleting stale branches created for temporary research or experimental purposes.

4.5.3 Documentation

UWAFt maintains a collection of documentation and training materials designed to support the PCM team's development workflow, where these resources serve to assist new developers in onboarding efficiently and to provide existing members with guidance on how to perform complex tasks in adherence to team conventions. Upon joining the PCM subteam, new members are directed to a series of onboarding challenges which are a set of exercises developed by UWAFt to introduce the team's software development toolchain and demonstrate the complete lifecycle of a control feature. These challenges encourage hands-on engagement within the MathWorks environment and guide members through activities such as model development, simulation, testing and deployment. Each challenge also includes expected results, such that participants can understand key milestones and expected outcomes across the stages of feature development. Additionally, the team maintains a library of pre-recorded videos documenting the software model's evolution over the past four years, providing historical context and insight into the controller's progression.

To complement the onboarding challenges, UWAFt also offers tutorial videos and supplementary documentation focused on more advanced or specialized tasks that do not necessarily require standalone challenges. These include detailed guides on code deployment to the Speedgoat hardware, how to perform HIL testing and managing version control workflows. The tutorials are designed to be followed step-by-step, allowing team members to independently complete tasks while referencing the instructions. These materials are especially valuable for experienced members and team leads alike, as they provide concise and standardized material for reference to ensure consistency in practice across all levels of the PCM subteam. Collectively, these resources form a robust knowledge base that enhances learning and preserves technical continuity within UWAFt's controls development efforts.

Finally, to manage and organize all documentation, UWAFt utilizes a combination of private team servers, Confluence and the Microsoft 365 suite, including Teams and OneNote. This shift toward

Microsoft's ecosystem was driven by its seamless integration with the team's existing collaboration workflows, which allows information to be centralized and easily accessible. UWAFt strives to always emphasize proper documentation practices, capturing everything from meeting minutes (to ensure members can review decisions and outcomes) to detailed records of design decisions and system schematics. This structured approach enables quick reference during reviews to help justify and comprehend the evolution of all engineering decisions and not just the PCM sub team's. This ensures that critical knowledge is preserved and accessible to both current and future team members.

Chapter 5

Methodology Application Walkthrough, Validation and Outcomes

This chapter presents a concrete application of UWAF T's development workflow through the creation of a critical feature within the PCM subteam controller. The goal is to demonstrate how the structured, robust and competition-tailored workflow presented in Chapter 4 can be effective when generated by applying the methodology outlined in Chapter 3. The discussion will walk through each phase of the software development lifecycle for a specific feature, ranging from requirement generation and task creation to testing, validation and long-term maintenance, to showcase how these processes align to produce a reliable development structure. Furthermore, this section will highlight the actual outcomes and successes achieved by UWAF T's PCM subteam through the application of this generated workflow, particularly its role in enabling consistent performance and adaptability during competition events despite periods of uncertainty with vehicle functionality. Lastly, the section will outline future directions and planned improvements, detailing how the team intends to enhance both the controller and the supporting development framework, ensuring continued evolution and applicability of UWAF T's structured process to future EcoCAR challenges and related student design team initiatives.

5.1 Feature Example

This section presents an overview of the development of a function within the Pedal Map feature, which translates driver pedal inputs into vehicle acceleration commands. The feature's creation serves as a practical demonstration of UWAF T's development workflow, showcasing how processes ranging from requirement definition to validation have been applied to produce a safe and responsive control strategy.

5.1.1 Feature Explanation

As an ECU was removed from the donated Cadillac LYRIQ, UWAF T's PCM controller was tasked with replicating several critical functions previously handled by the removed unit. One of these functions is the generation of a torque command, which is achieved through the Pedal Map feature and maps driver inputs and vehicle states to generate a torque request. More specifically, the torque request is determined based on the current speed of the vehicle and the percentage of throttle input, allowing for accurate control of the drive units' torque output. In addition to the primary torque

generation logic, UWAFI has developed tertiary control logic that further processes the torque request to ensure vehicle safety and performance. This includes features such as battery protection, request saturation, vehicle direction management and brake pedal arbitration, which help ensure the vehicle operates safely under varying conditions.

The pedal map path is divided into several key stages, beginning with the generation of the torque request, as shown in Figure 44 (and more specifically the subsystem labelled as Item 1). At this stage, the torque request is derived from the pedal map illustrated in Figure 46, which uses vehicle speed and accelerator pedal position as primary inputs. The resulting torque command is then passed to Item 2, where it undergoes calibration and signal conditioning. This secondary subsystem refines the torque request by regulating characteristics such as the rate of change of signals to prevent undesirable effects such as excessive jerk, while also applying internal calibration limits to simplify tuning within the software. The process continues to Item 3, which manages torque negation and direction control. This stage ensures that driver-intended direction is respected and that the torque output is properly nullified in the event of a fault condition or brake pedal application. Following this, the processed torque request is split as shown previously in Figures 15 and 16 and saturated according to predefined hardware protection limits, namely power, current, or temperature thresholds as shown in Figure 45, before finally being transmitted over CAN to the drive units for execution.

To add, in Figure 45, the green box delineates the portion of the logic that is responsible for applying saturation that is drive unit level, defined for each individual motor and represents the lowest level of saturation within the physical system. These saturation limits are calculated through an algorithm that considers both temperature and electrical thresholds. The red subsystem is responsible for applying saturation limits that are driven by the capacity of the battery, specifically from an electrical point of view. It limits the torque to ensure that the entire system level torque is safe and of a magnitude that is achievable, when the energy to meet the request is sourced from the battery. Finally, the yellow subsystem is responsible for assigning the respective torque command (for front and rear axles) to their associated CAN signal.

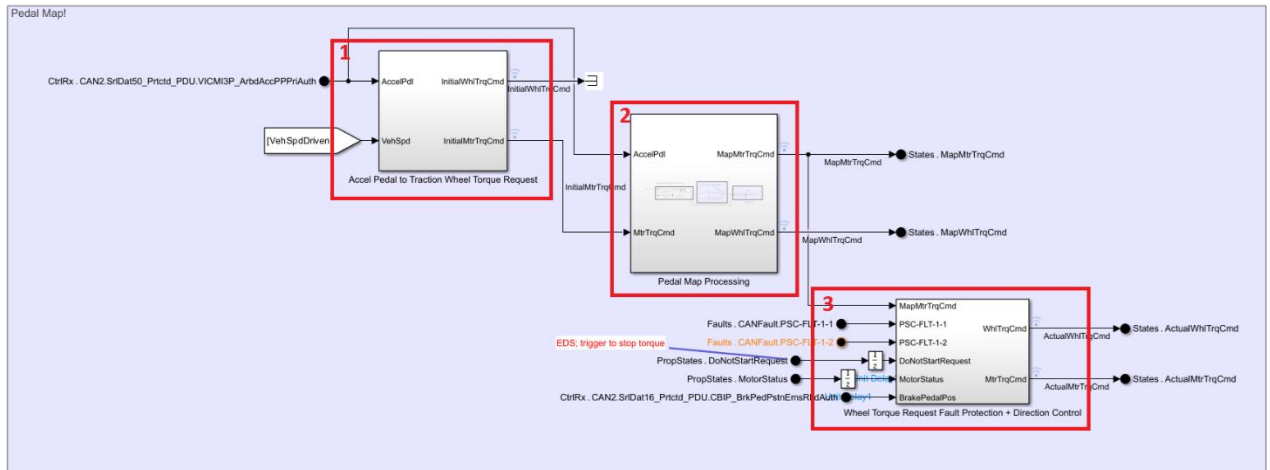


Figure 44 : Initial Pedal Map Subsystems

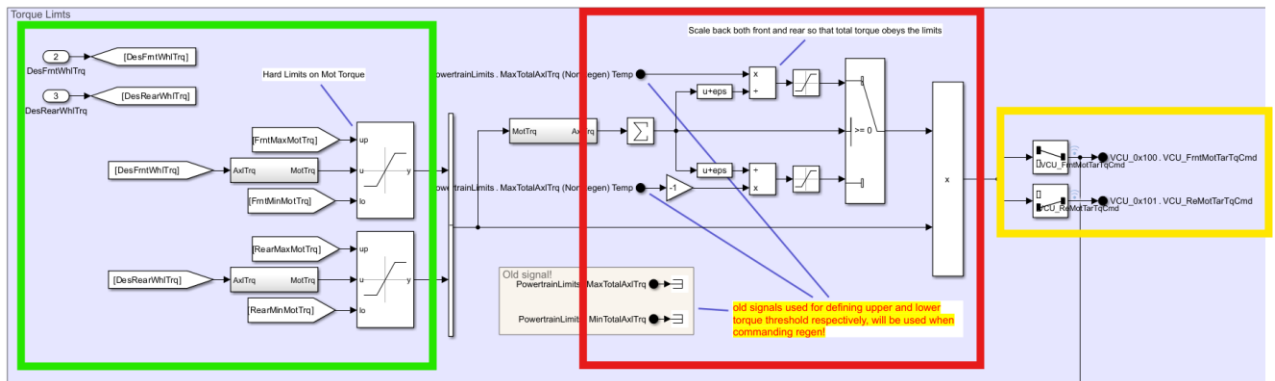


Figure 45 : Final Torque Request Saturation

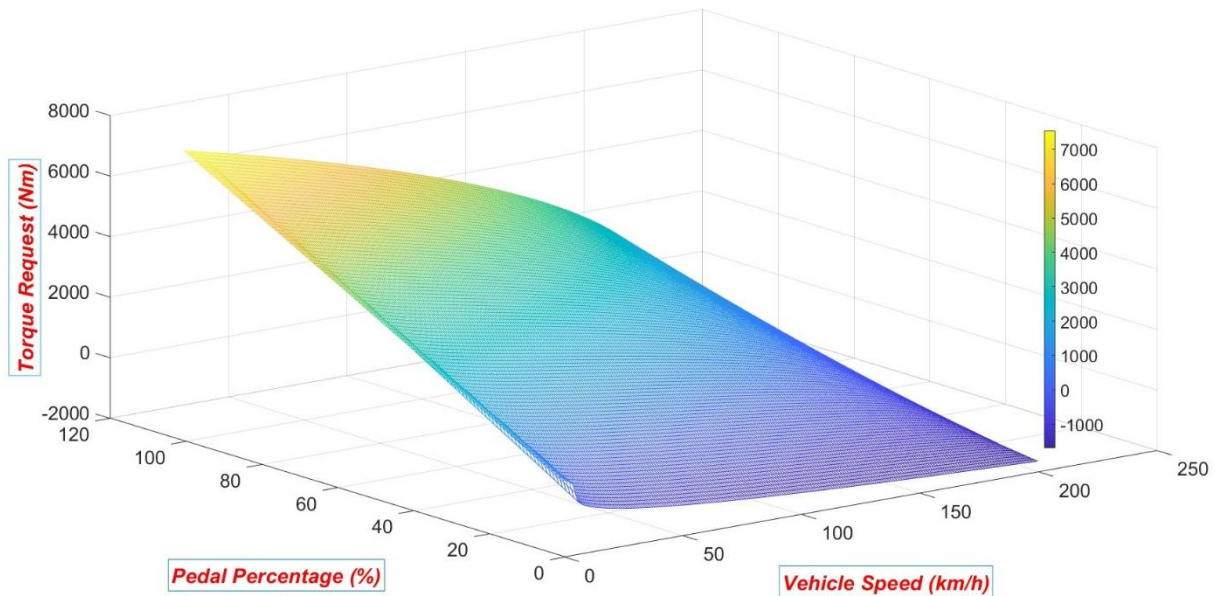


Figure 46 : Torque Request Generation Map

5.1.2 Requirement Sources, Generation and Task Creation

Before beginning development, it is essential to establish a comprehensive understanding of the feature to clearly define the objectives that PCM developers will work toward. This understanding can come from various sources such as team research or supplier documentation. Equally important is the early definition of internal performance goals and behavioral characteristics, which serve as benchmarks for proper software functionality. For this feature, the primary sources of information included GM control documentation, specifically the LPSO, Magna's drive unit documentation, internally defined team objectives related to functionality, safety and tunability and finally, safety analyses that have been performed to identify edge cases of interest. These sources collectively shaped both the scope and technical direction of the Pedal Map's development:

1. GM Documentation
 - a. Highlights serial data and networking considerations
 - b. ***Highlights vehicle level functionality integration (gear selector, physical driver interface, battery considerations and protections) ****
 - c. Highlights mechanical and physical limitations to consider

2. Magna Documentation

- a. Highlights serial data and networking considerations
- b. *Highlights drive unit level functionality integration (torque command signals, inverter hardware limitations, details on rear axle decoupler mechanism)****
- c. Highlights mechanical and physical limitations to consider

3. Team Goals

- a. Feature must be easily tunable and calibratable
- b. *Must contain multiple layers of redundancy and easily allow for added diagnostics****
- c. Allow for easy debugging
- d. Must take care of respecting hardware limits from all sources
- e. *Must prioritize occupant safety in unintended scenarios of acceleration****

4. Safety Analyses

- a. Must consider dual pedal input edge case*
- b. *Must consider redundancy when determining direction of intended motion****
- c. Must safely protect battery and drive units in the case of excessive thermal system increase
- d. Must ensure accurate torque split between front and rear to avoid unintended vehicle dynamics

Note*: *highlighted goals are the most relevant to the requirements analyzed during Chapter 5.1*

Once the main functionality goals and characteristics have been analyzed from their respective document sources, formal requirements should then be generated to concisely communicate what the software must literally be able to do. The examples below illustrate a subset of the requirements that were created and refined throughout the development process to guide feature implementation and ensure that the resulting software meet UWAFIT's specific needs:

- PSC-PWR-7.1 - The Wheel Torque Request Fault Protection and Direction Control system shall ensure that only negative values of torque are transmitted when the "Motor Status" flag is 3 indicating "Reverse"
- PSC-PWR-7.2 - The Wheel Torque Request Fault Protection and Direction Control system shall ensure that only positive values of torque are transmitted when the "Motor Status" flag is 2 indicating "Drive"
- PSC-PWR-7.4 - The Wheel Torque Request Fault Protection and Direction Control system shall ensure no torque values are transmitted when the Brake Pedal is pressed any amount.

While these represent only three of the many requirements defining the feature's complete functionality, they highlight the critical control behaviors that the team prioritized. Each requirement was developed following UWAF's standardized conventions (as outlined in Chapter 3.2.7) and clearly identifies the target subsystem, the intended functionality and the expected behavior of all relevant signals and states. For instance, requirement PSC-PWR-7.4 specifies that when the brake pedal position registers any nonzero input, the software must immediately saturate the torque command to 0 Nm, ensuring safe and predictable behavior. For the remainder of this chapter, PSC-PWR-7.1 and PSC-PWR-7.2 shall be the requirements of interest.

Subsequently, tasks must be generated to guide the development of the requirements into actual code including all non-development related work. This necessitates further decomposing the requirements into tangible and actionable tasks that need to be completed. Several example tasks have been created and presented below, including their formalization as work items in Jira

☐ Backlog (5 of 26 work items visible)	000	Create sprint
☒ UWAF-6193 Researching direction control tactics	TORQUE PATH	IN PROGRESS Nov 7
☒ UWAF-6197 Testing (MIL)	TORQUE PATH	IN R&D Nov 28
☒ UWAF-6196 Development of test cases (MIL)	TORQUE PATH	BACKLOG Nov 21
☒ UWAF-6194 Determine torque path integration with ETRS and driver inputs	TORQUE PATH	BACKLOG Nov 14
☒ UWAF-6195 Development of PWR-PWR-7.1	TORQUE PATH	BACKLOG Nov 14

Figure 47 : Sample Tasks for Requirement PSC-PWR-7.1

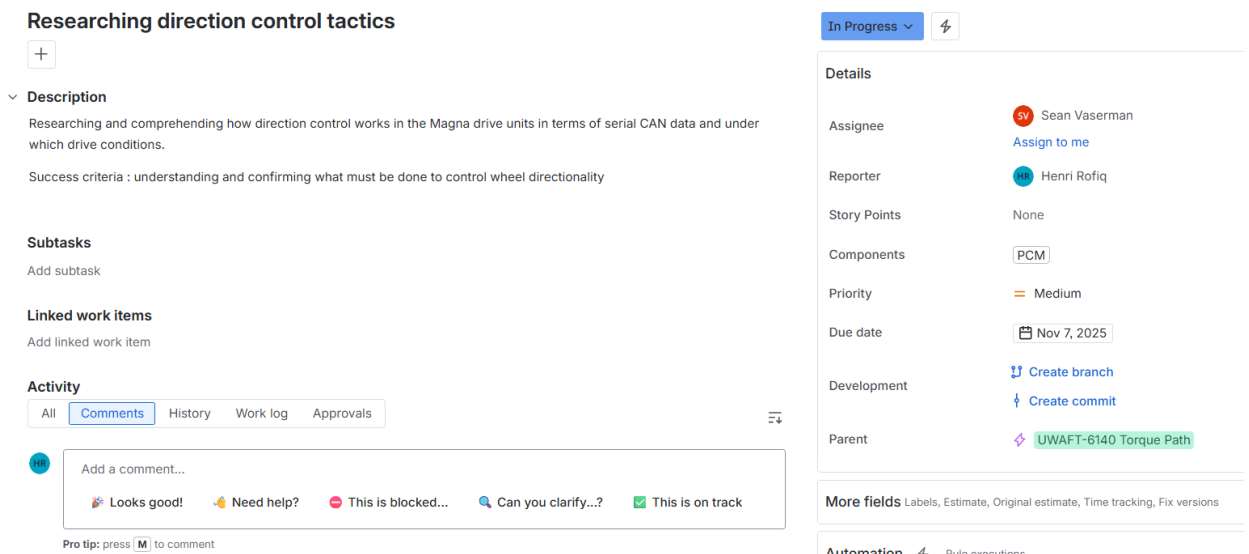


Figure 48 : Detailed Task Description

5.1.3 Logic Overview

Following the planning and definition of PSC-PWR-7.1 and PSC-PWR-7.2, the development phase represents the next critical stage in the software lifecycle. In this chapter, UWAFT’s approach to fulfilling these requirements through its developed control logic is examined in detail. While development methodologies and conventions can vary, the primary focus of this section is to demonstrate that the successful fulfillment of requirements remains the primary objective in robust control design. Furthermore, aspects such as run-time performance and code readability which are often considered implementation details, can themselves be also formalized as requirements. This point of view further emphasizes that well-structured requirements extend beyond defining just system functionality, but that they can also enforce good development practices, ensuring that software integrity is maintained throughout the design and implementation process.

The logic corresponding to previous requirements is contained within Item 3 of Figure 44 and further detailed in Figure 49. This subsystem performs a multiplicative transformation on the torque command, where the torque command is multiplied by 0, 1, or -1 depending on the state of the vehicle. If a fault is detected or if the vehicle is in a condition where torque should never be applied, the command is multiplied by 0, effectively negating any torque output. The same behavior occurs in a dual-pedal input scenario, where the brake input takes precedence over acceleration. When the vehicle is in Drive, the torque command is multiplied by 1, maintaining the original torque value,

whereas in Reverse, it is multiplied by -1 to command the drive units to initiate reverse motion only after the vehicle is stationary and the range change is confirmed.

The design of this feature began with identifying all scenarios requiring torque modification or negation and reviewing the corresponding requirements to ensure alignment with what is intended of the feature. This led to the consolidation of all related transformation logic into a single subsystem, positioned after torque generation and torque shaping subsystems, to handle this state-machine-like behavior effectively. The desired functionality was achieved using a minimal set of Simulink components, primarily composed of logical operators and a series of conditional (if-else) statements which effectively captured all necessary decision-making behavior. Additionally, the design was also generated by keeping in mind the most relevant technical goals for the Pedal Map feature presented in Chapter 5.1.2, namely goal 1b, 2b, 3b, 3e, 4a and 4b.

A multiplicative approach was chosen because torque direction and negation decisions can be discrete, infrequent events rather than continuous ones, making a simple algebraic transformation more appropriate than a dynamic algorithm. For example, range selection (Drive or Reverse) typically remains constant throughout a drive cycle and does not require continuous redetermination. While the logic exhibits discrete state behavior, a Stateflow implementation was intentionally avoided, as the system manipulates existing signals rather than generating new ones. Nevertheless, this block operates synchronously with the electronic transmission and range selection (ETRS) logic, which is a standardized GM feature required across all EcoCAR teams, but is discretized due to the difference in base functionality.

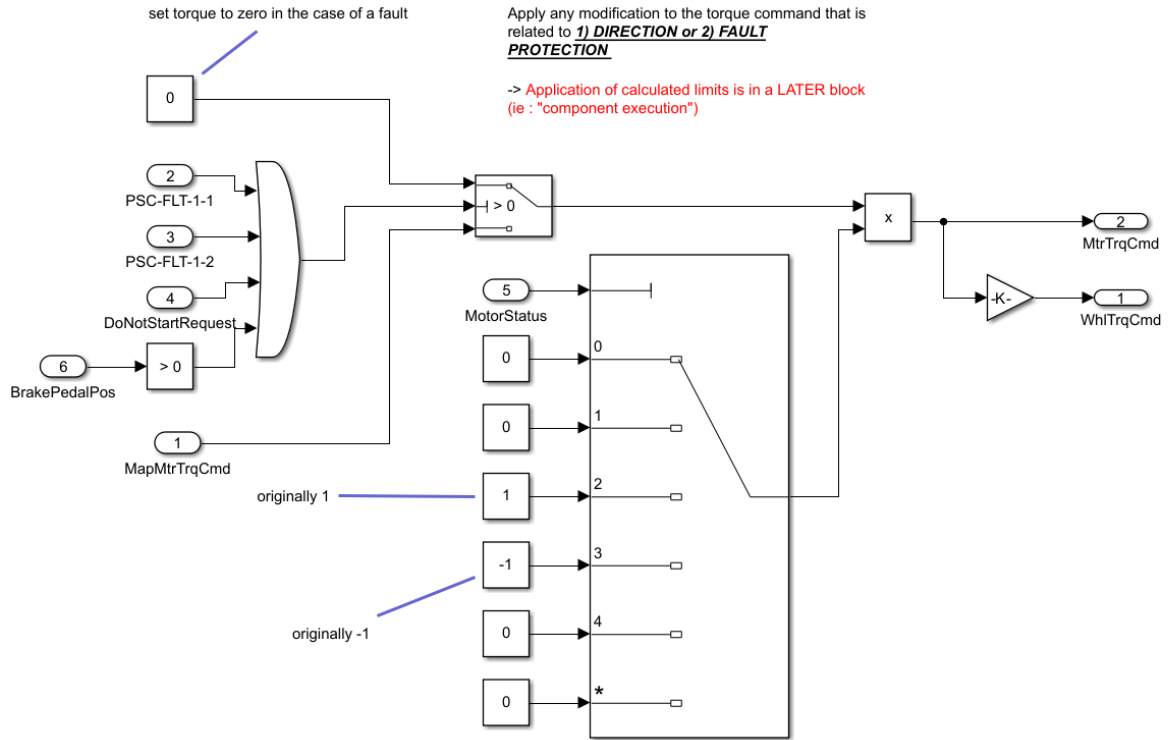


Figure 49 : Torque Command Directional Control Logic

5.1.4 Software Integration and Version Control

A dedicated development branch was created to contain all work associated with this feature and its corresponding requirements:

CTRL/feature/5967/PedalMapRework

This branch served as the central workspace for implementing the logic described in Chapter 5.1.3. Given the critical nature of the Pedal Map feature, all progress was regularly committed and pushed to the remote repository after each work session to maintain synchronization between local and cloud versions. This eliminated issues with outdated code being disseminated to multiple developers working in parallel, with the goal of primarily minimizing integration conflicts.

The merge into the main branch was intentionally deferred until all stages of testing were completed and validated. At the time of development, the feature had successfully passed preliminary MIL and HIL simulation stages (which will be elaborated upon in Chapter 5.1.5), thus achieving a level of operation deemed safe for VIL testing with the vehicle immobilized on a hoist (i.e., raised

wheels and no risk of unintended movement). However, unforeseen issues arose during VIL testing, compounded by logistical constraints related to vehicle shipping for the year-end competition, necessitating a temporary return to MIL and HIL testing stages. Once the issue was resolved during on-site validation at competition, the feature branch was successfully merged into the main branch and subsequently flashed to the Speedgoat for long term operation.

5.1.5 Testing and Validation

5.1.5.1 MIL and HIL Testing

To start the testing phase, specifically MIL testing, the first step involves generating a simulation environment capable of stimulating the developed logic and validating its outputs under controlled conditions. As outlined in Chapter 4.4.1 and illustrated in Figures 25 and 26, UWAFt employs a standardized test harness and environment that remain consistent across all features. This approach ensures a seamless connection between the development and testing environments, minimizing discrepancies and eliminating the need for different versions of software between simulation and deployment (except in specific scenarios that are typically rare).

Several test cases / temporal assessments (which are shown in Figure 50) that accurately evaluated the performance of the logic generated for requirements PSC-PWR-7.1 and PSC-PWR-7.2, were developed by directly translating each requirement's intent into the specific conditions and system states the logic must attain when injected with representative test data. Ultimately, UWAFt is aiming to assess the behaviour of the generated torque command signal and whether or not the logic abides by the outlined requirement, as evaluated via Simulink using the defined temporal assessments. This process was streamlined by UWAFt's established conventions, which emphasize that all requirements should be objectively testable and easily translatable into measurable and objective evaluations.

ENABL...	NAME	ASSESSMENT
☒	• PSC-PWR-7.4	▶ At any point of time, whenever BrakePedal > 0 is true then, with a delay of at most 0.5 seconds, TorqueCommand == 0 must be true
☒	• PSC-PWR-7.1	▼ At any point of time ... ▼ trigger: whenever is true condition: MotorStatus == 3 ▼ delay: With a delay of at most max-time (sec): 0.5 ▼ response: must be true condition: TorqueCommand <= 0
☒	• PSC-PWR-7.2	▶ At any point of time, whenever MotorStatus == 2 is true then, with a delay of at most 0.5 seconds, TorqueCommand >= 0 must be true

Figure 50 : Directional Control Temporal Assessments

To accompany Figure 50, Figures 51 and 52 are diagrams which represent the passing and failing cases respectively, that the temporal assessments are looking for. The “TRIGGER” represents the desired value that the primary signal must attain in order to execute an evaluation of the remainder of the test case. The “RESPONSE” represents what are the intended and unintended behaviours, in Figure 50 and 51 respectively, of the secondary response signal when the trigger signal transitions to the desired state.

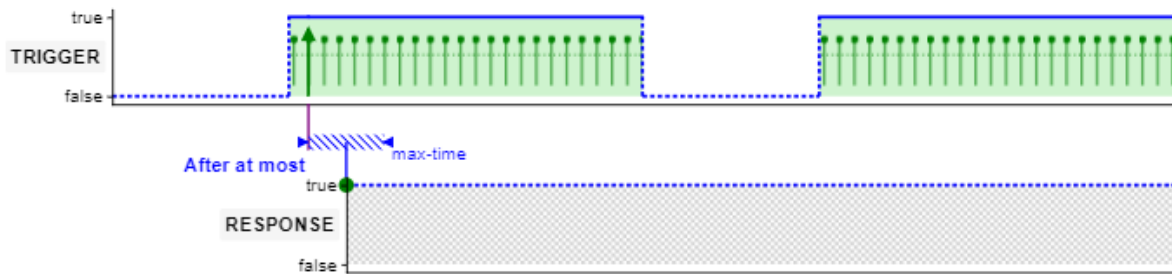


Figure 51 : Passing Visualization of the Temporal Assessment

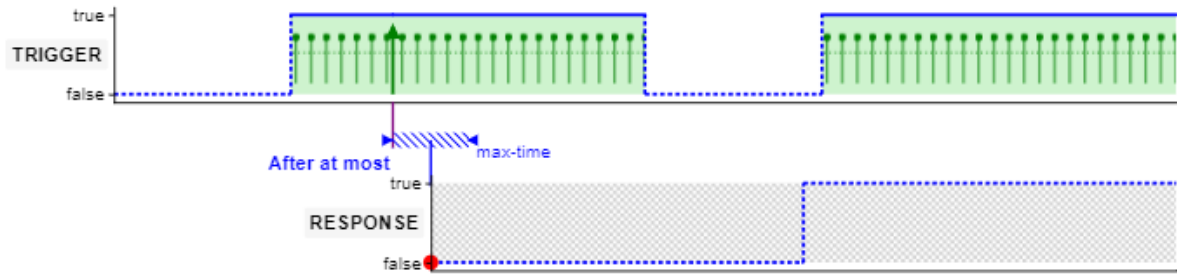


Figure 52 : Failing Visualization of the Temporal Assessment

Regarding the data that was injected during testing, a combination of temporary simulated inputs and previously recorded CAN data was utilized. The selected CAN log captured a representative scenario from when the Cadillac LYRIQ was in its stock configuration, where the vehicle was powered on, shifted between Drive and Reverse and allowed to move forward when the brake is released, causing the phenomenon of “creep”, similar to that of internal combustion vehicles, where a constant forward motion occurs due to continuous engine operation. In this dataset, the accelerator pedal signal was absent (logged as a constant zero), requiring it to be spoofed (with an accelerator pedal signal originating from a secondary drive cycle log) for testing purposes, as it is an essential signal for torque command generation within UWAF’s control logic. Meanwhile, signals such as motor status, brake input and other relevant parameters were directly imported from the CAN log.

A shorter dataset was intentionally chosen over a full drive cycle, as longer recordings significantly increase simulation time, especially during HIL testing, where real-time execution makes long simulations impractical (ex : a 600-second drive cycle requires 600 seconds of real time). Additionally, the employed dataset closely represents the type of maneuvers that UWAF would perform during VIL testing, making it ideal for MIL and HIL testing as well as verifying low-level functionality. At this stage, the goal was to validate base logic behavior, specifically ensuring correct torque magnitude (positive vs. negative) and confirming that the core control logic that was developed, in pursuit of achieving the requirements, performed as intended before progressing to full pedal map integration. The bulk of the testing that was performed followed an “open-loop” testing framework, similar to what was described in Chapter 3.4.1. This testing approach was selected because the requirements were not confirming a physical response which could only be generated using a vehicle model and dynamic feedback. Instead, the testing is uni-directional in the sense that when the control logic has achieved a specific state (which it is commanded to enter), a distinct

behaviour should be observed. The requirement is ignoring signal validity and impacts on downstream dynamics.

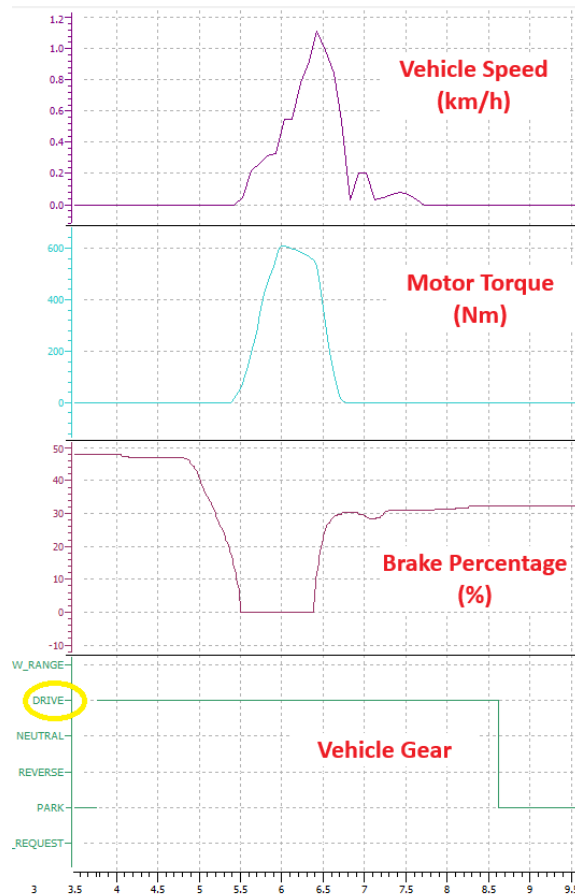


Figure 53 : Relevant Injected CAN Test Data

Once the data was injected into the test harness during MIL testing and the temporal assessments were evaluated, the logic successfully passed all evaluations, which demonstrates that the logic can function according to the intended behavior as defined by requirements PSC-PWR-7.1 and PSC-PWR-7.2. Figures 54 and 55 illustrate the data actively monitored by the Simulink Test Manager, confirming that the logic consistently adhered to the assessment criteria. These plots display the gear transition (representing motor status), accelerator pedal position and the resulting torque command, which is the key signal undergoing the multiplicative transformation described in Chapter 5.1.3. As shown, when the vehicle is in the correct gear range and the accelerator pedal (always a positive

signal) is applied, the generated torque assumes the appropriate magnitude of positive, when in Drive and negative, when in Reverse.

While the figures provide a visual confirmation of this relationship, it is important to note that they are primarily for illustrative purposes. In practice, test cases are always designed to objectively evaluate defined assessments and to automatically flag any deviations from expected behavior (with the added benefit of pinpointing failure sources for debugging). Further HIL testing produced identical results, which was expected as requirements should remain consistent across test environments, with only the test setup differing. Once the relevant models were configured (as outlined in Chapter 4.4.1, Figures 34 and 35), HIL validation highlighted that the implemented logic exceeded timing performance requirements and was able to respond nearly fifty times faster than required (0.01 s vs 0.5s). This demonstrated that the control logic was not only functionally correct, but also safe and responsive, highlighting that the hardware and logic could react reliably in response to drivers reacting to hazards.



Figure 54 : Drive Gear Logic Response

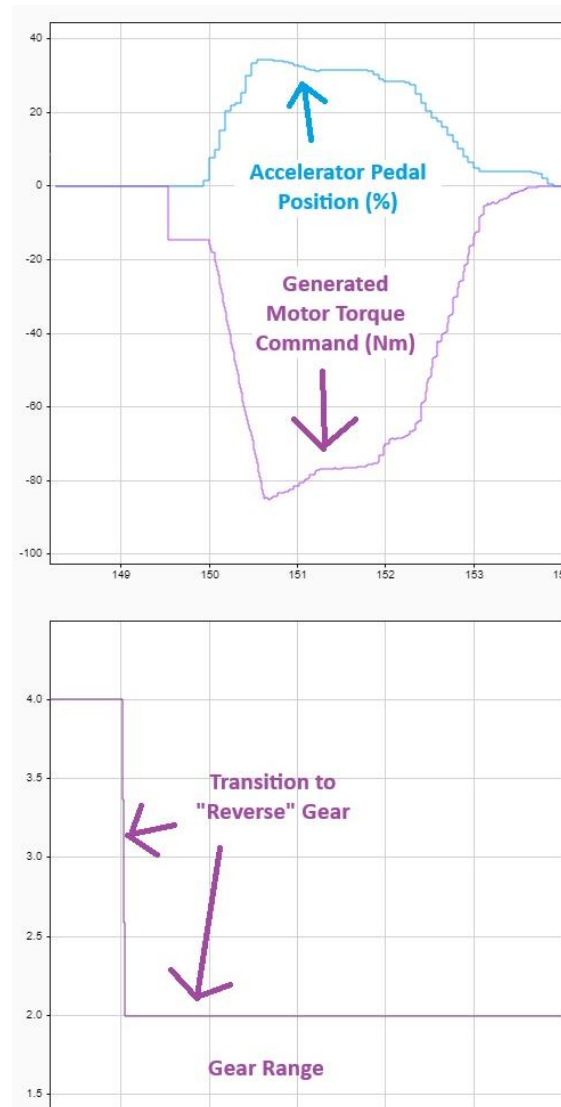


Figure 55 : Reverse Gear Logic Response

5.1.5.2 VIL Testing

After completing all virtual validation stages, the team determined that the software was ready for deployment to the vehicle through the use of a preliminary VIL setup, conducted prior to closed-course testing, with the intention of safely and effectively verifying that the implemented logic satisfied the defined requirements under controlled conditions. As shown in Figure 56, the vehicle is fully immobilized on a lift, ensuring there was no possibility of unintended movement during testing.

Unlike virtual testing, VIL evaluation cannot be automated as it requires a human operator to manually execute test cases and assess vehicle performance as objectively as possible. The correlation of this testing phase with MIL and HIL testing was enhanced by the team's intentional choice of selecting representative test data and injected scenarios during virtual testing. As a reminder, the simulated data represented a simple and repeatable maneuver, of a range change followed by a small torque request to verify wheel direction and torque response. This approach provided a safe and practical method to control behavior in a real vehicle environment without requiring a full drive cycle, which would not be feasible or safe to perform on a vehicle lift.



Figure 56 : Preliminary VIL Testing Setup

During VIL testing, it was observed that while the control logic performed successfully in Drive, it failed in Reverse as the torque command was not being honored, and the wheels did not rotate as expected. This outcome provided two key insights with the first being that the implemented logic was fundamentally sound but required refinement to ensure full bidirectional functionality. The second insight was that despite the issue, the safety of the system was upheld as the observed failure mode was a lack of motion rather than unintended acceleration, which is typically a much safer outcome for both occupants and the vehicle.

This discovery necessitated a return to MIL and HIL testing, particularly as the vehicle was scheduled for shipping. Fortunately, UWAF's robust data logging practices allowed the recorded VIL data to be reinjected into the simulation environment, with the goal of replicating the real-world vehicle state for debugging. This cyclical testing approach, where real data is used to drive virtual validation, is essential in increasing testing fidelity and ensuring continuous improvement within the development framework.

Further analysis identified the root cause being within downstream logic, specifically in the state estimation and torque limiting logic subsystems shown in Figures 13 and 18, where unit errors caused unnecessary saturation of torque requests. After correcting this issue and validating additional related requirements, the software was redeployed to the Cadillac LYRIQ for another round of VIL testing at competition. These tests, conducted under controlled conditions during the Vehicle Technical Inspection (VTI) with the vehicle safely immobilized on a lift, successfully verified the fix and reaffirmed the effectiveness of UWAF's iterative testing workflow.

5.1.6 Maintenance, Debugging and Documentation

Following deployment and the subsequent debugging process pre-competition, several key initiatives were undertaken post competition to prevent recurrence of the issues that were encountered. To maintain the stability of the codebase, the team conducted a comprehensive review of the entire software model, where sign and magnitude checks across all relevant subsystems were performed. This extended beyond the torque path and actuator control systems to include diagnostics, data calculation and serial communication subsystems, that, while not directly affecting the driver, are essential to ensuring overall vehicle functionality. The lessons learned were formalized as a new requirement within the RTM and additional comments were added throughout the model to enhance clarity. Documentation summarizing the debugging process, corrective actions and outcomes was also generated and stored in the Microsoft Teams repository for easy access by team members. The final validated model was successfully merged into the main branch and appropriately tagged as a milestone release, marking a key point in the controller's development timeline.

The visualization dashboard was also important during post-competition debugging when a new issue arose. This issue was related to the fact that the vehicle suddenly exhibited shaking behavior when in the Drive gear, but not in Reverse. Using the dashboard (illustrated in Figure 42), torque command signals were monitored in real time and it was revealed that there existed oscillations that

correlated with the shaking. This insight led the team to hypothesize that a calculation error within a subsystem triggered the oscillatory behavior. The root cause was traced to a battery-related fault that was triggered externally to UWAF's controller, which placed the vehicle in a state of reduced functionality. The UWAF PCM controller correctly responded by limiting functionality in accordance with defined safety logic, demonstrating that the controller operated as intended, even though the remedial action manifested in a less than desirable format. This incident further reinforced the importance of visualization tools for real-time diagnostics and the effectiveness of UWAF's monitoring approach.

5.2 Validation, Advantages and Outcomes

The application of the workflow generation process in developing the UWAF workflow has proven highly effective in establishing a methodology that is meticulous and emphasizes robustness throughout every stage of the controller development lifecycle. The implementation of this workflow has produced several tangible outcomes, containing both upsides and downsides, related back to not only how the team approaches software development, but also how it aligns with the broader goals of the EcoCAR competition which is to enable students to develop a deep familiarity with industry-standard tools, methodologies and best practices. In doing so, it bridges the gap between academic research and industrial automotive engineering, preparing students for the real-world demands of vehicle control system development.

5.2.1 Competition Outcomes

Within the EcoCAR competition, particularly during Year 3, several notable successes emerged as a direct result of implementing the UWAF-developed PCM workflow. Despite limited vehicle access in the early stages of competition, the PCM team successfully completed all technical controls inspections and functional checks, which include charging validation, vehicle motion tests and vehicle startup/shutdown sequencing on the first attempt. This achievement highlights the rigor and reliability of the team's workflow, which emphasizes thorough simulation and virtual validation prior to vehicle deployment. The structured approach ensured that by the time VIL testing was conducted, the control logic was stable and predictable, thus enabling all tests to be completed in just two days while enduring multiple time constraints and external pressures.

In addition to technical success, reviews and presentations with industry partners further validated the team's methodology. When showcasing the complete workflow, from requirements management through to control logic implementation by using the generation of the torque path as a guiding example, UWAFI placed in the top three among thirteen competing teams. Judges and professionals from leading OEMs and suppliers commended the team's workflow for its end-to-end traceability, robust iterative testing framework and strong emphasis on functional safety, all of which mirrored real-world automotive development practices.

Beyond competition results, the workflow has had a lasting impact on PCM sub team members and alumni. The structured training, conventions and documentation derived from the workflow have accelerated new member onboarding and strengthened technical competency. Many current and former team members have used their skills to secure internships and full-time positions in the automotive industry, with some transitioning into controls and systems engineering roles from unrelated disciplines. Several alumni have even cited UWAFI's workflow documentation and training materials during their employment, which ultimately demonstrates how closely aligned the team's practices are with current industry standards.

5.2.2 Team Outcomes

The implementation of UWAFI's PCM development workflow led to several significant team-specific improvements that transformed how the subteam operates and manages its work. Firstly, the structured workflow greatly enhanced project management and tracking capabilities which allowed the team to clearly understand task dependencies and their relationship to both internal and EcoCAR competition deadlines. This visibility improved scheduling and resource allocation while also creating a record of task durations and difficulty, which now serves as a valuable reference when planning future development efforts and estimating workload requirements.

Secondly, the team achieved far greater traceability and transparency across all development stages. By linking requirements directly to tasks, testing and validation results, members gained a clearer understanding of the purpose and priority of their work. This traceability not only optimized time management but also enabled better collaboration, as developers could identify dependencies, seek support from the right sources and communicate current status more effectively to leads and project managers.

Thirdly, the emphasis on a rigorous and well-documented workflow established a culture of consistency and accountability. Every design decision and task is now backed by reasoning and a well documented workflow such that development follows a logical progression rather than experimentation. This structured approach proved especially beneficial for a student team such as UWAF, where competing academic and personal responsibilities often limit availability and by defining a clear roadmap, contributors can engage more effectively and confidently in the project.

Before this workflow was introduced, many of these processes lacked structure. While UWAF's tool selections (e.g., MATLAB, Simulink, Speedgoat) was predetermined by sponsorships, the core aspects of requirements management, controller development, testing and maintenance were loosely defined. Jira usage and sprints were inconsistent, thus leaving the team uncertain about its pace relative to deadlines or other sub teams. Testing was also fractured as frequent broken links between development and validation environments occurred due to configuration issues. Industry reports suggest that software testing cycles often span 8 weeks to 8.5 months [59], whereas under UWAF's refined two-week sprint workflow, the team was able to implement feature development, starting from requirements generation all the way to vehicle deployment within roughly 4 weeks per feature, thus demonstrating a major leap in efficiency.

However, one key challenge that persists is managing dependencies across sub teams. While the PCM subteam has full visibility of its internal tasks and milestones, coordinating shared goals and aligning critical paths with other teams, especially when overlapping goals and efforts exist, continue to be an ongoing difficulty. Improving cross-team communication and dependency management will be a focus for refinement as UWAF aims to improve its processes in future competition years.

5.3 Future Endeavors

As demonstrated throughout Chapters 5.1 to 5.2, the application of the workflow outlined in Chapter 4 has produced highly encouraging preliminary results and established a solid foundation for the continued refinement and maintenance of UWAF's control software codebase. The methodology has enabled the team to design and validate software in a more consistent and structured manner, ensuring that new features can be developed efficiently while maintaining system robustness and safety. However, the current controller has several outstanding and ongoing tasks that fall into three primary categories: feature expansion, controller refinement and improved testing fidelity.

From a functionality standpoint, the Cadillac LYRIQ has reached a stage of stable operation, capable of reliable propulsion, predictable torque response and basic fault detection and diagnostic functionality. With this baseline being established, UWAFIT's next phase of development targets more advanced vehicle-dynamics-based control features, including adaptive torque splits, wheel flare management and enhanced acceleration response with the goal of improving both vehicle performance and drive quality. Additionally, the team plans to integrate regenerative braking, with the goal of optimizing energy recovery and overall system efficiency.

Alongside these new developments, efforts will also focus on refining the usability and consumer acceptance of the existing control logic through improving startup sequencing, fine-tuning control calibrations and revalidating existing algorithms for better consumer acceptance.

Finally, UWAFIT also seeks to enhance the fidelity of its testing framework by improving both the quality of injected data and the accuracy of vehicle models used in simulation. The primary objective is to ensure that virtual testing environments more closely reflect the current state of the LYRIQ, particularly following mechanical and electrical integration updates. This effort is already underway, as data collected from limited VIL testing sessions are continually being repurposed for more accurate "open loop" simulations. Moving forward, UWAFIT aims to develop robust plant models that accurately capture the LYRIQ's real-world behavior, enabling more precise MIL and HIL testing during "close loop" simulations for features requiring parameter tuning and dynamic performance validation.

Each of these improvements will continue to leverage UWAFIT's established workflow, ensuring consistency in documentation, testing and traceability. Ultimately, the work completed to date represents just the beginning of an iterative process that continues to evolve as the team applies its structured development framework to expand functionality and prepare for future competition milestones.

Chapter 6

Conclusion

6.1 Summary of Work

The increasing integration of software in modern vehicles has become an integral part of automotive innovation, as it has enabled a variety of advanced features ranging from safety systems such as automatic emergency braking to performance technologies like traction control. The design and implementation of such features are inherently complex and time-consuming, yet the tools and methodologies required to develop them are rarely covered in traditional post-secondary education. As students transition from academia into the automotive industry, more specifically within the rapidly growing zero-emission vehicle sector, familiarity with industry-standard workflows becomes critical. Student design teams, such as those participating in programs like the EcoCAR EV Challenge, serve an ideal environment for encouraging hands-on learning and incubating interest with controls design however, they often face resource constraints that limit their ability to develop software aligned with professional standards. The objective of this thesis is to establish a structured and adaptable control logic development workflow that offers clear guidance to student teams and engineers on how to design, test and maintain automotive control systems.

While literature on controller design primarily focuses on architectural structuring and control feature development within both academic and production contexts, this thesis advances the field by presenting a methodology for generating and evaluating a customized workflow. The proposed framework addresses all key considerations necessary for the deployment of control logic that is tailored to the needs and limitations of smaller teams.

Developing control software requires reconciling several high-level considerations. First, selecting the appropriate tools and hardware for development and deployment is fundamental, as these decisions influence cost, ease of implementation and team accessibility. Second, requirement generation is critical since defining clear and testable objectives ensures that the controller performs as intended and prevents inefficiencies such as scope creep or misunderstandings. Third, establishing robust software architecture conventions and version control strategies ensures consistency across efforts undertaken by software developers. Fourth, defining repeatable testing procedures verifies the software's behavior under realistic conditions and increases confidence in software reliability upon

vehicle deployment. Finally, outlining structured maintenance and documentation practices preserves long-term functionality, supporting future iteration and reducing the risk of obsolescence.

Applying this workflow to UWAFIT's participation in the EcoCAR EV Challenge, the team adopted a MBD approach using MATLAB and Simulink for developing software that is ultimately deployed to a Speedgoat real-time computer. Version control was managed through Git, while project management followed a hybrid Agile methodology. Requirements were derived from multiple sources, including supplier documentation (GM and Magna), internal performance targets and safety analyses that informed critical system behaviors and edge cases. This resulted in a layered, component-based and pipes-and-filters architecture that separated control logic from communication interfaces. A comprehensive testing framework that encompasses MIL, HIL and VIL stages was employed to validate critical features such as directional control, ensuring safe and predictable performance. Continuous maintenance practices, including documentation, repository clean-ups and periodic code refactoring ensured the long-term health and stability of the software ecosystem.

Ultimately, this work propels the formalization of controller development techniques in an easy to comprehend format that is aligned with industry trends, but also applicable and scalable to academic initiatives such as student design teams. It provides an accessible yet technically rigorous framework that empowers student design teams to produce reliable software systems under realistic engineering constraints. Moreover, it contributes to the disciplines of engineering management and educational learning as it offers a scalable model for training the next generation of engineers in the principles of modern automotive software development.

6.2 Current Limitations and Future Work

While the workflow generation methodology presented in this thesis has yielded promising results, which culminated in the successful development and deployment of UWAFIT's PCM controller to a Cadillac LYRIQ, it is not without its limitations. Firstly, although the workflow outlines clear strategies for task management and parallelism within a single controller's development, it provides limited guidance on managing multiple simultaneous projects competing for shared resources. For instance, UWAFIT is required to develop additional controllers capable of supporting ADAS functionality and thus, the coordination and interdependency between these similar efforts present new challenges. The existing development workflow can be extended to support ADAS development, but this would require refinement when managing resource allocation, cross-team communication and

synchronization between concurrent software projects. Future work should involve consulting industry experts and project management frameworks to strengthen these multi disciplinary collaboration strategies.

Additionally, while CI/CD pipelines were discussed as an optional enhancement in Chapter 3.4.4, their absence within UWAFIT's implementation limits the team's exposure to, and experience with, automated testing frameworks. A properly configured CI/CD pipeline could substantially improve testing repeatability and therefore, future research should focus on formalizing CI/CD integration across diverse toolchains to improve automation and minimize manual validation efforts.

Finally, further investigation is needed to explore how the proposed workflow could be adapted for internal combustion engine (ICE) vehicle architectures. While many of the management principles, hardware integration methods and testing conventions can carry over, the safety standards, control challenges and software integration depth differ between ICE and EV systems. EV development emphasizes high-voltage safety and electric powertrain efficiency, whereas ICE systems introduce additional complexity through challenges such mechanical actuation and emissions control. While the EV sector is rapidly growing, sales of ICE vehicles are still occurring and the functional safety frameworks governing these systems are more mature, having evolved over decades of standardization.

In conclusion, addressing these areas of future research would enable the creation of a more holistic and flexible control development methodology. Such advancements would extend the applicability of this research beyond UWAFIT and the EcoCAR competition, supporting a broader segment of the automotive industry and extracurricular efforts. Ultimately, this future work would not only enhance the robustness and safety of control logic software but also further inspire the next generation of engineers to pursue innovation, within the rapidly evolving field of automotive design and electrification.

References

- [1] Zoepf, S., & Heywood, J. B. (2012). Characterizations of deployment rates in Automotive Technology. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 05(2), 541–552. <https://doi.org/10.4271/2012-01-1057>
- [2] California Air Resources Board. (n.d.). On-Board Diagnostic II (OBD II) Systems Fact Sheet. <https://ww2.arb.ca.gov/resources/fact-sheets/board-diagnostic-ii-obd-ii-systems-fact-sheet#:~:text=All%201996%20and%20newer%20model,meet%20the%20OBD%20II%20requirements>.
- [3] Universal interface for OBD II scan. (1999). *SAE International Recommended Practice*. https://doi.org/10.4271/j2201_199907
- [4] Al-Qassab, H., Pang, S., Al-Qizwini, M., Kent, D., & Radha, H. (2019). Active safety system for connected vehicles. *SAE International Journal of Connected and Automated Vehicles*, 02(3), 191–200. <https://doi.org/10.4271/12-02-03-0013>
- [5] *Student Headcounts: Institutional Analysis & Planning: University of Waterloo*. Institutional Analysis & Planning | University of Waterloo. (2025, October 1). <https://uwaterloo.ca/institutional-analysis-planning/university-data-and-statistics/student-data/student-headcounts>
- [6] Nam-Wong, S. (2025, September 10). UWAFST Survey of Student Demographics. Waterloo; University of Waterloo.
- [7] University of Waterloo. (2025, November 3). *Directory of teams: Sedra student design centre: University of Waterloo*. Sedra Student Design Centre | University of Waterloo. <https://uwaterloo.ca/sedra-student-design-centre/directory-teams>
- [8] Arami, A. (2022). *Course Outline. ME360 Introduction to Control Systems*. Waterloo; University of Waterloo.
- [9] Staron, M. (2021). *Automotive Software Architectures: An introduction* (2nd ed.). Springer International Publishing Springer.
- [10] Upadhyay, D., & Sampalli, S. (2019). SCADA (Supervisory Control and Data Acquisition) systems: Vulnerability Assessment and Security Recommendations. *Computers and Security*, 89, 101666. <https://doi.org/10.1016/j.cose.2019.101666>

- [11] Dwyer, M. (n.d.). *Hybrid vehicle sales continue to rise as electric and plug-in vehicle shares remain flat*. Today In Energy. <https://www.eia.gov/todayinenergy/detail.php?id=65384>
- [12] Takeichi, A., Kosaka, K., NOBE, D., Suzuki, T., Miyake, S., & Tsukamoto, N. (2023). Development of control system for parallel hybrid system with turbo engine. *SAE Technical Paper Series, I*. <https://doi.org/10.4271/2023-01-0547>
- [13] Hasan, S. M. N., & Irgens, P. (2025). Electronic Control Unit hardware design challenges for software defined vehicle. *SAE Technical Paper Series, I*. <https://doi.org/10.4271/2025-01-8136>
- [14] Fricano, A. (2019). *Design of Torque Vectoring system for Formula SAE electric vehicle* (thesis).
- [15] Loof, J. W., Besselink, I. J. M., & Nijmeijer, H. (2014). Traction control of an electric formula student racing car. In Proceedings of the FISITA 2014 World Automotive Congress, 2-6 June 2014, Maastricht, Eindhoven University of Technology
- [16] The MathWorks. (n.d.). *Simulink - simulation and model-based design*. Simulation and Model-Based Design - MATLAB. <https://www.mathworks.com/products/simulink.html>
- [17] Hejase, M., Kurt, A., Aldemir, T., & Ozguner, U. (2018). Identification of risk significant automotive scenarios under hardware failures. *Electronic Proceedings in Theoretical Computer Science*, 269, 59–73. <https://doi.org/10.4204/eptcs.269.6>
- [18] Tan, Z., Yang, X., Li, Y., Shen, J., Zhang, J., Liu, H., Shuai, X., Zhao, F., & Yu, W. (2025). Design of a rapid validation platform for model-based Powertrain Control Strategies. *SAE Technical Paper Series, I*. <https://doi.org/10.4271/2025-01-5053>
- [19] Campo, K. X., Teper, T., Eaton, C. E., Shipman, A. M., Bhatia, G., & Mesmer, B. (2022). Model-based systems engineering: Evaluating perceived value, metrics, and evidence through literature. *Systems Engineering*, 26(1), 104–129. <https://doi.org/10.1002/sys.21644>
- [20] National Instruments. (n.d.). *What is ni labview? Graphical Programming for Test & Measurement*. What Is NI LabVIEW? https://www.ni.com/en/shop/labview.html?cid=PSEA-701VU00000DbXTpYAN-CA-GOGSE_CA%2BBEN%2BLabVIEW%2BBrand&utm_keyword=labview&s_kwcid=AL%216304%213%21776776036782%21b%21%21g%21%21labview&gad_source=1&gad_campaignid=23074254887&gbraid=0AAAAAD_JoLnjgYUcIs-

eBWK20mkuLU529&gclid=CjwKCAiA8vXIBhAtEiwAf3B-
g6uM36b81lV114zPiPMx2bmp3eNQh0REAalbumAqrp-bGE7A2X4QhBoCywwQAvD_BwE

- [21] Barr, M., & Massa, A. (2007). *Programming Embedded Systems with C and GNU Development Tools* (2nd ed.). O'Reilly Media, Inc.
- [22] Fuller, S., Greiner, B., Moore, J., Murray, R., van Paassen, R., & Yorke, R. (2021). The Python Control Systems Library (python-control). *2021 60th IEEE Conference on Decision and Control (CDC)*. <https://doi.org/10.1109/cdc45484.2021.9683368>
- [23] Ozgur, C., Colliau, T., Rogers, G., & Hughes, Z. (2021). Matlab vs. python vs. R. *Journal of Data Science*, 15(3), 355–372. [https://doi.org/10.6339/jds.201707_15\(3\).0001](https://doi.org/10.6339/jds.201707_15(3).0001)
- [24] Applied Intuition. (n.d.). *CarSim Information*. CarSim Overview. <https://www.carsim.com/products/carsim/index.php>
- [25] IPG Automotive. (n.d.). *CarMaker*. Applications | IPG Automotive. <https://www.ipg-automotive.com/en/applications/>
- [26] Xu, G., Mitchell, N., Arnold, M., Rountev, A., & Sevitsky, G. (2010). Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-Scale Object-Oriented Applications. *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, 421–426. <https://doi.org/10.1145/1882362.1882448>
- [27] Salmony, P. (n.d.). *Top microcontrollers for embedded systems: Octopart: The pulse*. Top Microcontrollers for Embedded Systems. <https://octopart.com/pulse/p/top-microcontrollers-embedded-systems>
- [28] Hoyos-Gutiérrez, J., Cardona-Aristizabal, J., Muñoz-Gutiérrez, P., & Ramirez-Jimenez, D. (2023). A systematic literature review on Rapid Control Prototyping Applications. *IEEE Revista Iberoamericana de Tecnologías Del Aprendizaje*, 18(1), 76–85. <https://doi.org/10.1109/rita.2023.3250559>
- [29] Parai, M. K., Das, B., & Das, G. (2013). An Overview of Microcontroller Unit: from Proper Selection to Specific Application. *International Journal of Soft Computing and Engineering*, 2(6), 228–231.

- [30] International Organization for Standardization. (2024). Road vehicles — Controller area network (CAN) (ISO Standard No. 11898-1:2024). <https://www.iso.org/standard/86384.html>
- [31] International Organization for Standardization. (2025). Road vehicles — Local Interconnect Network (LIN) (ISO 17987-1:2025). <https://www.iso.org/standard/85125.html>
- [32] IEEE standard for local and Metropolitan Area Networks: Overview and Architecture. (2025). *IEEE Std 802-2024*, 1–99. <https://doi.org/10.1109/ieeestd.2025.10935844>
- [33] International Organization for Standardization. (2025). Road vehicles — FlexRay communications system (ISO ISO 17458-1:2013). <https://www.iso.org/standard/59804.html>
- [34] Visconti, P., Giannotta, G., Brama, R., Primiceri, P., Malvasi, A., & Centuori, A. (2017). Features, operation principle and limits of spi and I2C communication protocols for smart objects: A novel spi-based hybrid protocol especially suitable for IoT applications. *International Journal on Smart Sensing and Intelligent Systems*, 10(2), 1–34. <https://doi.org/10.21307/ijssis-2017-211>
- [35] Stuhlfauth, R. (2021). (rep.). *Wireless Communications for Automotive Applications* (Vol. 1, pp. 4–7). Munich, Germany: Rohde & Schwarz.
- [36] GitLab. (n.d.-a). What is version control? <https://about.gitlab.com/topics/version-control/>
- [37] Zolkifli, N. N., Ngah, A., & Deraman, A. (2018). Version control system: A Review. *Procedia Computer Science*, 135, 408–415. <https://doi.org/10.1016/j.procs.2018.08.191>
- [38] Teamwork.com. (n.d.). *Project Management Methodologies*. Project Management Methodologies - Examples & Overview. <https://www.teamwork.com/project-management-guide/project-management-methodologies/?show-chat=true>
- [39] Park University. (2024, October 16). *From agile to waterfall: A breakdown of Project Management Methodologies*. From Agile to Waterfall: A Breakdown of Project Management Methodologies. <https://www.park.edu/blog/from-agile-to-waterfall-a-breakdown-of-project-management-methodologies/>
- [40] Silvius-Zuchi, D., & Silvius, G. (2024). Predictive, adaptive and hybrid project approaches. *The Handbook of Project Management*, 293–318. <https://doi.org/10.4324/9781003274179-25>
- [41] Argonne National Laboratory. (2025). (rep.). *Vehicle Design Rules and Requirements* (pp. 161–162).

- [42] Federation Internationale de l'Automobile . (n.d.). (rep.). *FIA 2026 F1 Regulations - Section C [Technical]* (pp. 78–79).
- [43] International Organization for Standardization. (2018). Road vehicles — Functional safety (ISO Standard ISO 26262-6:2018). <https://www.iso.org/standard/68388.html>
- [44] Data link layer. (2022). *SAE International*, 10, 1–55.
https://doi.org/10.4271/j1939/21_202205
- [45] Weber, M., & Weisbrod, J. (2003). Requirements Engineering in automotive development: Experiences and challenges. *IEEE Software*, 20(1), 16–24.
<https://doi.org/10.1109/ms.2003.1159025>
- [46] Autosar. (n.d.). *Autosar features*. AUTOSAR Features.
<https://www.autosar.org/standards/features>
- [47] Abboush, M., Knieke, C., & Rausch, A. (2024). A virtual testing framework for real-time validation of automotive software systems based on hardware in the loop and fault injection. *Sensors*, 24(12), 3733. <https://doi.org/10.3390/s24123733>
- [48] Szalay, Z. (2021). Next generation X-in-the-loop validation methodology for Automated Vehicle Systems. *IEEE Access*, 9, 35616–35632. <https://doi.org/10.1109/access.2021.3061732>
- [49] Donca, I.-C., Stan, O. P., Misaros, M., Gota, D., & Miclea, L. (2022). Method for continuous integration and deployment using a pipeline generator for Agile Software Projects. *Sensors*, 22(12), 4637. <https://doi.org/10.3390/s22124637>
- [50] The MathWorks. (n.d.). *Use simulink real-time UI to create and execute a real-time application*. MATLAB Help Center. <https://www.mathworks.com/help/slrealtime/build-and-run.html>
- [51] Mauborgne, P., Deniaud, S., Levrat, E., Bonjour, E., Micaëlli, J.-P., & Loise, D. (2016). Operational and system Hazard Analysis in a safe systems requirement engineering process – application to automotive industry. *Safety Science*, 87, 256–268.
<https://doi.org/10.1016/j.ssci.2016.04.011>
- [52] Abdulkhaleq, A., & Wagner, S. (2025, October). *Experiences with Applying STPA to Software-Intensive Systems in the Automotive Domain. STAMP Workshop 2013*. Cambridge; USA.

- [53] Marrone, J. F., Yoo, J., McAfee, A., & Fernando, S. (2023). (tech.). *UWAFt Architecture Selection Proposal*. Waterloo, Ontario: UWAFt.
- [54] General Motors. (2025). (rep.). *LYRIQ Propulsion Systems Overview* (pp. 25–30).
- [55] Nidhra, S. (2012). Black box and white box testing techniques - A literature review. *International Journal of Embedded Systems and Applications*, 2(2), 29–50.
<https://doi.org/10.5121/ijesa.2012.2204>
- [56] MathWorks. (2025, November 11). *What is the Nyquist theorem*. What Is the Nyquist Theorem - MATLAB & Simulink. <https://www.mathworks.com/discovery/nyquist-theorem.html>
- [57] Vector Informatik GmbH. (2025). *Analyzing Automotive ECUs and Networks With CANalyzer*. CANalyzer | ECU & Network Analysis | Vector.
<https://www.vector.com/int/en/products/products-a-z/software/canalyzer/>
- [58] Vector Informatik GmbH. (2025). *CAPL Used With CANoe And CANalyzer*. The use of CAPL - Article | vector. <https://www.vector.com/at/en/know-how/capl/>
- [59] Kytainyk, V. (2024, August 13). *Software development time estimation: How long should it take to develop a product?*. Forbes.
<https://www.forbes.com/councils/forbesbusinesscouncil/2022/12/02/software-development-time-estimation-how-long-should-it-take-to-develop-a-product/>