Automatic Generation of vTESTstudio Test Case from Natural Language Requirements Using Large Language Models

Master's thesis in Information and Communication Technology

Shuaixin Pan & Tianqi Zhao

DEPARTMENT OF Electrical Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Automatic Generation of vTESTstudio Test Case from Natural Language Requirements Using Large Language Models

Shuaixin Pan & Tianqi Zhao

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Automatic Generation of vTESTstudio Test Cases from Natural Language Requirements Using Large Language Models
Shuaixin Pan & Tianqi Zhao

Supervisor: Siddhant Gupta, Volvo Cars AB
Supervisor: André Bezerra de Freitas Diniz, Department of Electrical Engineering
Examiner: Paolo Monti, Department of Electrical Engineering

Master's Thesis 2026
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 72 782 2552

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Automatic Generation of vTESTstudio Test Cases from Natural Language Requirements Using Large Language Models
Shuaixin Pan & Tianqi Zhao
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Automotive testing still involves substantial manual effort when natural-language requirements are translated into tool-compatible test artefacts. As vehicle functions, carlines and software releases increase, this translation step becomes a growing bottleneck in test development.
This thesis investigates a two-phase pipeline for generating vTESTstudio-compatible test artefacts from natural-language automotive requirements using Large Language Models (LLMs). The first phase converts a requirement into a structured intermediate representation of logical test steps. The second phase grounds these steps in valid domain resources, including signals and reusable functions, before constructing the final Vector Test Table (VTT) artefact. This separation makes the generation process easier to inspect, evaluate and control. The thesis further studies retrieval-based grounding, parameter-efficient fine-tuning for intermediate representation generation, and retrieved skill guidance for improving logical planning.
On the small evaluation sets used in this thesis, the proposed pipeline produced useful test artefacts in selected cases, but human review remained necessary. This is assessed up to the generated logical test steps and the coverage of the VTT artefacts against reference cases, not their syntactic validity or execution in vTESTstudio or CANoe. Within these limits, retrieval tended to improve grounding indomainness and signal coverage, while fine-tuning improved the validity, consistency and domain style of intermediate representations. Retrieved skills helped planning-oriented aspects such as logical adequacy and structural quality, though larger skill contexts could make downstream grounding harder. Overall, the thesis suggests that requirement-driven automotive test generation is more controllable when requirement interpretation, domain grounding and artefact construction are treated as separate stages.

Keywords: Automotive software testing, Testcase Generation, Large Language Models(LLMs), Retrieval-augmented generation(RAG), Intermediate representation, vTESTstudio, Skill-guided prompting.

Acknowledgements

We would like to thank our academic supervisor, André Bezerra de Freitas Diniz, for the guidance, feedback, and continuous support throughout this thesis work. The suggestions and comments during the project helped us plan the thesis process and improve the quality of the final report.
We are also grateful to our company supervisor, Siddhant Gupta, at Volvo Cars AB for giving us the opportunity to carry out this thesis in an industrial setting. His guidance and practical feedback helped us better understand the automotive testing workflow and the challenges involved in translating natural-language requirements into executable test artefacts.
We would also like to thank Zihan Liu at Volvo Cars AB for his technical support, especially with access requests, permission handling, and cloud-related setup. His support helped us work more effectively within the technical environment required for this thesis.
Our thanks also go to the colleagues in Exterior Functions at Volvo Cars AB for their support, discussions, and willingness to share practical knowledge about requirements, test cases, signals, functions, and the use of vTESTstudio in practice. Their input helped ground the thesis work in a realistic industrial context.
We would like to acknowledge Chalmers University of Technology and the Department of Electric Engineering for providing the academic environment and resources that made this thesis possible. We also thank Paolo Monti for reviewing the work and providing academic guidance during the thesis process.
Finally, we would like to thank our families and friends for their encouragement, patience, and support throughout the project.

Tianqi Zhao
Gothenburg, May 2026

Shuaixin Pan
Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms used throughout this thesis, listed in alphabetical order:

AI	Artificial Intelligence
API	Application Programming Interface
BF16	Brain Floating Point 16-bit
BLEU	Bilingual Evaluation Understudy
BM25	Best Matching 25
CDF	Cumulative Distribution Function
CUDA	Compute Unified Device Architecture
ECU	Electronic Control Unit
GPU	Graphics Processing Unit
IR	Intermediate Representation
JSON	JavaScript Object Notation
LLM	Large Language Model
LoRA	Low-Rank Adaptation
ML	Machine Learning
MRR	Mean Reciprocal Rank
NCCL	NVIDIA Collective Communications Library
NF4	NormalFloat 4-bit
PEFT	Parameter-Efficient Fine-Tuning
QLoRA	Quantized Low-Rank Adaptation
RAG	Retrieval-Augmented Generation
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
SDK	Software Development Kit
SFT	Supervised Fine-Tuning
TRL	Transformer Reinforcement Learning
TSS	Three-Stage Score
VTT	Vector Test Table
XML	Extensible Markup Language

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables used throughout this thesis. Symbols that occur only within a local evaluation formula are defined in the surrounding text rather than repeated here.

Indices

h	Index for a historical requirement–testcase case
h'	Historical-case index used when normalising BM25 scores
p	Purpose-field subscript
d	Description-field subscript
s	Signal-field subscript
k	Rank cut-off used in Recall@k and top-k retrieval
c	Index for a candidate signal or function entry in Phase 2 retrieval

Sets

$\mathcal{H}$	Set of historical requirement–testcase cases in the retrieval store
$\mathcal{C}$	Set of candidate entries (signals or functions) searched during Phase 2 retrieval
Q	Set of retrieval queries used when computing Recall@k and Mean Reciprocal Rank (MRR)

Parameters

w_p	Weight assigned to purpose-field similarity in the historical-case retrieval score
w_d	Weight assigned to description-field similarity in the historical-case retrieval score

w_s	Weight assigned to signal-field similarity in the historical-case retrieval score
α	Mixture coefficient between BM25 lexical matching and dense similarity; used in Phase 1 signal scoring and Phase 2 candidate retrieval
α_{LoRA}	Scaling parameter used by the adapter during QLoRA fine-tuning
r	Adapter rank used during QLoRA fine-tuning
d_{out}	Output dimension of a model layer adapted with LoRA
d_{in}	Input dimension of a model layer adapted with LoRA
δ	Value-match bonus added to a Phase 2 candidate score when the target value appears in its allowed-values string
τ	Pass threshold used in Pass@τ for the three-stage score

Variables

q_t	Input requirement or query text
$\mathbf{q}_v$	Dense query vector produced from the input requirement text
$E(\cdot)$	Embedding function used to encode text into dense vectors
$\mathbf{W}$	Adapted weight matrix after applying the LoRA update
$\mathbf{W}_0$	Frozen base-model weight matrix in LoRA adaptation
$\Delta\mathbf{W}$	Low-rank weight update learned by the LoRA adapter
$\mathbf{A}$	Down-projection LoRA adapter matrix
$\mathbf{B}$	Up-projection LoRA adapter matrix
N_h	Tokenised signal-name document for historical case h
$S(q_t, h)$	Overall historical-case retrieval score for query q_t and case h
$S_{\text{sig}}(q_t, h)$	Hybrid signal-field retrieval score for query q_t and case h
S_{adequacy}	Logical-adequacy score used in the end-to-end pipeline evaluation
S_{rule}	Rule-based component of the logical-adequacy score
$S_{\text{structural}}$	Structural component of the logical-adequacy score
S_{domain}	Domain-fit component of the logical-adequacy score

S_{coverage}	Combined behavioural-coverage score used in the end-to-end pipeline evaluation
S_{step}	Step-coverage component of the behavioural-coverage score
S_{signal}	Signal-coverage component of the behavioural-coverage score
Q_1	Stage 1 quality representative used in the three-stage score
Q_2	Stage 2 grounding representative used in the three-stage score
Q_3	Stage 3 coverage representative used in the three-stage score
$\text{BM25}(q_t, N_h)$	Raw BM25 lexical score between query q_t and signal-name document N_h
$\widetilde{\text{BM25}}(q_t, N_h)$	Query-normalised BM25 score for historical case h
d_z	Standardised paired-effect size reported for Wilcoxon comparisons
$\mathbf{v}_p^h$	Stored purpose vector for historical case h
$\mathbf{v}_d^h$	Stored description vector for historical case h
$\mathbf{v}_s^h$	Stored signal-related vector for historical case h
$\cos(\cdot, \cdot)$	Cosine similarity between two vectors
s_t	Intent text of a logical step, used as the query in Phase 2 retrieval
$\mathbf{s}_v$	Dense embedding vector produced from a step intent text s_t
$\mathbf{c}_v$	Precomputed description vector for candidate c
N_c	Tokenised description document for candidate c, used in BM25 matching
$R(s_t, c)$	Phase 2 hybrid retrieval score for step intent s_t and candidate c
$\hat{R}(s_t, c)$	Bonus-adjusted Phase 2 retrieval score after applying the value-match bonus

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

Electronic Control Units (ECUs) and distributed vehicle functions have become increasingly complex in modern automotive software. Functions in areas such as body and exterior behaviour may involve several interacting components, signals, states and timing conditions. As a result, verification work has to handle frequent software changes while preserving traceability between requirements and executable test artefacts.
Many automotive test cases still start from natural-language requirements. A test engineer reads the requirement, identifies preconditions, triggers and expected outcomes, and then translates this understanding into the structures supported by the test environment. In this thesis, the target environment is Vector's vTESTstudio workflow. vTESTstudio is Vector's test-design environment for creating and maintaining automated ECU tests, commonly used together with CANoe for simulation and test execution [1]. In this setting, test behaviour has to be expressed through ordered actions, checks, waits and function calls that can be represented as vTESTstudio-compatible Vector Test Table (VTT) artefacts.
Large Language Models (LLMs) make it possible to treat this translation as a generation task. The task is constrained by more than language quality: the generated artefact has to preserve the requirement intent, refer to valid signals and functions, and follow the structure expected by the industrial test toolchain.

1.2 Problem Description

The practical bottleneck addressed in this thesis is the gap between requirement text and vTESTstudio-compatible test design. Manual translation consumes engineering time, can introduce inconsistencies between requirement intent and implementation, and limits the reuse of existing test knowledge.
This is difficult to automate directly. Requirements may omit preconditions that are obvious to experienced testers, describe several behaviours in one sentence, or use terminology that does not exactly match the signal and function resources available in the test environment. At the same time, generated test artefacts are sensitive to small structural mistakes: an output that looks plausible as text may still be unusable if it refers to an invalid signal, misses a required check, or violates the expected VTT structure.
The problem is therefore not to generate a readable test description. The generated

output has to be inspectable test logic that is grounded in the available domain resources and can be turned into a vTESTstudio-compatible artefact.

1.3 Research Gap and Contribution

Existing work has explored LLM-based test generation in both general software testing and automotive test automation. General software-testing approaches mainly focus on source-code-level unit tests, where the input is usually a function, class or program fragment. Automotive studies are closer to the setting of this thesis, but they often start from test specifications that are already organised into steps, or examine how generative AI assistants can be integrated into existing test automation workflows.
Less attention has been given to requirement-driven generation where the input is a natural-language automotive requirement and the output must remain compatible with an industrial test-development environment. This setting requires the system to infer test logic, ground it in valid domain resources, and produce an artefact that engineers can inspect and validate.
This thesis addresses the gap through a two-phase pipeline that generates vTESTstudio-compatible test artefacts from natural-language automotive requirements. The first phase converts a requirement into a structured intermediate representation of the intended test logic. The second phase grounds this representation in valid signal and function resources, and then constructs the final VTT artefact. Separating requirement interpretation from tool-specific artefact construction makes the generated logic easier to inspect and evaluate before the final artefact is produced.

1.4 Aim

The aim of this thesis is to design, implement and evaluate an AI-assisted pipeline that generates structured test-case representations and vTESTstudio-compatible artefacts from natural-language automotive requirements. The work focuses on a staged generation process in which requirements are first converted into logical test steps and then grounded in available signal and function resources before final VTT construction.

1.5 Research Questions

The thesis treats requirement-to-test generation as a pipeline problem rather than a single text-generation task. The generated output first has to capture the intended test logic, then be grounded in valid domain-specific signals and functions, and finally be evaluated as a test artefact rather than only as generated text. From this view, the thesis is guided by the following research questions:

- RQ1: How can natural-language automotive requirements be automatically transformed into vTESTstudio-compatible test artefacts using LLMs?

- RQ2: How can LLM-generated test artefacts be grounded in valid domain-specific signals and functions?
- RQ3: How does parameter-efficient fine-tuning affect the structural validity, type consistency, and domain style of generated intermediate representations?
- RQ4: How can recurring generation errors and implicit domain conventions be addressed without modifying model weights?
- RQ5: How should LLM-generated automotive test artefacts be evaluated beyond textual similarity?

1.6 Scope and Limitations

The thesis focuses on selected automotive requirements and the corresponding historical test material available in the project context. It covers the construction of a logical-step intermediate representation, retrieval of historical cases and domain resources, supervised fine-tuning for requirement-to-logical-step generation, skill-guided prompting, grounding of generated steps, and construction of vTESTstudio-compatible VTT artefacts.
The work does not aim to replace human test engineers or provide a certified safety process. It also does not cover full industrial deployment, long-term maintenance of the generated tests, or execution of all generated artefacts in a complete vehicle-in-the-loop or hardware-in-the-loop environment. The evaluation is limited to the data, signals, functions and requirement styles available within the project.
The evaluation reaches the logical test steps produced in Phase 1 and a coverage comparison of the generated VTT artefacts against reference test cases. It does not check the syntactic validity of the generated artefacts, nor execute them in vTESTstudio or CANoe. Whether a generated artefact is syntactically accepted by the toolchain and runs correctly against a simulated or real ECU is therefore outside the scope of this thesis.
All requirement identifiers, test case names, signal and function names, and system-internal references appearing in examples throughout this thesis have been anonymised to comply with confidentiality requirements. The examples are derived from real industrial data but do not reveal proprietary implementation details.

1.7 Thesis Structure

Chapter 2 introduces the theoretical background and related work on requirements-based testing, LLMs, retrieval-augmented generation, intermediate representations and LLM-based test generation. Chapter 3 describes the data preparation, pipeline design, model configurations and evaluation methodology. Chapter 4 presents the experimental results for retrieval, fine-tuning, end-to-end pipeline quality and skill retrieval depth. Chapter 5 summarises the contributions, discusses the main findings and limitations, and outlines directions for future work.

2
Theory

The core challenge addressed in this thesis is generating executable test artefacts from natural-language automotive requirements. Requirements are informal and domain-specific, the target output has to follow a tool-specific format, and generated content has to reference valid signals and functions instead of freely invented entities. The problem therefore involves both linguistic interpretation and domain grounding. This chapter introduces the concepts needed to address it. Requirements engineering and the vTESTstudio context define the problem setting. LLMs, Retrieval-Augmented Generation (RAG) and Intermediate Representations (IRs) then provide the mechanisms used for interpretation, grounding and structured generation, and iterative correction supports the reliability of the generated output. The chapter ends with a review of related work that places the proposed pipeline among existing LLM-based test generation approaches.

2.1 Requirements Engineering and Automated Test Generation

The input to this thesis is a natural-language automotive requirement, so the first challenge is understanding what makes such requirements difficult to process automatically. Requirements engineering provides the conceptual vocabulary: requirements are commonly expressed in natural language to remain accessible to multiple stakeholders, but this informality introduces ambiguity, implicit assumptions, and varying levels of detail that make their direct use in automated test generation unreliable [2, 3].

In this thesis, a requirement refers to a natural-language specification of expected system behaviour under defined conditions. In practice, such requirements are often formulated in patterns such as “if ... , ... shall be ...” or “when ... , then ... shall ...” [4]. They may specify that a signal or system state shall assume a certain value, remain in a valid state, or transition to another value after a given trigger. In more complex cases, one condition may lead to coordinated changes across multiple related signals or states.

A test case can be understood as a structured verification artefact derived from one or more requirements. Unlike the requirement itself, which remains descriptive, a test case has to define the relevant conditions, triggers, observable signals or states, and expected outcomes used to decide whether the requirement is satisfied. The relationship is not always one-to-one in practice, since a single requirement may need to be verified through multiple scenarios. Automated test generation is therefore

not only a matter of rephrasing text; it is a process of turning informal behavioural descriptions into structured and testable verification logic.
The transformation is non-trivial for several reasons. Natural-language requirements may omit details that are obvious to a human expert but are not explicit enough for automated processing. The same intended behaviour can also be expressed in many different ways, which makes purely rule-based conversion unreliable. Structured test artefacts also have to satisfy both semantic and syntactic constraints: they need to reflect the original requirement correctly and be precise enough for downstream execution.
This makes requirement-to-test generation a structured transformation problem. It requires converting informal behavioural specifications into explicit verification logic that remains faithful to the original requirement and is precise enough for automated testing [5]. Figure 2.1 illustrates the decomposition for a compact automotive requirement. The need for both semantic interpretation and executable precision motivates the use of language models for requirement interpretation, and also calls for mechanisms that constrain and validate the generated test logic.

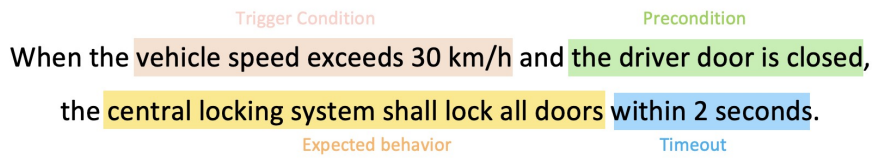
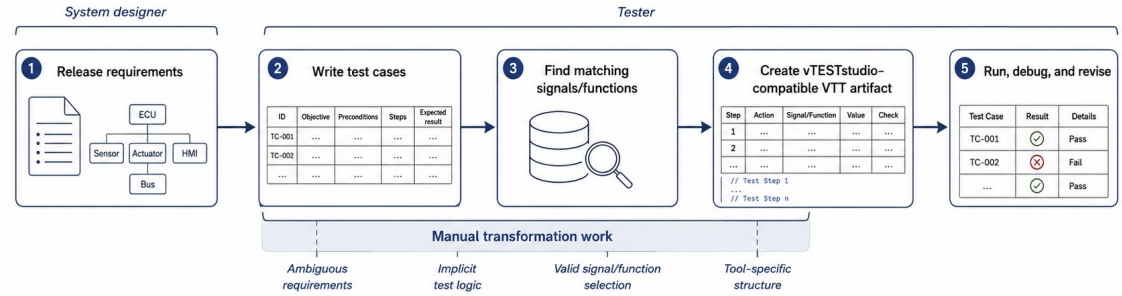


Figure 2.1: Decomposition of a natural-language automotive requirement into structured test-intent elements. The decomposition separates triggering conditions, preconditions, expected behaviour, timing constraints, and candidate grounding entities before platform-specific VTT construction.

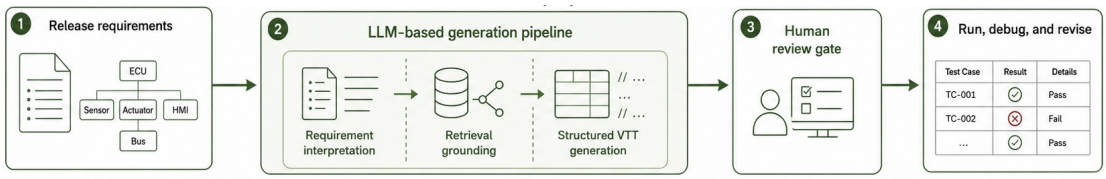
2.2 Automotive Software Testing and vTESTstudio / VTT

The target output of this thesis must be compatible with an industrial automotive testing workflow, which means understanding what that workflow requires and what constraints it imposes on generated test content. Automotive software testing covers distributed electronic control units, communication networks, and vehicle variants [6], making it not only a final quality check but a central verification activity throughout development. Systematic processes, traceability from requirements to tests, and repeatable execution are therefore central requirements [7].
These conditions make requirements-based testing particularly important. In practice, many automotive test cases are derived from requirement specifications and related design artefacts rather than directly from source code, so the quality of the resulting test case specification depends on the quality and availability of those inputs. Empirical studies report recurring problems such as incomplete or late requirements, ambiguous descriptions, outdated signal definitions, and high communication effort between test designers and testers [8]. In this thesis, such problems directly affect the transformation from natural-language requirements into executable test

artefacts.
Automotive testing also relies on structured test descriptions that can be reused across validation levels. This has motivated work on model-based and automated testing, where systematic design, automation, and reuse are important for systems described by signals, states, and time-dependent behaviour [9]. Shin and Lim [10] likewise show that automatic test case generation can reduce manual effort by converting structured development artefacts into software and hardware test cases. The main challenge is therefore not to produce textual test descriptions, but to create structured test representations that can be implemented and executed reliably.
In this thesis, the target representation is a test artefact used in Vector Informatik's automotive testing toolchain. In this toolchain, `vTESTstudio` is used to design automated test cases, while `CANoe` supports test execution and simulation for ECU testing [11]. The generated artefact follows the VTT format, a table-based representation of executable test behaviour consisting of ordered steps, signal assignments, checks, waits, and function calls.



(a) Current manual workflow.



(b) LLM-assisted workflow considered in this thesis.

Figure 2.2: Overview of the manual requirement-to-test workflow and the automation scope addressed in this thesis. The proposed LLM-based pipeline targets the repetitive transformation from natural-language requirements to grounded, `vTESTstudio`-compatible test artefacts, while keeping human review and execution feedback in the loop.

Figure 2.2 illustrates the scope of the problem. The goal is not only to interpret a requirement correctly, but to produce an artefact that is compatible with an industrial automotive testing workflow. The generated output has to preserve the behavioural meaning of the original requirement and ground the result in concrete testing entities such as signals, values, timing constraints, and executable step structure. This is the main motivation for the staged design used in the project, where requirement understanding and test-script construction are treated as separate steps rather than a single unconstrained generation task. The first of these steps requires a model

that can interpret natural-language requirements and extract structured test logic, which motivates the use of LLMs.

2.3 Large Language Models and Model Adaptation

Interpreting natural-language requirements and extracting structured test logic from them requires a model that can capture conditional relationships and behavioural semantics. LLMs fit this need: they have shown strong capabilities in natural-language understanding, structured information extraction, and code-like generation [12], which makes them well suited to a task where behavioural descriptions have to be turned into explicit verification logic. LLMs can capture trigger-response patterns and dependencies between conditions, signals, states, and expected outcomes, all of which are common in requirements-based testing.

Despite these strengths, general-purpose LLMs are not inherently reliable in domain-specific engineering settings [13]. They may produce outputs that are syntactically plausible but semantically inaccurate, omit important constraints, or reference signal and state names that do not exist in the target domain. This limitation is especially relevant in automated test generation, where correctness depends not only on fluent text generation but on faithful and controlled interpretation of the original requirement.

To improve task-specific performance, LLMs are commonly adapted through supervised fine-tuning [14] and related post-pretraining methods [15]. Supervised fine-tuning helps the model learn more consistent mappings between requirement descriptions and structured target representations, which is useful when the output must follow predefined verification logic or schema constraints. Parameter-efficient adaptation methods reduce fine-tuning cost by keeping the original model weights mostly fixed and training only additional lightweight parameters. Low-Rank Adaptation (LoRA) achieves this by adding low-rank trainable adapter matrices to selected model layers. Concretely, the weight update is approximated as a low-rank decomposition (Eq. 2.1). Quantized Low-Rank Adaptation (QLoRA) extends LoRA by using a quantized base model in 4-bit precision during adapter training, which further reduces memory requirements [16].

$$\mathbf{W} = \mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \frac{\alpha_{\text{LoRA}}}{r}\mathbf{BA}, \quad \mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r},\ \mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}},\ r \ll \min(d_{\text{out}}, d_{\text{in}}). \tag{2.1}$$

where $\mathbf{W}_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ denotes the frozen pretrained weights, $\mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}$ and $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}$ are the trainable low-rank matrices with rank $r \ll \min(d_{\text{out}}, d_{\text{in}})$, and α_{LoRA} is a scaling factor that controls the magnitude of the low-rank update relative to the original weights.

For requirement-to-test generation, the LLM therefore acts as a semantic transformation model rather than a text-fluency engine: it turns behavioural requirement descriptions into more explicit and testable forms. Model adaptation alone is not always sufficient for grounded and reliable generation in domain-specific settings,

which motivates the retrieval mechanisms described in the following section.

2.4 Retrieval-Augmented Generation and Embedding Retrieval

Although LLMs can capture complex semantic relations in natural-language requirements, their parametric knowledge is not sufficient for reliable generation in domain-specific engineering tasks. In requirement-to-test generation, correctness depends not only on linguistic understanding but also on access to external domain knowledge, such as valid signal names, function definitions, value conventions, and previously used test patterns. Retrieval-Augmented Generation (RAG) addresses this by retrieving relevant external knowledge and incorporating it into the model input before generation [17].
RAG combines information retrieval with conditional text generation, so the model can produce outputs from both the input requirement and supporting evidence from an external knowledge source. A common retrieval mechanism is embedding-based retrieval, in which texts are mapped into dense vector representations so that semantically similar items are located close to one another in the vector space [18, 19]. Unlike exact keyword matching, this lets the system retrieve relevant examples even when similar requirement meaning is expressed with different wording, abstraction levels, or domain-specific terminology.
For requirement-to-test generation, retrieval provides both semantic and domain grounding. It can supply examples of how similar requirements have been translated into structured test representations, and it exposes the model to valid engineering entities and terminology from the target domain [20]. This is particularly important in automotive testing, where generated outputs have to align with existing signal definitions, valid system states, and executable test scripts. Figure 2.3 contrasts unsupported model-only generation with retrieval-grounded generation for the same requirement. Retrieval alone is not enough for controlled test generation, because the retrieved information still has to be transformed into a structured form suitable for downstream execution. This motivates the use of IRs and constrained structured generation.

2.5 Intermediate Representation and Constrained Structured Generation

A key design choice in this thesis is to avoid direct end-to-end generation of VTT from a natural-language requirement. A direct mapping would force the model to interpret the requirement, derive the test logic, select signals and functions, map values, and produce platform-specific syntax in a single step. This increases the search space and makes errors harder to inspect. Research on coarse-to-fine generation and controlled intermediate forms supports a staged approach, where higher-level meaning is produced before lower-level realization [21, 22].
In this thesis, two intermediate representations are used to decompose the task. The

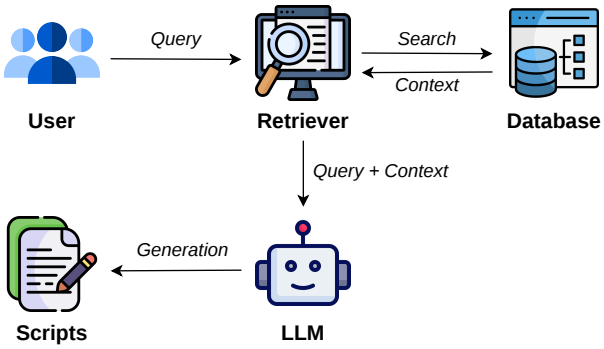


Figure 2.3: Conceptual comparison of requirement-to-test generation with and without retrieval grounding. Without retrieved domain context, the model may produce plausible but unsupported signal names or incomplete test logic. With RAG, retrieved signal definitions and similar test patterns constrain the generated steps toward valid, domain-grounded test artefacts.

first captures test intent in a structured but syntax-free form, separating semantic content from tool-specific formatting. The second is produced after retrieval and encodes grounded step content using candidate signals and functions rather than freely generated names, turning part of the generation into constrained selection rather than unconstrained text production. Similar arguments appear in syntax-aware and structure-aware generation, where separating meaning from final form has been shown to reduce malformed executable output [23, 24]. The specific design of both representations is described in Chapter 3.
Work on constrained decoding shows the value of explicit structural limits for formal outputs [25], and recent studies report similar gains from IRs in
generation [26]. The staged design used in this thesis is therefore easier to control and debug than direct requirement-to-VTT generation. Structured representations alone do not eliminate all generation errors, which is why iterative feedback and reusable skill guidance are also used as correction mechanisms.

2.6 Iterative Self-Reflection and Skill Memory System

Even with structured generation and retrieval support, a single-pass output may still miss checks, use inconsistent signals, or omit useful test patterns. Iterative self-reflection addresses this by evaluating an initial output and using explicit feedback to guide a revised attempt. Recent work shows that language models can improve through such feedback loops without modifying model weights [27, 28]. Chain of Hindsight further demonstrates that language feedback can be preserved and reused to improve later outputs [29].
In this thesis, reflection is used as a practical correction mechanism for a specific class of residual errors. An LLM adapted for requirement interpretation can capture the primary trigger-action relationship in a requirement, but it may still miss steps

that a domain expert would add implicitly, such as preconditions needed to reach a valid initial state, ordering constraints between verification steps, or cleanup actions that restore the system afterwards. Such implicit expectations are often absent from the requirement text itself, so they are unlikely to be fully recovered through fine-tuning or retrieval alone. A system that generates a candidate output, identifies these problems against the task requirements, and then revises the result can correct such residual errors without retraining the model.
Useful feedback should also persist beyond a single query. Voyager shows the value of an expanding skill library [30], and Generative Agents shows how past observations can be summarized into higher-level reflections for later use [31]. This motivates a skill memory system as an optional enhancement to the present project: reflections can be distilled into compact rules and examples, retrieved for similar requirements, and injected into the prompt as reusable guidance. This mechanism aims to improve consistency and reduce repeated errors across related cases.

2.7 Related Work on Test Generation with LLMs

While the previous sections introduced the theoretical concepts underlying this thesis, this section situates the work in relation to existing studies on LLM-based test generation. It first reviews general approaches to test generation with LLMs, focusing on context selection, validation, and assertion correctness. It then discusses generative AI applications in automotive test automation, where generated artefacts must be grounded in domain-specific signals, tools, and workflows. The section ends by summarising the remaining gap addressed by this thesis.

2.7.1 General LLM-Based Test Generation

Automated test generation has been studied long before the recent use of LLMs. Traditional approaches include feedback-directed random testing and search-based test generation. Randoop uses feedback obtained during test execution to guide the generation of method-call sequences and to avoid redundant or invalid tests [32]. EvoSuite is a search-based approach for Java programs that generates whole test suites and optimises them towards coverage criteria, with assertions added to capture observed program behaviour [33]. These techniques show that test generation can be automated to a large extent, but they mainly operate on source code and tend to optimise for coverage rather than requirement-level intent or domain-specific test logic.
Recent studies have explored LLMs for test generation. TestPilot evaluates the use of LLMs for automated unit test generation in JavaScript and TypeScript packages, using function signatures, implementations, and usage examples as prompt context [34]. ChatUniTest further addresses practical limitations of LLM-based unit test generation by introducing adaptive focal context construction and a generation-validation-repair workflow, including syntactic, compilation, and runtime validation [35]. A3Test focuses on assertion quality and test signature correctness, showing that assertion-specific knowledge and verification mechanisms can improve the correctness of generated test cases [36].

These studies are relevant because they show that LLM-based test generation depends on context selection, validation, repair, and correctness checks beyond textual similarity. However, they mainly target source-code-level unit tests. Their inputs are typically focal methods, classes, or project code, and their outputs are unit tests in general-purpose programming languages. In contrast, this thesis focuses on requirement-driven automotive test generation, where the input is a natural-language requirement and the output must be compatible with a specific industrial testing workflow.

2.7.2 Generative AI for Automotive Test-Script Generation

Generative AI has also been investigated in automotive test automation, where generated artefacts have to be grounded in domain-specific signals, parameters, test platforms, and toolchains. Wynn-Williams et al. study the conversion of informal automotive test case specifications into executable ecu.test scripts [37]. Their approach uses retrieval-augmented generation and few-shot prompting to generate test instructions from prompts that contain both the informal test step and retrieved context. The evaluation is based on 200 unique pairs of informal test step descriptions and executable test instructions, and the results show that the quality of generated scripts depends strongly on prompt design, model choice, and context accuracy.
That study is closely related to the problem addressed in this thesis because it also looks at generative AI for automotive test-script generation. Its input, however, consists of test management specifications that are already structured into test steps with summaries and details, so the task is closer to translating informal but step-structured test procedures into executable test instructions. This thesis instead focuses on turning natural-language automotive requirements into structured test logic before generating vTESTstudio-compatible artefacts.
Karlsson et al. investigate the integration of GitHub Copilot into an automotive Hardware-in-the-Loop testing framework, focusing on Volvo's Test Automation Framework [38]. Their action research study compares AI-generated test cases with manually written ones and evaluates the challenges of integrating generative AI into existing industrial testing workflows. The study shows that generative AI can support test-script creation, but also that functional correctness, reliability, and workflow integration remain important challenges.
Together, these automotive studies show that generative AI can assist industrial test automation, but they also highlight the need for grounding, validation, and human oversight. In automotive settings, generated tests must not only resemble existing scripts; they must also use valid domain entities, follow tool-specific conventions, and remain inspectable by engineers.
The studies reviewed in Sections 2.7.1and 2.7.2 cover two related but distinct directions of LLM-based test generation. In general software testing, the main focus has been on source-code-level unit tests, where the input is usually a function, class, or program fragment and the generated output must compile, execute, and contain meaningful assertions. In automotive test automation, the setting is closer to this thesis, but existing work often starts from test specifications that are already organised into steps, or studies how generative AI assistants can be integrated into

existing Hardware-in-the-Loop testing workflows. Less attention has been given to requirement-driven generation in which the input is a natural-language automotive requirement and the output must remain traceable to that requirement while being executable within the vTESTstudio toolchain. This gap motivates the staged design used in this thesis, where requirement interpretation is separated from domain grounding and final vTESTstudio artefact construction.

3
Methodology

This chapter describes the methodology used to address the research problem introduced in the previous chapter. As discussed in Chapter 2, a key bottleneck in the current automotive testing workflow is the manual transformation of natural-language requirements into executable test cases for vTESTstudio. This process requires engineers to interpret requirement intent, identify relevant signals and functions, and implement the corresponding test logic in a tool-compatible format.
This thesis proposes a generation pipeline that separates requirement interpretation from final test-script construction, with the aim of reducing manual workload while preserving structural validity and domain grounding. Instead of generating a complete VTT file directly from a requirement, the pipeline first structures the requirement logic and then grounds it in the available industrial signals and functions before producing the final artefact. The following sections describe the data preparation process, pipeline design, skill memory mechanism, model configuration, fine-tuning settings, and evaluation methods used in this thesis.

3.1 Data Sources and Preparation

This section describes the data used in the project and how it was prepared for training, retrieval, and test generation. Three groups of data formed the basis of the project: historical requirement–testcase pairs, signal catalogues, and reusable test functions. Text embeddings were later created from requirement descriptions, signal descriptions, and function descriptions so that the same data could be searched during retrieval. The preparation step kept the original test logic, but represented it in a form that could be used consistently by the generation pipeline.

3.1.1 Requirement–Testcase Pairs

The main dataset was collected from the organisation's requirements management system and test-script repository. Each requirement record typically contained a requirement identifier, a short name, and a natural-language description of the expected software behaviour. When available, the original purpose statement was retained. For requirements without an explicit purpose field, a short purpose statement was added during preprocessing to summarise the main test objective. The corresponding test case was stored as an ordered sequence of executable steps in vTESTstudio format. These steps included signal assignments, signal checks with timeout conditions, waits, and calls to reusable helper functions.

The two sources were connected through requirement-to-testcase references already present in the requirements management system. For each requirement, the referenced test case identifier was resolved against the test-script repository. If the test case could be found, the requirement and its test sequence were kept for further processing. Cases with missing test case references, duplicate mappings, or framework-level test scripts that did not represent requirement behaviour were removed.
Some test cases contained several smaller scenarios under one larger test objective. In these cases, each scenario was treated as a separate data sample. For example, a test objective that checked whether a door lock could be opened and closed correctly could contain separate scenarios for the left and right side. Splitting these cases at scenario level produced shorter sequences and reduced repeated setup or verification steps. This also made the training samples more focused, since very long sequences with near-duplicate steps tended to obscure the underlying logical structure.

3.1.2 Logical-Step Representation

The historical test cases were available as vTESTstudio test scripts and stored in the `.vtt` format. In the collected repository, these files followed an Extensible Markup Language (XML)-based structure. A `.vtt` file contains both the test procedure and tool-specific information required by vTESTstudio, including metadata, execution settings, nested XML elements, report information, and references to signals or functions.
Although this representation is suitable for execution in vTESTstudio, it is not convenient as a learning target for the language model. The XML structure is verbose, sensitive to small tag or attribute errors, and difficult to inspect manually. In addition, many XML elements describe tool configuration or reporting behaviour rather than the underlying test intent. Directly generating complete `.vtt` files from natural-language requirements would therefore mix requirement interpretation with low-level file-format construction.
For this reason, the VTT scripts were abstracted into ordered `logical_steps`. A logical step represents the intended meaning of one test action in a compact form, without the surrounding XML syntax or tool-specific details. The representation was based on recurring action types observed in the historical scripts, including signal assignments, signal checks with expected values and timeouts, explicit waits, and calls to reusable helper functions. These action types later formed the basis of the generation pipeline, where the model generated logical steps first and the final VTT structure was produced deterministically afterwards.
The preprocessing was carried out in five stages:

1. Requirements were paired with their referenced test cases using the cross-references maintained in the requirements management system.
2. Non-procedural entries, such as report headers and comments, were removed from the step sequence when they did not affect the test logic.
3. Signal-related steps were normalised by resolving encoded signal descriptors and rewriting the action as either a signal-setting step or a verification step (see Table 3.2 for an example).

Table 3.1: Logical-step types used in the intermediate representation.

Type	Description
`set`	Set a signal to a specified value.
`verify`	Check that a signal reaches an expected value within a given timeout (milliseconds).
`call`	Invoke a reusable helper function.
`intent`	A short statement describing the purpose of one or more subsequent steps, derived from an informative comment in the original script. Optional; omitted when no such comment existed.
`expected`	A statement describing the expected outcome after preceding steps, derived from an informative comment. Optional; omitted when no such comment existed.

4. Function calls were linked to short descriptions from the function library described below.
5. The resulting records were checked for duplicate entries and malformed test case names before being assigned dataset identifiers.

Each logical step was assigned one of five types, summarised in Table 3.1. The `set`, `verify`, and `call` types correspond directly to the three main executable action categories found in the historical test scripts. The `intent` and `expected` types were derived from informative comments embedded in the original test steps; when such comments were absent for a given step, the field was left empty. These two types were not required for every step but provided additional context when available.

Table 3.2 gives a simplified example of the full conversion from raw test material to logical steps. The names are fictitious, but the structure reflects the kind of information found in the original material.

The example shows how the logical-step representation was derived from the original test material. The processed sequence preserves the order and intent of the executable test case, while removing details specific to the original VTT/XML structure.

3.1.3 Signal Catalogues

Two signal catalogues were used: one for database signals (`DBSignal`) and one for system variables (`SysVar`). Both catalogues were kept fixed throughout the project, as the signal names and allowed values had to match the existing test environment. The `DBSignal` catalogue listed bus-level signals exchanged on the vehicle network. Each entry contained the signal name together with the network, configuration and frame information needed to address the signal in the test environment, and a value field describing its admissible values. This value field took one of two forms: an enumerated set of named states, or a numeric range accompanied by a scaling factor, an offset and a physical unit. The `SysVar` catalogue listed system variables used by the test platform. Each such entry was identified by a variable name and a namespace path, and was likewise associated with a value field.

The full field schema of the two catalogues is given in Table 3.3 and Table 3.4. The

Table 3.2: Example of converting raw test material into logical steps.

(a) Requirement	
ID	REQ-001
Name	`DeactivateDoorLock`
Purpose	Verify that the rear door locks are deactivated when the vehicle is stationary.
Description	When `DeactivateDoorLock` is called from the body control module, the system shall deactivate the rear door locks if `DoorLockSensorStatus = Active` and `VehicleSpeed = 0`.
(b) Excerpt of raw test steps	
Step 1	`[function-call] ActivateDrivingMode()`
Step 2	`[set]` signal path includes `RearLeftDoorLockSts`, value = `Active`
Step 3	`[awaitvaluematch]` signal path includes `RearLeftDoorLockSts`, operator = eq, value = `Active`, timeout = 8 000 ms
Step 4	`[function-call] RequestDoorLockChange(Deactivate)`
Step 5	`[awaitvaluematch]` signal path includes `DoorLockActivationType`, operator = eq, value = `Deactivated`, timeout = 2 000 ms
(c) Logical steps after preprocessing	
Step 1	Activate driving usage mode.
Step 2	Set rear-left door lock status to Active.
Step 3	Verify rear-left door lock status equals Active within 8 000 ms.
Step 4	Request door lock state change to Deactivate.
Step 5	Verify door lock activation type equals Deactivated within 2 000 ms.

example values are fictitious but reflect the structure of the original entries. These schemas are directly relevant to test generation: the addressing fields (the bus, configuration and frame fields for `DBSignal`, and the namespace path for `SysVar`) were retained because Phase 2 used them to expand a bare signal name into the fully qualified object path expected by vTESTstudio during VTT rendering (Section 3.2.2), while the value field constrained the values that a generated step could assign or verify.

Table 3.3: Field schema of the `DBSignal` catalogue. The example values are fictitious.

Field	Description	Example
`bus_type_id`	Numeric identifier of the bus system the signal belongs to.	`11`
`node_or_frame_name`	Network node that sends or receives the signal.	`BodyEthBb1`
`network_alias`	Alias of the communication network.	`BodyEthBb1`
`cfg_name`	Signal-database configuration that defines the signal.	`BodyDbCfg`
`frame_name`	Frame that carries the signal.	`BodyPdu07`
`frame_param`	Numeric frame parameter used with the frame name.	`0`
`frame_flag`	Boolean flag qualifying the frame entry.	`True`
`signal_name`	Unique signal identifier; the key used for grounding and lookup.	`RearLDoorLockSts`
`value`	Admissible values: an enumeration of named states, or a numeric range with factor, offset and unit.	`0:Unlk|1:Lk`
`Expand`	Readable description; embedded for retrieval.	rear-left door lock status...

Table 3.4: Field schema of the `SysVar` catalogue. The example values are fictitious.

Field	Description	Example
`var_name`	Name of the system variable; the key used for grounding and lookup.	`rearDoorLockState`
`namespace_path`	Hierarchical namespace locating the variable on the test platform.	`HilSystem`
`value`	Admissible value or values of the variable.	`0`
`Expand`	Readable description; embedded for retrieval.	rear door lock state...

Many signal names followed abbreviated industrial naming conventions, so readable descriptions were added where needed. Existing descriptions were kept when available. Otherwise, the abbreviated names were expanded using the information already present in the catalogue. This was done purely to aid retrieval and readability; the original signal names and their semantics were left unchanged.

3.1.4 Function Library

The test repository contained a set of engineer-defined reusable helper functions for recurring preparation steps and vehicle function calls. These functions recurred across many historical test cases and were already embedded in the existing vTESTstudio scripts. They were therefore collected and organised into a function library to ensure consistent usage across both preprocessing and generation.
The library distinguished two kinds of functions. Test-table functions were defined within vTESTstudio and referenced by name; they made up the large majority of the library. CAPL-based functions, such as the environment `startup` and `teardown` routines, were a small fixed group used for test setup and clean-up. Each function record stored the function name and the signals that the function read or wrote, while the full procedural definitions were kept in a separate definition file. Depending on the function, some calls included parameters while others did not.
Both the stored function definitions and the executable scripts produced later in Phase 2 were expressed using a fixed vocabulary of step types, listed in Table 3.5. Named functions (`ttfunction` and `caplfunction`) were referenced only by name, whereas parameterised steps such as signal assignments, value checks and waits carried their parameters inline. These executable step types are more fine-grained than the five logical-step types in Table 3.1: a logical `verify` step, for example, is realised as an `awaitvaluematch` step, and a logical `call` step as a `ttfunction` or `caplfunction` reference. Fixing this vocabulary kept generated scripts well-formed and made them straightforward to validate before VTT rendering.

Table 3.5: Executable step types and their required fields.

Step type	Description	Fields
`ttfunction`	Call to a reusable test-table function, referenced by name.	`name`
`caplfunction`	Call to a CAPL-based function, such as a `startup` or `teardown` routine.	`name`
`set`	Assign a value to a signal or variable.	`dbobject`, `value_type`, `value`
`awaitvaluematch`	Wait until a signal reaches an expected value within a timeout.	`dbobject`, `operator`, `valuetable_entry`, `timeout_const`, `timeout_unit`
`wait`	Pause execution for a fixed duration.	`const`, `unit`
`for`	Repeat a nested block of steps over a counter range.	`loopvar`, `loopvartype`, `startvalue`, `stopvalue`, `increment`, `steps`
`comment`	Non-executable annotation describing test intent.	`title`
`report`	Write a message to the test report.	`text`

Many function names followed abbreviated technical naming conventions, similar to the signal names. Existing function summaries were reused when available. For functions without an explicit summary, short readable descriptions were added based

on the function name, its parameters when present, and its related signals. The resulting lookup table was used during preprocessing to attach readable descriptions to function calls, and later during generation to support candidate function selection. Table 3.2 illustrates how function calls appeared in practice: steps 1 and 4 show two typical helper-function invocations and the corresponding logical-step descriptions produced after preprocessing.

3.1.5 Training and Retrieval Formats

The preprocessing pipeline produced two categories of data. For training, each sample paired a requirement text with its corresponding logical-step sequence, formatted as instruction-response pairs for supervised fine-tuning. For retrieval, the processed data was embedded into a set of vector indices so that the same material could be searched at generation time.
Four vector indices were constructed, summarised in Table 3.6. They served two distinct retrieval purposes. The *historical-case index* supported Phase 1: each historical requirement–testcase case was represented by three separate vectors, computed from its purpose text, its requirement description, and its signal-related text, so that case similarity could be scored field-by-field (Section 3.2.1). The three *grounding indices*—one for `DBSignal` signals, one for `SysVar` variables, and one for helper functions—supported Phase 2: each entry was represented by a single vector computed from its readable description, so that an individual logical step could be matched against valid signal or function candidates (Section 3.2.2). The historical-case index and the three grounding indices were kept separate because they were queried at different pipeline stages and with different query units—a whole requirement in Phase 1, and a single logical step in Phase 2. All indices were built with the embedding model selected for retrieval, which is compared and fixed in the evaluation (Section 3.5).

Table 3.6: Vector indices constructed for retrieval.

Index	Stage	Indexed unit	Vectors per entry
Historical-case	Phase 1	Requirement–testcase case	3 (purpose, description, signals)
`DBSignal`	Phase 2	Database signal	1 (description)
`SysVar`	Phase 2	System variable	1 (description)
Function	Phase 2	Helper function	1 (description)

Worked examples of the resulting data formats—a training instruction-response pair, a historical-case index entry, and a grounding-index entry—are provided in Appendix A.2.

3.2 Pipeline Design

The overall workflow is shown in Figure 3.1. The generation system was organised as a two-phase pipeline. The first phase, referred to as Req2Logic, converted an

input requirement into a sequence of logical steps. The second phase, Logic2VTT, took these logical steps and produced a vTESTstudio-compatible file by resolving signal and function references and rendering the final XML structure.

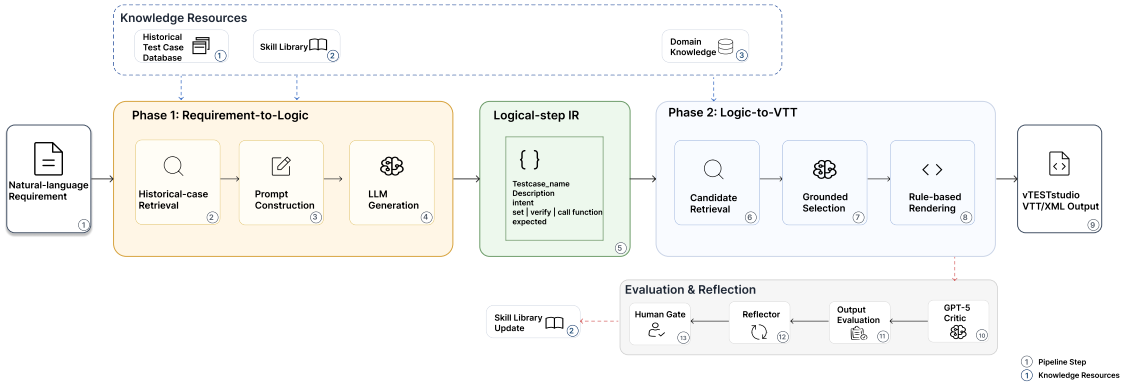


Figure 3.1: Overview of the requirement-to-VTT generation pipeline.

As illustrated in Figure 3.1, the pipeline starts from a natural-language requirement (Step 1). Phase 1 (Steps 2–4) transforms the requirement into a structured logical-step intermediate representation, which is exposed as Step 5. The format of this intermediate layer has been defined during data preprocessing and is described in Section 3.1.2; here, it primarily serves as an inspectable abstraction of test intent before grounding.

Phase 2 (Steps 6–8) takes this intermediate representation and produces a grounded vTESTstudio VTT/XML file (Step 9). Steps 10–13 form an evaluation and reflection loop, including automated critics and human validation.

Three types of resources are used throughout the pipeline: a historical test-case database (Resource 1), a skill library (Resource 2), and domain knowledge for signal and function grounding (Resource 3). These resources are invoked at different stages of the pipeline, as detailed below.

The separation reflects the two kinds of decisions involved in the task. Interpreting a requirement is mainly about understanding the intended test behaviour. Producing a VTT file requires selecting valid industrial signals and functions and following the structure expected by vTESTstudio. Generating the complete XML file in one step would mix these decisions together and make errors harder to trace. Inserting the logical steps between the two phases makes the test logic inspectable before it is grounded to concrete test artefacts.

3.2.1 Phase 1: Requirement-to-Logical-Steps Generation

Phase 1 corresponds to Steps 2–4 in Figure 3.1. It takes a single natural-language requirement as input and produces a structured logical-step representation (Step 5), which serves as an intermediate, inspectable abstraction of test intent before grounding. This phase interacts with the historical test-case database (Resource 1) and the skill library (Resource 2).

Step 1: Historical-case retrieval Before generation, the pipeline retrieved historical cases that were similar to the input requirement. Let $\mathcal{H}$ denote the set of indexed historical cases, and let $h \in \mathcal{H}$ denote one historical-case record. The raw requirement text q_t, a natural-language string, was encoded once with the selected embedding model $E(\cdot)$, producing the query vector $\mathbf{q}_v = E(q_t)$. The query was not split into signal, purpose or description fields. Instead, those distinctions were stored on the historical-case side. Each indexed case had three precomputed vectors: `VectorPurpose`, `VectorDescription`, and `VectorSignals`. The purpose field and signal-related text were prepared offline during data processing, while the description field came from the original requirement text. The `VectorSignals` vector was computed from the signal-related text for dense semantic matching; raw signal-name tokens were stored separately for lexical matching.
In the retrieval equations below, bold mathematical symbols denote dense embedding vectors. Non-bold symbols denote natural-language strings, indexed records, token documents, functions, scalar scores or scalar weights.
During retrieval, the query vector was compared with the stored purpose, description and signal representations of each historical case. The fields were scored separately because cases can be similar for different reasons: they may share signal names, express the same functional purpose, or use similar natural-language phrasing. For a query text q_t and a historical case h, the retrieval score was:

$$S(q_t, h) = w_p \cdot \cos(\mathbf{q}_v, \mathbf{v}_p^h) + w_d \cdot \cos(\mathbf{q}_v, \mathbf{v}_d^h) + w_s \cdot S_{\text{sig}}(q_t, h). \tag{3.1}$$

In Equation 3.1, $S(q_t, h)$ is a scalar retrieval score. The symbols $\mathbf{q}_v$, $\mathbf{v}_p^h$ and $\mathbf{v}_d^h$ are vectors in the embedding space: $\mathbf{q}_v$ is the query vector, and $\mathbf{v}_p^h$ and $\mathbf{v}_d^h$ are the stored purpose and description vectors for historical case h. The symbol h denotes the case record itself. The parameters w_p, w_d, and w_s are scalar field weights, and $S_{\text{sig}}(q_t, h)$ is the signal-field score:

$$S_{\text{sig}}(q_t, h) = \alpha \cdot \widetilde{\text{BM25}}(q_t, N_h) + (1 - \alpha) \cdot \cos(\mathbf{q}_v, \mathbf{v}_s^h). \tag{3.2}$$

In Equation 3.2, $\alpha \in [0, 1]$ is a scalar interpolation weight, where $\widetilde{\text{BM25}}(q_t, N_h)$ denotes the normalised Best Matching 25 (BM25) score measuring lexical similarity between the query text and the signal-name document, and $\mathbf{v}_s^h$ is the dense vector computed from the signal-related text of historical case h. The symbol N_h denotes the tokenised signal-name document for case h. It was treated as a token sequence or multiset rather than as a mathematical set: if a signal occurred several times in a testcase, its tokens were repeated in the lexical index. Signal names were tokenised using common separators such as underscores, colons, dots and whitespace. This lexical component was included because signal identifiers are often short, abbreviated and domain-specific, making them difficult for dense embeddings to represent reliably. To make the BM25 score comparable with cosine similarity, it was normalised within each query:

$$\widetilde{\text{BM25}}(q_t, N_h) = \frac{\text{BM25}(q_t, N_h)}{\max_{h' \in \mathcal{H}} \text{BM25}(q_t, N_{h'})}. \tag{3.3}$$

In Equation 3.3, $\text{BM25}(q_t, N_h)$ is the unnormalised BM25 score between the query

text q_t and the signal-name document N_h. If the maximum BM25 score for a query was zero, the normalised score was set to zero for all historical cases.
The parameter α controlled the balance between lexical signal matching and dense signal similarity. The field weights w_p, w_d, and w_s controlled the contribution of the purpose, description and signal fields to the final score. Several embedding models and weight settings were compared during system development; the best-performing configuration was then fixed for the remaining experiments. The retrieval comparison is described in Section 3.5, and the final selected weights are reported in Chapter 4.

Step 2: Prompt construction The top-ranked historical cases were then inserted into the prompt as few-shot examples. As shown in Figure 3.1, the prompt was constructed by combining:

- the new requirement,
- the retrieved historical cases from Resource 1,
- the expected output schema, and
- optionally relevant entries from the skill library (Resource 2).

The schema bound the output to the logical-step format established during preprocessing, keeping Phase 1 focused on test-logic generation and deferring signal and function grounding entirely to Phase 2.

Step 3: LLM generation and logical-step IR Given the constructed prompt, the selected LLM (e.g. GPT-4o or a fine-tuned Llama-3.3 model) generated a structured logical-step representation. As indicated in Figure 3.1, this output formed an intermediate representation (Step 5), referred to as the logical-step IR. As described in Section 3.1.2, this representation defines a structured format for test intent and enables inspection, debugging, and feedback before grounding into concrete test artefacts.

3.2.2 Phase 2: Candidate Retrieval, Grounded Selection, and VTT Rendering

Phase 2 received the logical-step sequence produced in Phase 1 and converted it into a vTESTstudio-compatible `.vtt` file through three steps: candidate retrieval, grounded selection, and rule-based rendering. The Grounded-Selection step (the second step of Phase 2, described below) used GPT-4o regardless of whether Phase 1 used GPT-4o or the fine-tuned Llama model, so Phase 2 remained constant across both pipeline configurations.

Step 1: Candidate Retrieval. The first step in phase 2 resolved each logical step from Phase 1 into a short list of signal or function candidates. The step `type` field — set to one of `set`, `verify`, `call`, or `wait` during preprocessing — determined the retrieval path. Wait steps required no grounding and were passed directly to the next step. For `set` and `verify` steps, the intent text was encoded with the BGE-M3 embedder and searched against both the `DBSignal` and `SysVar` vector indices

described in Section 3.1.3. For `call` steps, the same encoded intent was searched against the function-library index (Section 3.1.4).
Each index search used a hybrid score combining a normalised BM25 component for lexical matching with a dense cosine similarity component. For a step intent text s_t and a candidate c from catalogue $\mathcal{C}$, the retrieval score was:

$$R(s_t, c) = \alpha \cdot \widetilde{\mathrm{BM25}}(s_t, N_c) + (1 - \alpha) \cdot \cos(\mathbf{s}_v, \mathbf{c}_v), \tag{3.4}$$

where $\mathbf{s}_v = E(s_t)$ is the embedded intent vector, $\mathbf{c}_v$ is the precomputed description vector for candidate c, N_c is the tokenised description document for c, and α is the scalar interpolation weight. Separate values of α were used for signal retrieval and function retrieval. The BM25 score was normalised within each query following the same convention as Equation 3.3. Including a lexical component was important because signal and function identifiers are abbreviated and domain-specific, making dense embeddings alone insufficient for reliable matching.
For `set` and `verify` steps the result lists from the `DBSignal` and `SysVar` searches were merged and re-ranked by score. A small bonus $\delta = 0.05$ was then added to any candidate whose allowed-values string contained the target value explicitly mentioned in the step intent. The magnitude was chosen to be smaller than typical inter-candidate score gaps observed in development, so the bonus acts as a tie-breaker between candidates of similar retrieval score rather than overriding the underlying ranking:

$$\hat{R}(s_t, c) = R(s_t, c) + \delta \cdot \mathbf{1}[\mathrm{val}(s_t) \in \mathrm{AllowedValues}(c)]\,, \tag{3.5}$$

where $\mathbf{1}[\,\cdot\,]$ denotes the indicator function (equal to 1 if the bracketed condition holds and 0 otherwise), $\mathrm{val}(s_t)$ extracts a target-value token from the intent text (for example, `Active` from "equals Active within 8 000 ms"), and $\mathrm{AllowedValues}(c)$ is the stored allowed-value string for candidate c. This bonus was not applied to function retrieval, where allowed values were not relevant. After re-ranking, the top-k candidates per step — carrying the signal name, source type (`DBSignal` or `SysVar`), formatted allowed values, and retrieval score — were assembled into a structured candidate prompt together with each step's typed intent text.

Step 2: Grounded Selection. The second step in Phase 2 used GPT-4o to select one candidate per step and produce a grounded intermediate JSON script. The prompt provided the test case name, description, the ordered list of logical steps with their typed intents, and the top-k candidates retrieved for each step. The model was instructed to act as an automotive test automation engineer: rather than selecting the highest-ranked candidate mechanically, it was directed to reason across all steps, maintain cross-step signal consistency (for example, the same signal must appear in a `set` step and in the subsequent `verify` step for the same variable), and choose whichever candidate made the most engineering sense for the feature under test. The model was explicitly restricted to selecting from the candidates listed for each step and was not permitted to introduce new signal or function names.
The output was an intermediate JSON object in which each step carried a mandatory `step` field for alignment and type-specific fields drawn from the executable step

vocabulary in Table 3.5. `Call` steps produced a `func` object with a `func_name` field; `set` steps produced an object with `signal_name`, `value_type`, and `value`; `verify` steps produced an `awaitvaluematch` object with `timeout_const`, `operator`, and an expected-value field; and `wait` steps were parsed from the intent text directly. After generation, the output was normalised: steps were aligned to their logical-step positions by step number, wait-step durations were extracted from the intent text when omitted, and the selected candidate names were validated against the retrieved candidate lists with warnings issued for out-of-list selections. Generating an intermediate JSON object rather than VTT XML directly simplified output validation, kept Phase 2 decoupled from the low-level file format, and made cross-step consistency errors easier to locate.

Step 3: Rule-based Rendering. Step 3 was entirely deterministic and involved no language model. Each bare signal name in the grounded selection was first expanded into the fully qualified `dbobject` path string expected by vTESTstudio. For a `DBSignal` entry, the resolution combined the `bus_type_id`, `node_or_frame_name`, `network_alias`, `cfg_name`, `frame_name`, `frame_param`, `frame_flag`, and `signal_name` fields from Table 3.3 into a pipe-delimited identifier string following the `DBSignal` object format required by the test environment. For a `SysVar` entry, the `var_name` and `namespace_path` fields from Table 3.4 were combined into the corresponding `SysVar` object format. Function steps were resolved to either `ttfunction` or `caplfunction` entries retrieved from the function database. If a signal name was not found in either catalogue, resolution raised an error, making grounding failures explicit before any file was written. The resolved steps were wrapped in a `testGroup` and `testCase` structure with a default preparation sequence (`Set_DefaultValue_UsedSignals`) and a `breakonfail` flag.

The fully resolved schema was then converted into a well-formed `.vtt` XML file. The output followed the standard Vector Test Table XML structure. A fixed test fixture was inserted around all test cases, with a `startup` CAPL function call in the preparation section and a `teardown` call in the completion section, matching the environment initialisation and finalisation routines expected by the test bench. Each step was rendered to its corresponding XML element according to its type: signal-assignment steps became `<set>` elements with a `<dbobject>` sink and a `<valuetable_entry>` or `<const>` source; verification steps became `<awaitvaluematch>` elements with nested `<timeout>` and `<compare>` blocks containing the comparison operator and expected value; wait steps became `<wait>` elements with a `<const>` duration and unit; and function calls became `<ttfunction>` or `<caplfunction>` elements referencing the function name. Test-table function definitions were collected from the function library and appended to the `<ttlib>` section of the file so that vTESTstudio could resolve all referenced functions at load time. This step introduced no new decisions about test logic or signal selection; its sole responsibility was to realise the grounded selection as a structurally valid test artefact.

3.3 Skill Memory Mechanism

The pipeline incorporated a Skill Memory Mechanism designed to accumulate structured domain knowledge from generation errors and apply that knowledge to future generations of similar requirements. Rather than discarding the evaluation outcome after each generation attempt, the mechanism captured recurring gaps as compact, reusable rules—called *skills*—and persisted them to a Skill Book backed by a PostgreSQL database. On subsequent runs, relevant skills were retrieved by vector similarity and injected into the prompt, making the Actor aware of patterns that had caused failures previously.

3.3.1 Skill Structure

Each skill was a structured record with six fields: `skill_name` (a short descriptive label), `trigger_condition` (an if-statement-style predicate describing when the skill applies), `rule` (the concrete guidance the Actor should follow), `example_good` and `example_bad` (illustrative contrasting cases), and `feature_scope` (either a normalised feature identifier or `null` for cross-feature rules). Skills with `feature_scope` set to the string `"general"` formed a separate frozen tier and were never created or modified at runtime.

3.3.2 Skill Tiers and Retrieval

Two tiers of skills were maintained. General skills had `feature_scope = "general"` and were always loaded in full at the start of every generation. They encoded broad automotive test conventions that applied regardless of feature type. Specific skills were either feature-scoped (tied to the current feature identifier) or cross-feature (`feature_scope = null`), and were retrieved dynamically by cosine similarity between the BGE-M3 embedding of the query and the precomputed skill vectors stored in the database. At each iteration, the top-k skills above a fixed similarity threshold were selected; the retrieved pool was then merged with any session-level or trial skills already active, deduplicated by a composite key of `skill_name` and `rule`.

3.3.3 Actor–Critic–Reflector Loop

The mechanism was implemented as a LangGraph state machine whose compiled graph is shown in Figure 3.2. The state carried all information required across nodes, including the current requirement, the generated artefacts, accumulated skills, and loop control variables. Each execution of the loop corresponded to one generation attempt, whose outcome was evaluated and potentially used to update the Skill Book.

Retrieve Skills At the start of each iteration, the *retrieve_skills* node encoded the query with the shared BGE-M3 embedder and called the skill retrieval function, which queried the PostgreSQL skill store for published skills whose cosine similarity to the query vector exceeded the configured threshold. General skills were loaded

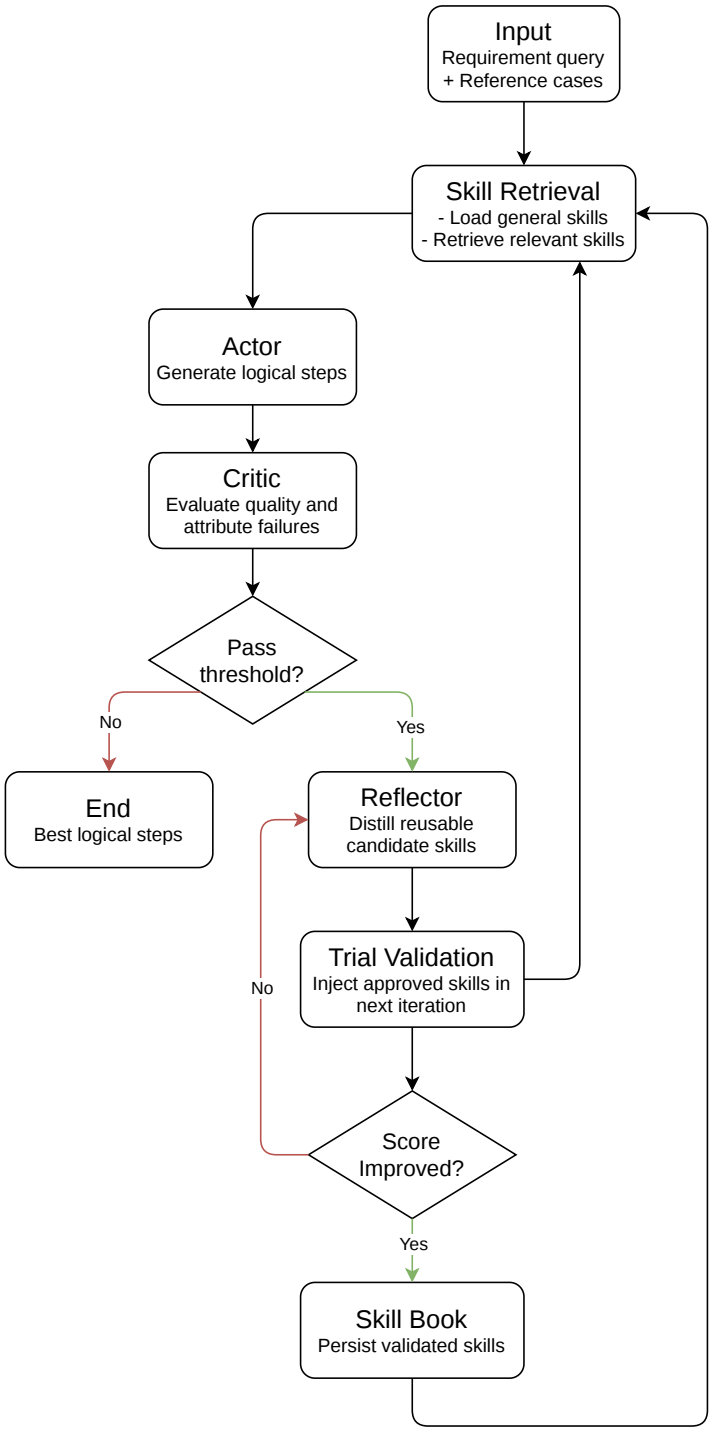


Figure 3.2: LangGraph state-machine diagram of the Actor–Critic–Reflector loop. Dashed edges are conditional transitions; solid edges are unconditional.

separately and kept in a distinct list. The two lists, together with any session or trial skills from the current run, were merged and deduplicated before being passed to the Actor.

Actor The *actor* node called the Phase 1 prompt-builder (Section 3.2.1) with skills injected, producing the logical steps and test-case name for the current iteration. On the first iteration, only the retrieved skills and reference cases were included. On subsequent iterations, the prompt was extended with the Critic's structured feedback from the previous round: a list of identified issues and a numbered must-fix checklist derived from those issues. This forced the Actor to address specific failure patterns rather than regenerating freely.

Critic The *critic* node evaluated the generated logical steps against quality criteria defined formally in Section 3.5. Briefly, the evaluation combined three metrics defined in that section: a logical-adequacy score (S_{adequacy}) assessing structural completeness, domain fit, and rule compliance; a grounding in-domainness score reflecting retrieval quality for signal and function candidates; and an attributed behavioural-coverage score (S_{coverage}) comparing the generated steps against ground-truth test behaviour when a reference test case was available. These were combined into a weighted overall score on a 0–10 scale using configurable weights. The default weighting placed 0.75 on adequacy, 0.25 on coverage, and 0 on grounding in-domainness. The reasoning behind these weights followed the causal alignment between Reflection's intervention and the metrics: Reflection's only intervention point is the Phase 1 prompt seen by the Actor, so S_{adequacy}, which directly measures Phase 1 output quality, is the metric most sensitive to skill injection and receives the largest weight; coverage retains a smaller weight because it captures downstream effects on the assembled artefact that Phase 1 changes can still influence; grounding in-domainness is determined almost entirely by the upstream historical-case retrieval and is not under Reflection's control, so it is set to 0 rather than dropped from the equation, to record that this dimension was considered and deliberately excluded. If trial skills that had been staged in the previous iteration were present in the state and the current score exceeded the pre-trial baseline by at least a minimum delta, those skills were immediately routed to the write node for permanent storage, regardless of whether the pass threshold had been reached. Otherwise, if the overall score reached or exceeded the configurable pass threshold, the loop was terminated with reason *passed_threshold*. If a configured early-stop patience counter was exhausted without improvement, the loop also terminated early. In all remaining cases the Critic's structured output—a score, a list of issues, and a list of actionable suggestions—was forwarded to the Reflector.

Reflector The *reflector* node received the Critic's issues, suggestions, and attribution data, together with the top semantically similar skills already in the Skill Book, and called GPT-4o to distil the failure patterns into structured skill proposals. The Reflector was instructed to classify each proposed skill as *duplicate* (the core test pattern was already covered by an existing skill), *specialisation* (a genuinely distinct sub-scenario not covered by any existing skill), or *new* (no existing skill addressed

the pattern). Duplicate proposals were discarded automatically; only specialisation and new skills were forwarded for human review. The scope of each proposed skill was constrained to either the current feature identifier or `null`; proposing new general skills was explicitly prohibited.

Human Review The *human_review* node used LangGraph's built-in `interrupt` primitive to suspend the graph and present the proposed skills to a human reviewer. The reviewer responded with a list of approved skill indices; all other proposals were discarded. An `auto_approve` configuration flag was provided to bypass the interrupt in batch or testing contexts.

Write Skills Approved skills were not written immediately. Instead, the *write_skills* node placed them into a *trial* buffer and returned control to the beginning of the generation loop: the iteration counter was incremented, skills were re-retrieved (now including the trial skills), and the Actor was invoked again to produce a new generation. After that iteration the Critic re-evaluated the output; if the new score exceeded the pre-trial baseline by at least the minimum delta, the trial skills were promoted to the permanent Skill Book. Promotion was performed by encoding the concatenated skill text (trigger condition, rule, and name) with the shared BGE-M3 embedder and inserting the skill record together with its dense vector into PostgreSQL via `create_skill` followed by `publish_skill`. Promoted skills were simultaneously added to a session-level list, making them immediately available for the rest of the current run without a database round-trip. If the score had not yet reached the pass threshold after promotion, the Reflector was invoked again on the existing generation output to extract further skill proposals, potentially initiating another trial cycle.

Loop Termination The loop was terminated by any of four conditions: the Critic score reaching the pass threshold (*passed_threshold*); a configurable early-stop patience counter expiring after a set number of consecutive iterations without improvement (*early_stop_no_improvement*); the maximum iteration count being reached while trial skills were still staged (*max_iterations_after_trial_stage*); or the global maximum iteration limit being reached (*max_iterations_reached*). The final node recorded the stop reason in the graph state for downstream logging and analysis.

3.4 Model Configurations and Fine-tuning

The model configurations were defined around the two-phase pipeline described above. Phase 1 was the main language-model interpretation stage, where a requirement was converted into logical steps, and this was therefore the point at which model choice was varied. One configuration used GPT-4o through the Azure OpenAI Application Programming Interface (API) (`gpt-4o`) for requirement-to-logical-step generation. The other used the same Phase 1 prompt with a QLoRA-fine-tuned Llama-3.3-70B-Instruct model. After Phase 1, both configurations passed through the same Phase 2 module — signal and function candidate retrieval, grounding,

script assembly and VTT rendering — without variation. Where Phase 2 itself required a language model (e.g. for script assembly), the same GPT-4o instance was used in both conditions. The comparison therefore isolated the requirement interpretation performed in Phase 1.

The open-weight model was fine-tuned to examine whether the requirement-to-logical-steps task could be adapted using the historical test material available inside the project. A proprietary hosted model provided a strong practical baseline, but its training data, internal adaptation procedure and model weights were not available for inspection. Fine-tuning was therefore limited to the open-weight model, where the training data, adapter parameters and deployment environment could be controlled directly. Limiting adaptation to local training also reduced the need to expose internal requirement and test-case records to an external service.

Fine-tuning was applied only to Phase 1, following the same split already present in the pipeline: Phase 1 handled language-model interpretation of requirement intent, whereas Phase 2 was a fixed downstream process built around retrieved signal and function candidates, grounding decisions, script assembly and VTT rendering. Keeping Phase 2 unchanged made the fine-tuning experiment narrower and easier to interpret.

The training set contained 850 scenario-level requirement–logical-step pairs produced by the preprocessing pipeline. Each record was stored as an instruction–response example with three fields: `instruction`, `input`, and `output`. The instruction described the task and the required schema, the input field contained the requirement text, and the output field contained the reference logical-step representation. A strict output-format reminder was appended to the instruction to make the expected output format explicit: raw JavaScript Object Notation (JSON) without markdown fences or explanatory text. The data were split into 679 training samples, 77 validation samples and 94 held-out test samples using stratified sampling by requirement family; no requirement family was allowed to appear in more than one split. Here, a requirement family denotes the group of scenario-level samples derived from the same original requirement identifier.

Training used QLoRA [16]. The base model weights were quantised to 4-bit NormalFloat 4-bit (NF4) with double quantisation and kept static, while LoRA adapter matrices were trained on the attention projection layers. Token statistics were computed with the Llama-3.3 tokeniser. The longest full sequence, including prompt and target output, stayed within the 4 096-token context window, so the maximum sequence length was set to 4 096 to avoid truncating the end-of-sequence token. Table 3.7 summarises the fine-tuning configuration.

During inference, low-randomness decoding was used for the fine-tuned model with a temperature of 0.1 and a limit of 2 048 new tokens. The generated text was passed through the same parsing routine used for the other Phase 1 outputs: the end-of-sequence marker was removed, repeated trailing text was truncated at the last valid JSON boundary, and the remaining JSON was parsed before being passed to Phase 2. Outputs that could not be parsed were recorded as invalid for evaluation, but no repair step was applied before downstream comparison.

Table 3.7: QLoRA configuration for Phase 1 fine-tuning.

Setting	Value
Base model	Llama-3.3-70B-Instruct
Quantisation	4-bit NF4 with double quantisation
LoRA rank r	32
LoRA scaling factor α_{LoRA}	64
LoRA dropout	0.05
Target modules	`q_proj`, `k_proj`, `v_proj`, `o_proj`
Maximum sequence length	4 096 tokens
Training epochs	1
Per-device batch size	1
Gradient accumulation	16 steps
Effective batch size	32 across two GPUs
Learning rate	2×10^{-5}
Learning-rate scheduler	Cosine
Warmup ratio	0.05
Weight decay	0.01
Precision	BF16
Distributed training	DeepSpeed ZeRO-2
Compute	2× NVIDIA A100-80 GB GPUs

3.4.1 Software and Compute Environment

All training and inference jobs were executed on a Microsoft Azure Machine Learning compute cluster equipped with two NVIDIA A100-80 GB Graphics Processing Units (GPUs). The generation pipeline and the fine-tuning experiment relied on separate software stacks, summarised in Table 3.8. The local project configuration in `pyproject.toml` targeted Python 3.11 for static analysis and packaging compatibility, while the project package declared a `requires-python` constraint of Python 3.11 or newer.

The pipeline runtime ran locally under Python 3.13 and communicated with the Azure OpenAI service for GPT-4o inference and hosted embeddings. Code quality was governed by the repository-level configuration: Ruff used a 100-character line length and a Python 3.11 target, while the shared project package declared Python 3.11 or newer as its runtime floor. Cosine similarity was computed directly with NumPy rather than through a dedicated vector-database engine, since the historical case index contained fewer than 100 entries and did not require approximate nearest-neighbour search. The fine-tuning environment on the Azure ML cluster used a separate Python 3.10 runtime; the Parameter-Efficient Fine-Tuning (PEFT) version was confirmed from the `peft_version` field written into `adapter_config.json` at training time, while the PyTorch version was pinned in the cluster environment specification.

Table 3.8: Software versions and package constraints used in the project. The upper group lists the main pipeline components; the middle group lists project-level package and code-quality configuration from `pyproject.toml`; the lower group lists the fine-tuning stack deployed on the Azure Machine Learning (ML) cluster. Versions marked "≥" are lower-bound constraints; the resolved runtime version may be higher.

Component	Version	Role
Generation pipeline		
Python	3.13.5	Local runtime; compatible with Python ≥3.11 project constraint
OpenAI Python SDK	2.21.0	Azure OpenAI API (GPT-4o + embeddings)
sentence-transformers	3.4.1	Local embedding models (BGE-M3, MiniLM, MPNet)
rank-bm25	0.2.2	BM25 lexical scoring for signal retrieval
NumPy	2.4.2	Cosine similarity and vector operations
azure-core	1.38.1	Azure authentication
Project package and code-quality configuration		
Python target	py311 / ≥3.11	Ruff static-analysis target and package runtime floor
Ruff line length	100	Formatting and linting convention
setuptools	≥69	Package build backend
wheel	declared	Build-system wheel support
LangGraph	≥1.1.10	Graph-based pipeline orchestration
SQLAlchemy	≥2.0.0	Database access layer for stored resources
psycopg[binary]	≥3.1.18	Optional PostgreSQL database driver
Fine-tuning (Azure ML cluster)		
Python	3.10	Training runtime
PyTorch	2.4.1	Deep-learning framework
CUDA	12.4	GPU compute (NCCL 2.20.5)
HF Transformers	≥4.40	Model loading and tokenisation
Hugging Face PEFT	0.18.1	QLoRA adapter implementation
bitsandbytes	≥0.43.0	4-bit NF4 quantisation
TRL	≥0.8.6	SFT trainer and data collator
DeepSpeed	≥0.14.0	Distributed training (ZeRO-2)
Accelerate	≥0.29.0	Multi-GPU orchestration

3.5 Evaluation Methodology

The evaluation was organised into four layers:

- **E1** measured historical-case retrieval quality.
- **E2** isolated Phase 1 requirement-to-logical-step generation.
- **E3** evaluated the complete requirement-to-VTT pipeline.
- **E4** analysed the retrieval depth used by the skill memory mechanism.

The retrieval layer tested how reliably similar historical cases could be found. The Phase 1 layer isolated the generation model before signal grounding or VTT rendering was applied, with particular focus on the effect of supervised fine-tuning. The end-to-end layer then examined the behaviour of the complete pipeline after retrieval, generation, grounding and artefact construction had been combined. The skill-depth layer varied the number of retrieved skill entries injected into the Phase 1 prompt in order to study the effect of skill retrieval depth independently from the main system comparison.

The layered design also made error tracing more explicit. A failed final artefact could come from missing or irrelevant retrieved examples, an incorrect Phase 1 interpretation, or incomplete grounding and assembly in Phase 2. Evaluating these stages separately made those failure sources easier to distinguish.

A single exact-match or text-overlap score was not suitable for this task. One requirement can often be tested through several valid action sequences, and historical VTT cases may include wrapper functions or steps belonging to neighbouring requirements. The evaluation therefore matched the metric to the artefact available at each layer. E1 used rank-based retrieval metrics, E2 compared the generated structured JSON, and E3–E4 used functional metrics for logical adequacy, domain grounding and behavioural coverage. Text-overlap metrics were kept only as supporting semantic indicators inside E2, not as stand-alone measures of test correctness.

3.5.1 Evaluation Design and Data-Leakage Prevention

The four evaluation layers separated retrieval, generation, complete artefact construction and skill-memory configuration. This separation made the source of errors easier to identify, and it also limited data leakage. In this thesis, data leakage means that information from the held-out test cases enters fine-tuning, few-shot example selection or retrieved prompt context during evaluation. If this happened, the reported performance could be overly optimistic.

The held-out test partition described in Section 3.4 was therefore kept out of fine-tuning and prompt-example selection. In few-shot conditions, all demonstration examples were drawn from the training partition. When retrieved examples were used, the paired reference case for the evaluated test requirement was removed from that requirement's retrieved context.

3.5.2 E1: Embedding and Historical-Case Retrieval Evaluation

The first evaluation layer, E1, evaluated the historical-case retrieval strategy used before Phase 1 generation. The aim was to test whether a new requirement or query could retrieve its linked historical requirement–testcase pair from the retrieval store. The experiment compared candidate embedding models, single-field retrieval strategies and weighted multi-field fusion across the purpose, description and signal fields described in Section 3.2.1.

3.5.2.1 Retrieval Query Set

The retrieval query set was built from the requirement–testcase pairs in the embedding retrieval store introduced in Section 3.1.1. For each pair, GPT-4o generated three query variants through a structured LLM prompt. The prompt included the requirement identifier, requirement name, purpose, requirement description, testcase description, up to 15 extracted signals with semantic expansions and occurrence counts, and up to 20 logical-step intents from the reference testcase. The model had to return a JSON object with the original requirement identifier and exactly three query objects. If the returned identifier did not match the source requirement identifier, it was corrected automatically. Records were discarded if they could not be parsed as valid JSON or did not contain exactly three query objects. The final query set contained 267 queries, with 89 queries for each type in Table 3.9.

Table 3.9: Synthetic query types used in E1 retrieval evaluation.

Query type	**Simulated user situation**	**Retrieval behaviour tested**
General intent	The user knows the functional goal but not the technical signal names.	Matching high-level functional intent to the stored purpose and description fields.
Signal specific	The user searches with signal names, signal meanings or signal-combination logic.	Recovering cases from technical signal evidence and signal-related descriptions.
Scenario based	The user describes the vehicle state and behavioural situation without naming the requirement.	Matching a behavioural situation to the underlying requirement and testcase.

These query variants were used only for retrieval evaluation. In the implemented pipeline, each incoming requirement was encoded once and compared against the stored purpose, description and signal fields using the retrieval function defined in Section 3.2.1.

3.5.2.2 Embedding Models and Retrieval Strategies

Five embedding configurations were evaluated, as listed in Table 3.10. For each model, the historical retrieval store contained three precomputed vectors per requirement–testcase pair: one for the purpose field, one for the description field, and one for the signal-related field.

Table 3.10: Embedding models evaluated in E1.

Model	Dim.	Type
`text-embedding-3-large`	1536	Hosted (OpenAI)
`text-embedding-3-large`	3072	Hosted (OpenAI)
BGE-M3	1024	Local (sentence-transformer)
all-MiniLM-L6-v2	384	Local (sentence-transformer)
all-mpnet-base-v2	768	Local (sentence-transformer)

Each synthetic query was embedded with the same model configuration as the retrieval store being evaluated. The resulting query vector was then used to retrieve historical cases under four strategies: purpose-only, description-only, signal-only (single-field strategies), and weighted multi-field fusion as defined in Section 3.2.1. The signal field used the hybrid dense-and-BM25 score from the same section, with $\alpha = 0.5$. This value was used as a fixed neutral setting rather than as a tuned hyperparameter: it assigned equal weight to lexical matching over signal identifiers and dense matching over expanded signal descriptions, avoiding an additional tuning dimension on the small historical-case retrieval set. For weighted fusion, model-specific field weights were selected via grid search and then fixed before reporting ablation results.

3.5.2.3 Metrics and Analyses

Retrieval quality was measured with Recall@k and Mean Reciprocal Rank (MRR), which are standard information-retrieval metrics for evaluating whether relevant items are returned and how early they appear in a ranked list [39, 40, 18]. They were appropriate here because the retrieval component returned a ranked list of historical cases and each query had one known relevant target: Recall@k measured whether that target appeared within the top-k retrieved cases, while MRR rewarded systems that ranked the target closer to the first position. For each query there was exactly one target item—the historical requirement–testcase pair sharing the same source requirement identifier. Scores were averaged across the full query set Q, where Q denotes the retrieval queries:

$$\text{Recall@}k = \frac{1}{|Q|} \sum_{q \in Q} \mathbb{I}[\text{rank}_q \leq k] \tag{3.6}$$

$$\text{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{rank}_q} \tag{3.7}$$

where rank_q is the position of the target item in the ranked list ($\text{rank}_q = \infty$ when the target is absent, contributing 0 to both metrics). The indicator $\mathbb{I}[\cdot]$ equals 1 when the condition inside the brackets is true and 0 otherwise. Recall@5, i.e. Recall@k with $k = 5$, was used as the primary reported setting because the Phase 1 prompt consumed five retrieved historical examples.
Three analyses were performed:

1. **Ablation study** Purpose-only, description-only, signal-only and weighted-fusion retrieval were compared for each embedding model.
2. **Weight-sensitivity study** All valid weight triplets satisfying $w_p+w_d+w_s = 1$ (step size 0.1) were enumerated and Recall@5 was visualised over the resulting ternary space.
3. **Query-type analysis** Recall@5 was reported separately for general-intent, signal-specific and scenario-based queries to test whether performance varied with the type of information available in the query.

3.5.3 E2: Phase 1 Model and Fine-Tuning Evaluation

The second evaluation layer, E2, evaluated the raw Phase 1 mapping from a requirement to a structured JSON test representation. Retrieval-based context was excluded from this layer so that changes could be attributed to the generation model and the QLoRA adapter. The retrieval component used in the deployed pipeline was evaluated separately in E1.
No signal grounding or VTT rendering was included in E2. All conditions were tested on the same held-out set of 94 scenario-level samples from the supervised fine-tuning split.
Five conditions were compared:

Table 3.11: Conditions used for Phase 1 model and fine-tuning evaluation.

ID	Model	Prompting strategy
C1	GPT-4o	Zero-shot
C2	GPT-4o	3-shot in-context learning
C3	Llama-3.3-70B-Instruct	Zero-shot, no adapter
C4	Llama-3.3-70B-Instruct	3-shot in-context learning
C5	Llama-3.3-70B-Instruct + QLoRA adapter	Fine-tuned

All conditions received the same task instruction and test input. For C2 and C4, three examples from the training partition were prepended to the prompt as demonstrations. The fine-tuned condition used the QLoRA adapter described in Section 3.4. During inference, low-randomness decoding was used with temperature 0.1 and a maximum of 2 048 new tokens. The generated text was stripped of the end-of-sequence marker, truncated at the last valid JSON-array boundary when repetition occurred, and parsed with a JSON parser.
Because the Phase 1 output is structured JSON, E2 did not treat the prediction as ordinary prose. Bilingual Evaluation Understudy (BLEU) is a standard n-gram

precision metric for machine translation [41], but here it would mix brackets, field names and paraphrased intent text into a single score. E2 therefore used five metric groups: structure, count, type, semantic similarity and domain style. These groups separated format failures from errors in step count, action ordering, intent wording and domain-specific type conventions. Precision, recall and F_1 were used for set-like comparisons because they balance over-generation and under-generation [42].
The structure metrics checked whether the output could be consumed by the downstream pipeline. For n test samples, with $v_i = 1$ when the prediction for sample i could be parsed as JSON and $z_i = 1$ when it also satisfied the required schema, the two rates were

$$R_{\text{parse}} = \frac{1}{n}\sum_{i=1}^{n} v_i, \qquad R_{\text{schema}} = \frac{1}{n}\sum_{i=1}^{n} z_i. \tag{3.8}$$

In Equation 3.8, R_{parse} and R_{schema} are scalar rates in $[0, 1]$, n is the number of evaluated samples ($n = 94$ in E2), and v_i and z_i are binary indicator variables for sample i.
The count metrics checked whether the model generated the expected number of logical steps. Let m_i and $\hat{m}_i$ be the number of reference and predicted steps for sample i. Count match and count ratio were computed as

$$R_{\text{count}} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}[m_i = \hat{m}_i], \qquad S_{\text{count}} = \frac{1}{n}\sum_{i=1}^{n}\left(1 - \frac{|m_i - \hat{m}_i|}{\max(m_i, \hat{m}_i, 1)}\right). \tag{3.9}$$

In Equation 3.9, R_{count} is the scalar exact count-match rate, S_{count} is the scalar mean count-ratio score, and m_i and $\hat{m}_i$ are non-negative integer step counts for the reference and prediction, respectively. The $\max(m_i, \hat{m}_i, 1)$ term prevents division by zero for empty outputs.
The type metrics compared the generated action-type sequence with the reference sequence. For a multiset A_i of reference items and $\hat{A}_i$ of predicted items, where the items were either individual step types or adjacent type bigrams, overlap precision and recall were

$$\begin{aligned} P_i &= \frac{\sum_a \min(c_i(a), \hat{c}_i(a))}{\max(\sum_a \hat{c}_i(a), 1)}, \\ R_i &= \frac{\sum_a \min(c_i(a), \hat{c}_i(a))}{\max(\sum_a c_i(a), 1)}, \\ F_{1,i} &= \begin{cases} \frac{2P_iR_i}{P_i+R_i}, & P_i + R_i > 0, \\ 0, & P_i + R_i = 0. \end{cases} \end{aligned} \tag{3.10}$$

In Equation 3.10, a ranges over the items being compared: either individual step-type labels for type multiset F_1, or adjacent step-type pairs for type bigram F_1. The functions $c_i(a)$ and $\hat{c}_i(a)$ are integer counts of item a in the reference multiset A_i and predicted multiset $\hat{A}_i$. The values P_i, R_i, and $F_{1,i}$ are scalar precision, recall and F_1 scores for sample i; when $P_i + R_i = 0$, $F_{1,i}$ was set to zero.
Type multiset F_1 ignored ordering and measured whether the model produced the right mix of `set`, `verify`, `call` and related actions. Type bigram F_1 applied the same formula to adjacent type pairs, so it rewarded local ordering patterns. A

separate type edit similarity used normalised Levenshtein distance [43] between the reference type sequence τ_i and predicted sequence $\hat{\tau}_i$:

$$S_{\text{edit},i} = 1 - \frac{d_{\text{Lev}}(\tau_i, \hat{\tau}_i)}{\max(|\tau_i|, |\hat{\tau}_i|, 1)}. \tag{3.11}$$

In Equation 3.11, $S_{\text{edit},i}$ is a scalar similarity score for sample i, d_{Lev} is the Levenshtein edit distance, and τ_i and $\hat{\tau}_i$ are the reference and predicted type sequences. The notation $|\cdot|$ denotes sequence length.

The semantic metrics focused on the natural-language `intent` or `text` fields of position-aligned steps. Intent token F_1 used the overlap formula in Equation 3.10 over lower-cased tokens. Recall-Oriented Understudy for Gisting Evaluation (ROUGE)-L F_1 was computed from the longest common subsequence between a reference intent token sequence u and a predicted sequence $\hat{u}$ [44], using

$$P_{\text{LCS}} = \frac{\text{LCS}(u, \hat{u})}{|\hat{u}|}, \qquad R_{\text{LCS}} = \frac{\text{LCS}(u, \hat{u})}{|u|}, \qquad F_{\text{ROUGE-L}} = \frac{2P_{\text{LCS}}R_{\text{LCS}}}{P_{\text{LCS}} + R_{\text{LCS}}}. \tag{3.12}$$

In Equation 3.12, u and $\hat{u}$ are token sequences from the reference and predicted intent text, $\text{LCS}(u, \hat{u})$ is the length of their longest common subsequence, and $|u|$ and $|\hat{u}|$ are token counts. P_{LCS}, R_{LCS}, and $F_{\text{ROUGE-L}}$ are scalar precision, recall and F_1 values.

ROUGE-L was included because longest-common-subsequence matching is less sensitive to small local word-order differences than exact matching [44]. BERTScore F_1 was also reported because contextual embedding similarity can recognise semantically similar paraphrases even when they share few surface tokens [45]. For a predicted intent token embedding set $\hat{X}$ and reference set X,

$$\begin{aligned} P_{\text{BERT}} &= \frac{1}{|\hat{X}|} \sum_{\hat{\mathbf{x}} \in \hat{X}} \max_{\mathbf{x} \in X} \cos(\hat{\mathbf{x}}, \mathbf{x}), \\ R_{\text{BERT}} &= \frac{1}{|X|} \sum_{\mathbf{x} \in X} \max_{\hat{\mathbf{x}} \in \hat{X}} \cos(\mathbf{x}, \hat{\mathbf{x}}), \\ F_{\text{BERT}} &= \frac{2P_{\text{BERT}}R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}. \end{aligned} \tag{3.13}$$

In Equation 3.13, X and $\hat{X}$ are sets of contextual token-embedding vectors for the reference and predicted intent text, respectively. Each $\mathbf{x} \in X$ and $\hat{\mathbf{x}} \in \hat{X}$ is a vector, $\cos(\cdot, \cdot)$ returns scalar cosine similarity, and P_{BERT}, R_{BERT}, and F_{BERT} are scalar BERTScore precision, recall and F_1 values.

Finally, domain style accuracy checked whether parseable outputs followed the step-type naming convention used in the reference annotations. This metric was included because downstream schema handling assumes stable domain labels; a semantically plausible label such as `check` instead of `Verify` may still require repair before grounding. Invalid or unparsable outputs received zero for content metrics that required parsed step lists.

3.5.4 E3: Retrieval Grounding and Skill Injection Ablation

The third evaluation layer, E3, evaluated the complete requirement-to-VTT workflow under a 2×2 ablation that crossed RAG-based grounding with skill injection. It was kept separate from E1 and E2 because it measured the implemented pipeline end to end rather than retrieval quality or Phase 1 model adaptation in isolation. Phase 2 grounding, schema assembly and VTT rendering were fixed across all four conditions, so that observed differences could be attributed solely to the two upstream factors under test: whether retrieval grounding candidates were supplied and whether retrieved skills were injected into the Phase 1 prompt. Each evaluated sample passed through the full pipeline and produced three intermediate artefacts, evaluated in the three stages summarised in Table 3.12.

Table 3.12: Evaluation stages in E3 and their primary metrics.

Stage	Artefact evaluated	What is measured	Primary metric
1	Generated logical steps	Adequacy of the Phase 1 test plan	S_{adequacy}
2	Candidate-grounding prompt	Whether grounding candidates are drawn from an in-domain region	Avg. top-k similarity, top-1 similarity, margin
3	Generated VTT schema	Behavioural coverage against the reference VTT case	S_{coverage}

Because the adequacy and coverage metrics rely partly on an LLM judge, they should be read as judge-assessed evaluation indicators rather than direct measurements of ground-truth correctness. The judge made it possible to compare generated test logic with requirement intent and reference artefacts when exact matching was too brittle, but no separate human validation study was conducted for these scores. The composite results built from them therefore inherit this limitation.

3.5.4.1 Logical Adequacy

Logical adequacy measured the quality of the generated Phase 1 test plan before VTT rendering. The score combined a deterministic rule component with an LLM-based judge component.

The rule score S_{rule} started at 1.0 and was reduced by a fixed penalty for each violation detected, as listed in Table 3.13. The penalty magnitudes encode a severity ranking rather than calibrated failure probabilities: two fatal violations bring S_{rule} down to 0.4, which marks the output as substantially deficient, while several minor violations still leave the score in a usable range. The rule score therefore behaves as a near-categorical filter at the fatal end and as a graded signal at the minor end.

Table 3.13: Rule violations and their penalties for the deterministic rule score.

Violation	Severity	Penalty
`Set` step intent does not match `Write <signal> to <value>`	Fatal	−0.30
`Set` step has no subsequent `Verify` for the same signal	Fatal	−0.30
No `Verify` step present in the generated output	Minor	−0.10
Fewer than two logical steps generated	Minor	−0.10

The LLM-based judge then assigned two independent scores on a 1–10 scale, each normalised to $[0, 1]$, as described in Table 3.14.

Table 3.14: LLM judge components for logical adequacy.

Component	Criterion
$S_{\text{structural}}$	Step-level completeness and ordering relative to the requirement.
S_{domain}	Feature-specific behaviour, signal usage and safety-relevant constraints.

The final logical-adequacy score was computed as

$$S_{\text{adequacy}} = 0.30 \cdot S_{\text{rule}} + 0.30 \cdot S_{\text{structural}} + 0.40 \cdot S_{\text{domain}}. \tag{3.14}$$

The weights were treated as rubric weights for a composite indicator rather than learned parameters; they make the evaluation priorities explicit and are intended to be interpreted together with the sensitivity analysis. Domain fit received the largest weight because the main failure mode of an otherwise well-formed plan is selecting implausible automotive behaviour, signals or constraints. Rule compliance and structural completeness are necessary preconditions, but on their own they do not establish domain correctness. Sensitivity to the choice of weights is reported in Section 4.4 (Figure 4.7). If no logical steps were generated, all components were set to zero.

3.5.4.2 Grounding In-Domainness

Grounding in-domainness measured whether the generated logical steps were mapped to relevant signal and function candidates. During Logic2VTT grounding, the candidate prompt listed retrieved candidates for each step together with their similarity scores, which were parsed after generation to compute three per-step values. For step i with k retrieved candidates scored $s_{i,1} \geq \cdots \geq s_{i,k}$:

$$\overline{s}_i = \frac{1}{k} \sum_{j=1}^{k} s_{i,j}, \qquad s_i^{(1)} = s_{i,1}, \qquad \Delta_i = s_{i,1} - \frac{1}{k-1} \sum_{j=2}^{k} s_{i,j}. \tag{3.15}$$

In Equation 3.15, $\overline{s}_i$ is the mean candidate similarity for step i, $s_i^{(1)}$ is the top candidate similarity, and Δ_i is the margin between the top candidate and the mean of the remaining candidates.
Query-level scores were obtained by averaging $\overline{s}_i$, $s_i^{(1)}$ and Δ_i across all N generated steps. If no candidate prompt was available, all scores were set to zero. This stage measured the quality of the intermediate grounding evidence used by Phase 2, not behavioural correctness directly.

3.5.4.3 Behavioural Coverage

Behavioural coverage compared the generated artefact with the relevant parts of the reference VTT case. Because a reference test case could contain steps for neighbouring requirements, the evaluation first identified the reference steps and signals relevant to the current requirement or query. Non-functional reference steps, such as comments, reporting steps, waits and loop constructs, were excluded from the functional step comparison.
Step coverage measured whether the generated schema covered the relevant functional actions and checks from the reference case. A judge first selected the reference steps relevant to the current requirement or query, then compared them with the generated set, allowing function calls to cover equivalent lower-level set or verify actions. The output was a score in $[0, 1]$ and a list of uncovered reference behaviours.
Signal coverage measured whether the generated schema used the relevant ground-truth signals. The matching pipeline proceeded in four steps:

1. **Relevance filtering.** An LLM judge filtered the full ground-truth signal set down to signals relevant to the current requirement or query.
2. **Exact matching.** Generated signals were extracted from the schema and matched against the relevant ground-truth set by exact string comparison.
3. **Semantic matching.** Unresolved ground-truth signals were passed to a semantic matching judge (DeepSeek-V3), which resolved naming-convention differences such as `rear_left` vs. `rear_driver`.
4. **Fallback.** If the semantic matcher was unavailable, the evaluator fell back to exact string matching only.

For detailed query formulations, an additional query-signal coverage diagnostic was computed by extracting signal-like terms from the query and checking whether they appeared in the generated signal set.
The combined behavioural-coverage score was computed as

$$S_{\text{coverage}} = 0.30 \cdot S_{\text{step}} + 0.70 \cdot S_{\text{signal}}. \tag{3.16}$$

Signal coverage received the larger weight because step coverage is affected by an abstraction-level mismatch: a vTESTstudio `call` step encapsulates equivalent lower-level `set` and `verify` operations, so a generated artefact and a reference artefact that express the same behaviour at different abstraction levels can disagree on step coverage even though they are functionally identical. Signal coverage is not affected by this mismatch and was therefore used as the primary measure of behavioural fidelity, with step coverage kept at a lower weight as a complement. If no generated schema or no reference pair was available, behavioural coverage was skipped and the coverage score was set to zero for that sample.

3.5.4.4 Three-Stage Score

The three stage scores were combined into a single composite metric, the Three-Stage Score (TSS), so that all four conditions could be compared on one score. The TSS used a two-level structure. First, a hard gate rejected any output whose rule compliance was below the maximum value. Rule compliance is treated differently from the other quality dimensions because a fatal rule violation is a different kind of failure: it makes the plan structurally unusable regardless of how favourable the coverage or grounding scores appear. Combining rule compliance into the weighted sum as just another component would let downstream metrics offset a fundamentally broken plan, so the hard gate treats it as a hard prerequisite rather than another weighted component. For outputs that passed the gate, a weighted sum of the three stage representatives was computed:

$$\text{TSS} = \begin{cases} 0.5 \cdot Q_1 + 0.3 \cdot Q_3 + 0.2 \cdot Q_2 & \text{if } S_{\text{rule}} = 1.0, \\ 0 & \text{otherwise,} \end{cases} \tag{3.17}$$

where $Q_1 = (S_{\text{structural}} + S_{\text{domain}})/2$ is the Stage 1 quality score with the rule component removed to avoid double-counting, Q_2 is the top-1 grounding similarity from Stage 2, and $Q_3 = S_{\text{coverage}}$ is the behavioural coverage score from Stage 3. The weight ordering reflects how directly each stage measures the usability of the final deliverable. Stage 1 (Q_1) is taken as the primary signal because a logically incoherent test plan cannot be repaired by downstream grounding or rendering. Stage 3 (Q_3) receives the next-largest weight because it measures behavioural alignment with the reference artefact, the closest available stand-in for ground-truth correctness. Stage 2 (Q_2) receives the smallest weight because grounding similarity is an intermediate retrieval-quality signal whose effect on the final artefact is mediated by Stages 1 and 3 rather than directly observable. A binary pass criterion, Pass@τ, was defined as $\mathbb{I}[\text{TSS} \geq \tau]$ with the recommended threshold $\tau = 0.50$. The weights and threshold are rubric choices rather than learned parameters; a sensitivity analysis in Section 4.4 (Figure 4.8, panel (f)) checks that the condition ranking is preserved under alternative weight configurations.

3.5.4.5 Data Partition

The requirement–testcase pairs were partitioned at the pair level rather than at the query level. This avoided placing the summary and detailed query formulations for the same requirement–testcase pair in different splits, and was constructed to keep related query formulations together while preserving the main feature-scope distribution as far as possible. Table 3.15 shows the resulting partition; it is used by both E3 and E4, although the two experiments draw from different portions of the non-induction analysis set.

3.5.4.6 Experimental Conditions

The E3 experiment used a 2×2 factorial design crossing RAG-based grounding (active or inactive) with skill injection (active or inactive), as listed in Table 3.16. Phase 2 grounding and VTT assembly were kept identical across all conditions so

Table 3.15: Requirement–testcase-pair split used in E3 and E4.

Partition	Pairs	Role
Induction	50	Skill acquisition only
Validation	15	Non-induction analysis set
Test	24	Non-induction analysis set
Total	89	

that observed differences could be attributed to these two upstream factors. Skill depth was fixed at $k = 1$ in all skill-enabled conditions. The samples evaluated in E3 corresponded to the 24 requirement–testcase pairs in the *test* portion of the analysis split (Table 3.15), each evaluated under both a summary and a detailed query formulation, for 48 samples per condition; paired statistical comparisons were performed on the 47 samples with complete outputs across all four conditions.

Table 3.16: Experimental conditions in E3.

Label	Description	RAG	Skill (k)
Baseline	No grounding candidates, no skill context	×	×
+Skill	Phase 1 prompt includes top-1 skill	×	✓ ($k = 1$)
+RAG	Retrieval candidates provided; no skill	✓	×
+RAG+Skill	Both grounding and skill injection active	✓	✓ ($k = 1$)

The results of E3 are reported in Section 4.4.

3.5.5 E4: Skill Retrieval-Depth Sensitivity Analysis

The fourth evaluation layer, E4, examined how the number of retrieved skills injected into the Phase 1 prompt affected pipeline output quality. It was designed as a configuration study rather than as a replacement for the full end-to-end comparison. Phase 2 grounding and VTT construction were kept fixed across all conditions so that observed differences could be attributed to the upstream skill injection.

E4 used the same requirement–testcase split as E3 (Table 3.15), but drew from the full non-induction analysis set: validation (15) and test (24), for 39 requirement–testcase pairs in total. Each skill-enabled condition was paired with a no-skill baseline on these same 39 pairs, so all reported values measured paired changes relative to the same input. All 39 pairs had a summary query formulation; 36 additionally had a detailed formulation, which accounts for the per-query-type sample sizes reported in the results. The four experimental conditions are listed in Table 3.17.

Each condition produced stage 1, stage 2 and stage 3 evaluation scores for every sample. The metrics reported for each sample are listed by source stage in Table 3.18. Each requirement was evaluated under two query formulations—summary and detailed—and results were aggregated to requirement level before statistical testing.

Table 3.17: Experimental conditions in E4.

Condition	Skills injected into Phase 1 prompt	k
No skill (baseline)	None	—
With skill, $k = 1$	Top-1 retrieved skill	1
With skill, $k = 3$	Top-3 retrieved skills	3
With skill, $k = 5$	Top-5 retrieved skills	5

Table 3.18: Metrics tracked in E4, grouped by evaluation stage.

Stage	Metric	Description
1	Logical adequacy (S_{adequacy})	Composite rule + structural + domain score.
1	Rule compliance	Deterministic rule-violation penalty score.
1	Structural quality	LLM structural score (step completeness and ordering).
1	Domain fit	LLM domain score (feature-specific behaviour and signals).
2	Avg. top-k similarity	Mean candidate similarity across all generated steps.
2	Top-1 similarity	Similarity of the highest-ranked grounding candidate.
2	Retrieval margin	Top-1 similarity minus mean of remaining candidates.
3	Behavioural coverage (S_{coverage})	Composite step + signal coverage score.
3	Step coverage	Fraction of relevant reference steps covered.
3	Signal coverage	Fraction of relevant ground-truth signals covered.
—	Generated step count	Number of logical steps produced by Phase 1.

Because each with-skill condition was paired with the no-skill baseline for the same requirement, statistical testing used paired comparisons. Significance was assessed with the Wilcoxon signed-rank test [46] (two-sided). The paired effect size was reported as Cohen's d_z,

$$d_z = \frac{\bar{\delta}}{s_\delta}, \tag{3.18}$$

where $\bar{\delta}$ is the mean paired difference and s_δ its standard deviation across requirement pairs. Mean deltas were reported together with 95% bootstrap confidence intervals computed from 5 000 bootstrap resamples of the requirement-level delta values. Three analyses were performed:

1. **Top-k dose response.** The three primary metrics (logical adequacy, in-domain retrieval, behavioural coverage) were plotted as a function of $k \in \{0, 1, 3, 5\}$ to identify whether gains scaled with retrieval depth.

2. **Metric decomposition.** The sub-components of logical adequacy (rule compliance, structural quality, domain fit) and of behavioural coverage (step coverage, signal coverage) were examined separately to identify which aspects of quality were most affected by skill injection.
3. **Query-form breakdown.** Results were split by query formulation (summary vs. detailed) to test whether skill injection benefited one query type more than the other.

The results of E4 are reported in Section 4.5.

4
Results

This chapter presents the evaluation results using the four-layer structure defined in Section 3.5. It first compares the candidate embedding models for historical-case retrieval, then examines the isolated Phase 1 model and fine-tuning results. Section 4.4 then reports the end-to-end pipeline comparison, a 2×2 ablation over retrieval grounding and skill injection, evaluated up to the logical test steps and a coverage comparison of the generated VTT artefacts against reference cases. The final section analyses how retrieval depth affects the skill-injection setting.

4.1 Evaluation Setup

The four result sections use different experimental units because they test different parts of the system. The retrieval experiment uses synthetic query variants over historical requirement–testcase pairs. The Phase 1 model comparison uses the held-out scenario-level fine-tuning test split. The skill-depth sensitivity analysis uses the requirement–testcase-level non-induction analysis set defined in Section 3.5.5. For that reason, each result section states the sample unit used for its metrics.
Table 4.1 summarises the results reported in this chapter and the sections in which they appear.

Table 4.1: Overview of result sections.

Layer	Focus	Reported metrics
E1	Embedding and historical-case retrieval strategy	Recall@5, MRR, Recall@5 by query type
E2	Phase 1 model and fine-tuning comparison	JSON/schema validity, count ratio, type F_1, BERTScore, ROUGE-L, domain style
E3	Retrieval grounding and skill injection ablation (2×2)	Three-stage metrics (S_{adequacy}, top-1 grounding similarity, S_{coverage}); composite TSS and Pass@τ
E4	Skill retrieval-depth sensitivity	Three-stage metrics (S_{adequacy}, top-1 grounding similarity, S_{coverage}); generated step count; paired change (Wilcoxon, Cohen's d_z)

4.2 E1: Embedding Model Comparison

This section gives the E1 retrieval results for the five embedding configurations defined in Section 3.5. It covers single-field retrieval, weighted multi-field fusion, field-weight sensitivity and query-type performance.

4.2.1 Experimental Conditions

The E1 conditions combined two factors, both defined in Section 3.5. The first was the embedding model: the five configurations listed in Table 3.10, spanning hosted OpenAI models and local sentence-transformer models. The second was the retrieval strategy: three single-field strategies (purpose-only, description-only and signal-only) and a weighted multi-field fusion, applied uniformly to every model. For weighted fusion, model-specific field weights were selected by grid search over all valid weight triplets and then fixed before reporting. Results are reported per model and per strategy on the synthetic query set.

4.2.2 Single-Field Retrieval and Weighted Fusion

Table 4.2 shows Recall@5 and MRR for each embedding model under four retrieval strategies. Weighted fusion gave the highest Recall@5 for every model. OpenAI-3072d and BGE-M3 reached the best overall Recall@5, both scoring 0.944 under weighted fusion.

Table 4.2: Ablation study results for single-field and weighted-fusion retrieval.

Model	Metric	Purpose	Description	Signals	Weighted fusion
OpenAI-1536d	Recall@5	0.854	0.824	0.509	**0.933**
OpenAI-1536d	MRR	0.709	0.684	0.352	**0.788**
OpenAI-3072d	Recall@5	0.861	0.824	0.517	**0.944**
OpenAI-3072d	MRR	0.722	0.691	0.352	**0.794**
BGE-M3	Recall@5	0.918	0.869	0.502	**0.944**
BGE-M3	MRR	0.762	0.719	0.350	**0.779**
MiniLM	Recall@5	0.843	0.783	0.494	**0.880**
MiniLM	MRR	0.678	0.620	0.345	**0.688**
MPNet	Recall@5	0.873	0.734	0.487	**0.910**
MPNet	MRR	0.689	0.564	0.338	**0.723**

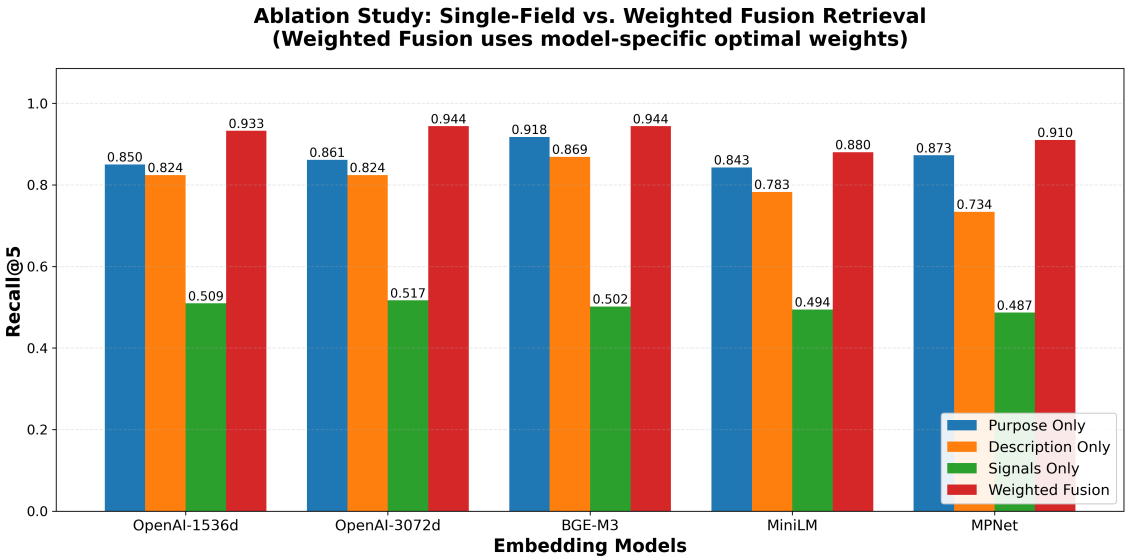


Figure 4.1: Recall@5 for single-field and weighted-fusion retrieval across all embedding models.

The purpose field was the strongest single field for every model, with Recall@5 ranging from 0.843 for MiniLM to 0.918 for BGE-M3. The signal field was consistently weakest, with Recall@5 between 0.487 and 0.517. Weighted fusion improved every model beyond its best single-field score.

4.2.3 Weight Sensitivity

The weight-sensitivity analysis was run separately for each embedding model. For each model, field-weight triplets satisfying $w_p + w_d + w_s = 1$ were explored at a step size of 0.1. The best settings are shown in Table 4.3.

Table 4.3: Best field-weight configuration for each embedding model.

Model	w_p	w_d	w_s	**Recall@5**
OpenAI-1536d	0.5	0.4	0.1	0.933
OpenAI-3072d	0.5	0.4	0.1	0.944
BGE-M3	0.5	0.4	0.1	0.944
MiniLM	0.4	0.5	0.3	0.880
MPNet	0.6	0.3	0.1	0.910

The two OpenAI configurations and BGE-M3 shared the same best weighting: $w_p = 0.5, w_d = 0.4, w_s = 0.1$. MiniLM needed more weight on description and signals, whereas MPNet used the highest purpose weight. The best Recall@5 values ranged from 0.880 for MiniLM to 0.944 for OpenAI-3072d and BGE-M3.

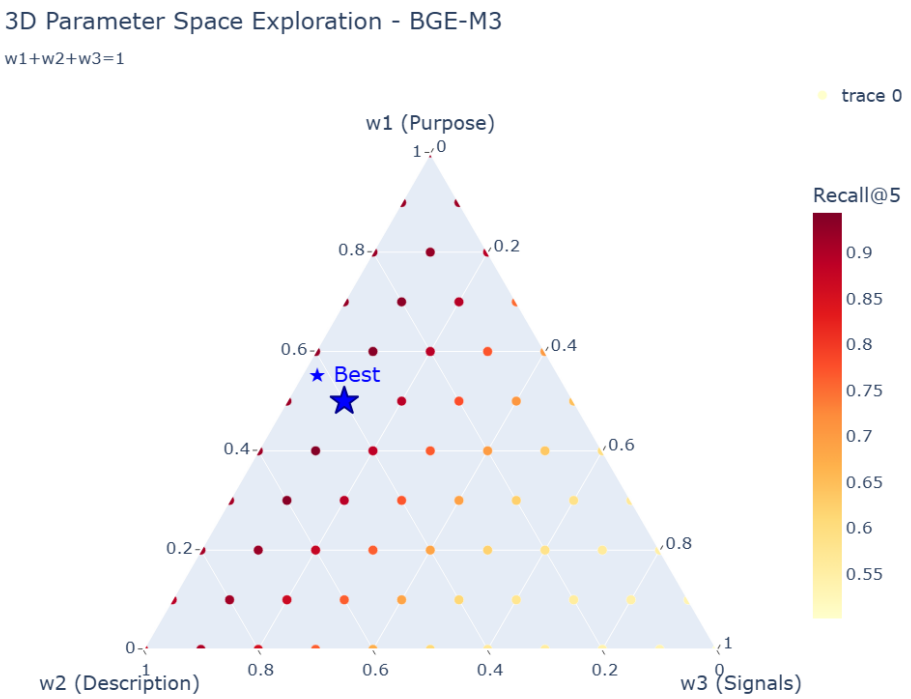


Figure 4.2: Ternary plot of Recall@5 as a function of field weights for BGE-M3. The star marks the best-performing configuration.

For BGE-M3, Recall@5 ranged from 0.502 at the signal-only corner to 0.944 at $w_p = 0.5, w_d = 0.4, w_s = 0.1$. The BGE-M3 ternary surface is shown in Figure 4.2.

4.2.4 Query-Type Breakdown

Table 4.4 breaks down Recall@5 by query type using the optimised weights for each model. MPNet achieved the highest general-intent score (0.978), BGE-M3 achieved the highest signal-specific score (0.978), and OpenAI-3072d and BGE-M3 tied for the highest scenario-based score (0.921). MiniLM had the lowest scenario-based score (0.787).

Table 4.4: Recall@5 by query type under model-specific optimised weights.

Model	General intent	Signal specific	Scenario based
OpenAI-1536d	0.966	0.933	0.899
OpenAI-3072d	0.966	0.944	0.921
BGE-M3	0.933	**0.978**	0.921
MiniLM	0.921	0.933	0.787
MPNet	**0.978**	0.888	0.865

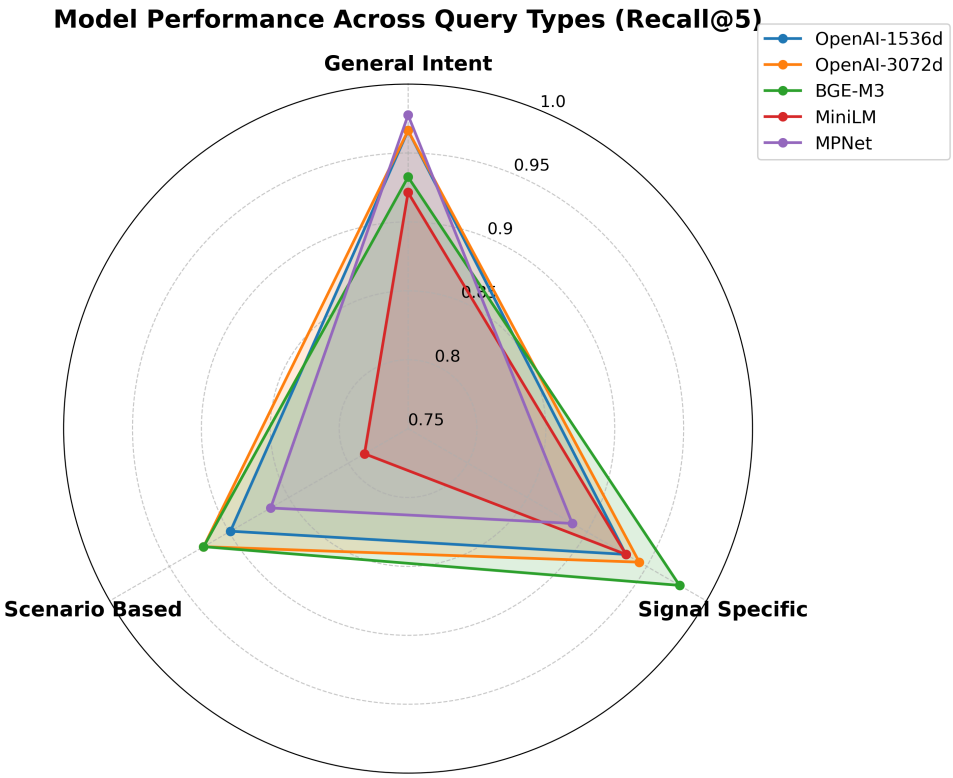


Figure 4.3: Radar chart of Recall@5 by query type for each embedding model under optimised weights.

No single model was best in all three query categories. OpenAI-3072d was the most balanced, with Recall@5 values of 0.966, 0.944 and 0.921. BGE-M3 had the highest signal-specific score and tied for the highest scenario-based score, while MPNet had the highest general-intent score.

4.2.5 Discussion

Weighted fusion improved retrieval for every embedding model, even though the signal field was weak when used alone. This suggests that signal evidence was useful as a complement to purpose and description information, but not reliable enough as a standalone retrieval field. A likely reason is that signal identifiers are short, abbreviated and often domain-specific, which makes nearest-neighbour matching less reliable without purpose or description context.

The best weight settings showed a similar pattern. The two OpenAI configurations and BGE-M3 placed most of the weight on purpose and description, while keeping a small signal contribution. For BGE-M3, the best score was not an isolated point: nearby configurations around $w_p = 0.5$, $w_d = 0.4$ and $w_s = 0.1$ stayed above 0.90 Recall@5. This made the selected weighting less sensitive to small tuning changes.

The query-type breakdown showed different strengths across models. MPNet performed best on general-intent queries but was weaker on signal-specific queries, which is consistent with a general-purpose paraphrase embedding model. BGE-M3 showed the opposite profile, with the strongest signal-specific performance.

OpenAI-3072d gave the most even performance across query types, but required 3072-dimensional vectors.
BGE-M3 was selected as the default embedding model for the remaining experiments. It matched the highest overall Recall@5 (0.944, tied with OpenAI-3072d), used lower-dimensional vectors (1024 vs. 3072), and could be deployed locally without an external service dependency. The selected retrieval setting used weighted fusion with $w_p = 0.5, w_d = 0.4, w_s = 0.1$.

4.3 E2: Phase 1 Model and Fine-Tuning Evaluation

This section presents the isolated Phase 1 model comparison, which evaluated the raw mapping from a requirement to a structured JSON test representation.

4.3.1 Experimental Conditions

The experiment used the five conditions defined in Table 3.11: GPT-4o zero-shot, GPT-4o 3-shot, Llama-3.3-70B-Instruct zero-shot, Llama-3.3-70B-Instruct 3-shot, and the QLoRA-adapted Llama-3.3-70B-Instruct model. Retrieval and Phase 2 grounding were not included in this experiment, so observed differences could be attributed to the generation model and the QLoRA adapter.

4.3.2 Automatic Metrics

Table 4.5 gives the E2 metrics for the held-out test set of 94 scenario-level samples. The metrics follow the structure, count, type, semantic and style layers defined in Section 3.5. To keep the table compact, the columns use condition IDs: C1 is GPT-4o zero-shot, C2 is GPT-4o 3-shot, C3 is Llama-3.3-70B-Instruct zero-shot, C4 is Llama-3.3-70B-Instruct 3-shot, and C5 is the QLoRA-adapted Llama-3.3-70B-Instruct model.

4.3.3 Fine-Tuning Comparison

Figure 4.4 compares the effect of C5 (the QLoRA-adapted Llama-3.3-70B-Instruct model) against the base model C3 (Llama-3.3-70B-Instruct zero-shot). These two conditions used the same base model and differed only by the QLoRA adapter, so the comparison isolated the effect of supervised fine-tuning from differences between hosted and local model families.
Fine-tuning improves most structural and semantic metrics, with more pronounced gains in type-level structure than in semantic similarity. Both C3 and C5 achieve perfect JSON parse rate and schema validity, indicating that fine-tuning mainly affects structural quality rather than format compliance.
The largest improvements are observed in type-level structure. Type multiset F_1 increases from 0.38 to 0.45, and type bigram F_1 from 0.15 to 0.23. Type edit

Table 4.5: Phase 1 model and fine-tuning results on the held-out test set ($n = 94$). Best result per metric is shown in bold.

Metric	C1	C2	C3	C4	C5
Structure					
JSON parse rate (%)	**100.00**	**100.00**	**100.00**	92.55	**100.00**
Schema valid rate (%)	92.55	85.11	**100.00**	92.55	**100.00**
Count					
Count match (%)	11.70	**18.09**	14.89	11.70	11.70
Count ratio	**0.6727**	0.6497	0.6638	0.6318	0.6561
Type					
Type multiset F_1	0.3810	0.3026	0.3872	0.3899	**0.4453**
Type bigram F_1	0.0755	0.0801	0.1537	0.1842	**0.2287**
Type edit similarity	0.2700	0.2446	0.3094	0.3128	**0.3640**
Semantic					
Intent token F_1	0.0932	**0.0984**	0.0876	0.0980	0.0950
BERTScore F_1	0.8727	**0.8736**	0.8723	0.8082	0.8735
ROUGE-L F_1	0.0772	0.0816	0.0729	**0.0849**	0.0808
Style					
Domain style accuracy (%)	40.43	9.57	82.98	87.36	**88.30**

similarity improves from 0.31 to 0.36, and domain style accuracy increases from 82.98% to 88.30%.

In contrast, semantic metrics show only modest gains, with smaller increases in intent token F_1, BERTScore F_1, and ROUGE-L F_1. Count prediction does not improve, as count match decreases while count rate slightly increases. The fine-tuned model therefore learned the local step-type patterns and naming convention better than the base model, but the number of steps per scenario remained a weakness.

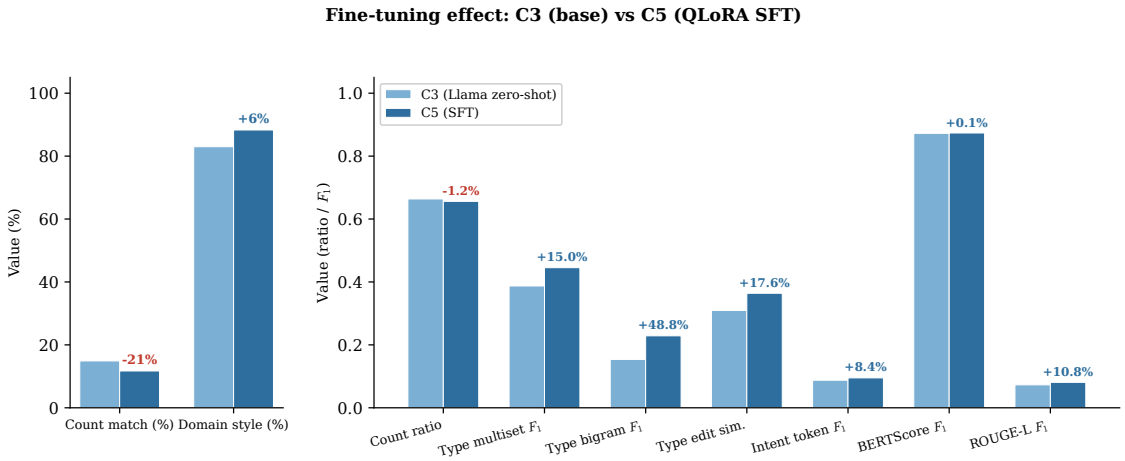


Figure 4.4: Effect of QLoRA fine-tuning on E2 metrics. Light bars show C3 (Llama-3.3-70B-Instruct zero-shot); dark bars show C5 (fine-tuned). Annotations give the relative change from C3 to C5. Metrics are defined in Section 3.5.

4.3.4 Comparison with GPT-4o

The hosted GPT-4o settings, C1 (zero-shot) and C2 (3-shot), serve as reference baselines for the isolated Phase 1 task. Figure 4.5 provides a layer-level view of E2

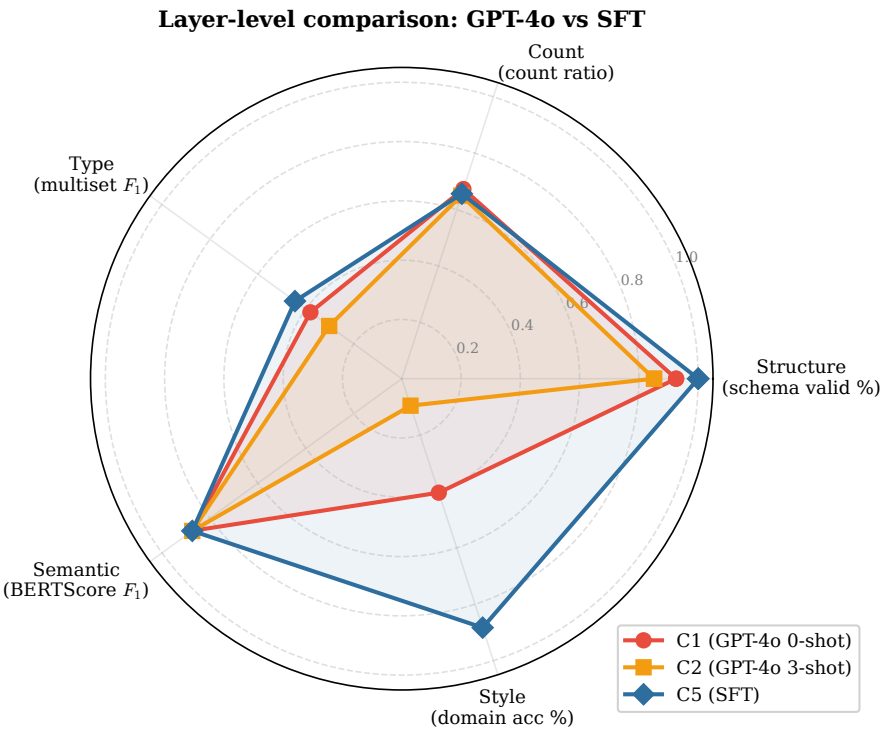


Figure 4.5: Layer-level comparison of C1 (GPT-4o zero-shot), C2 (GPT-4o 3-shot), and C5 (QLoRA fine-tuned) on the E2 evaluation. Each axis represents one evaluation layer using a single representative metric, normalised to $[0, 1]$.

performance for C1, C2, and the fine-tuned model C5.
C5 leads on the format- and convention-related layers. It reaches 100% schema validity and achieves markedly higher domain-style accuracy (88.30%) than either GPT-4o baseline. C5 also performs best on the Type layer, consistent with improved action-type organisation and more reliable local step patterns.
In contrast, the Semantic layer shows only minor differences: GPT-4o and C5 are essentially tied (e.g., BERTScore F_1 differs by only 0.0001 between C2 and C5). Count-related behaviour does not show a consistent improvement, indicating that step-count prediction remains a limitation. Overall, QLoRA fine-tuning mainly strengthens structured output conventions and type-level organisation, while semantic similarity and length-related behaviour change little.

4.3.5 Discussion

The Phase 1 results suggest that fine-tuning mainly improved the structure and local conventions of the generated representation. The clearest gains for the QLoRA-adapted model were in schema validity, action-type matching, local type ordering and domain style, rather than in intent-text similarity alone. Phase 2 requires stable step types and parseable fields before it can ground signals and render a VTT artefact, so these structural gains are the ones that directly help downstream processing.
The semantic metrics were less discriminating. BERTScore F_1 and ROUGE-L F_1

were close for several parseable conditions even when the type-level scores differed more clearly. Two outputs can use similar intent wording while still differing in the action sequence or in the domain-specific labels that matter for downstream processing. For that reason, the structured metrics were more useful than text similarity alone for judging Phase 1 output quality.
The main remaining weakness was step-count prediction. Fine-tuning improved the way steps were typed and styled, but it did not improve the number of steps generated for each scenario. Choosing an appropriate decomposition of a requirement into logical actions is therefore a separate problem from producing valid JSON or following the local annotation style. In the full pipeline, a well-formed Phase 1 output can still omit actions or introduce extra ones before grounding begins.
Compared with GPT-4o, the fine-tuned local model was stronger on the format and domain-convention metrics, while GPT-4o remained competitive on count and semantic similarity. The comparison therefore supports the use of the fine-tuned model when local output conventions and downstream compatibility are the main concern, while also showing that fine-tuning did not remove all planning errors from Phase 1.

4.4 E3: Retrieval Grounding and Skill Injection Ablation

This section reports the E3 results for the 2×2 ablation over RAG-based grounding and skill injection. The four conditions were evaluated on the same 48 requirement–query pairs, with paired statistical comparisons performed on the 47 samples having complete outputs across all four conditions. All conditions used the same Phase 2 grounding and VTT assembly logic; observed differences were therefore attributable to whether retrieval-augmented grounding and skill injection were active. Skill depth was fixed at $k = 1$ in all skill-enabled conditions to ensure a fair comparison with the no-skill baseline. The four experimental conditions are defined in Table 3.16 in Section 3.5.

4.4.1 Stage-Level Results

Table 4.6 reports the mean score for each metric across the three evaluation stages.

4.4.2 Adequacy Weight Sensitivity

Because the logical-adequacy score (Equation 3.14) combines three components under fixed weights, its conclusions could in principle depend on the particular weighting chosen. A post-hoc sensitivity analysis was therefore performed: the per-sample component scores S_{rule}, $S_{\text{structural}}$ and S_{domain} stored for every E3 sample were re-aggregated under seven alternative weight schemes, without re-running any model generation. The schemes covered an equal-weight baseline, the reported weighting $(0.30, 0.30, 0.40)$, two perturbations that shifted weight towards domain or structural quality, the more domain-heavy preset used internally by the Reflection critic

Table 4.6: E3 results across all four conditions. Best result per metric is shown in bold. Significance markers are relative to Baseline using the paired Wilcoxon signed-rank test ($^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$).

Metric	Baseline	+Skill	+RAG	+RAG+Skill
Stage 1 — Logical adequacy				
S_{adequacy}	0.501	0.600***	0.552**	**0.653*****
Rule compliance	0.431	0.670***	0.664**	**0.845*****
Structural quality	0.560	0.619	0.557	**0.643**
Domain fit	0.508	0.534	0.464	**0.519**
Stage 2 — Grounding in-domainness				
Top-1 similarity	0.798	0.814	**0.885**	0.877
Stage 3 — Behavioural coverage				
S_{coverage}	0.595	0.595	0.627	**0.635**
Step coverage	0.354	0.421	0.384	**0.462**
Signal coverage	0.698	0.670	**0.732**	0.640
Composite				
TSS (mean)	0.155	0.383***	0.359**	**0.507*****
Pass@τ=0.50 (%)	20.8	53.2	46.8	**70.2**

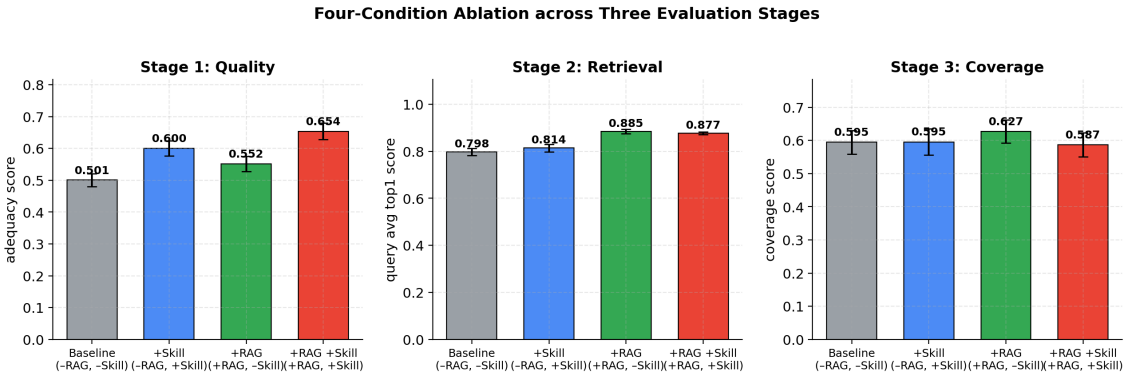


Figure 4.6: Mean score for the primary metric at each evaluation stage across the four E3 conditions. Error bars show ±1 standard error.

(0.10, 0.20, 0.70), and two degenerate single-component weightings included as sanity references.

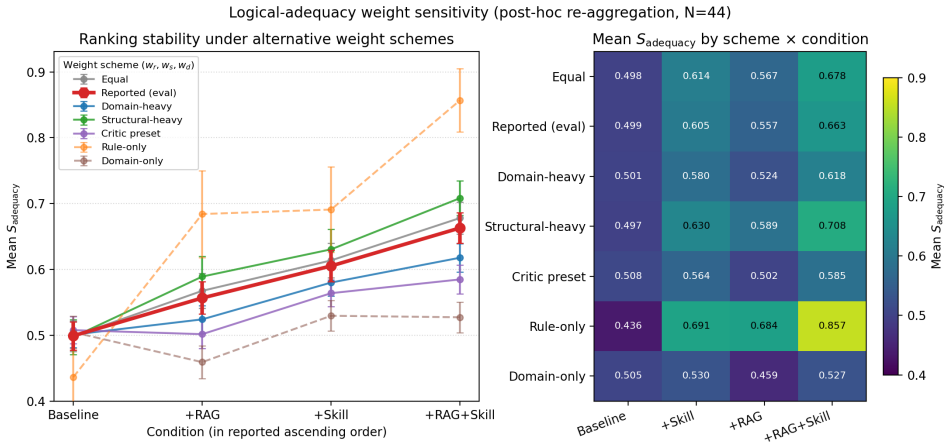


Figure 4.7: Post-hoc re-aggregation of S_{adequacy} under seven alternative weight schemes ($N = 44$ samples shared across all four E3 conditions). Left: mean adequacy by condition, with conditions ordered along the abscissa by the reported-scheme ranking; a monotonically increasing line therefore indicates that the scheme preserves that ranking. Solid lines denote non-degenerate schemes; dashed lines denote degenerate single-component weightings. The reported scheme (0.30, 0.30, 0.40) is shown in bold red. Error bars are ±1 standard error of the mean. Right: heatmap of the same means for quick numeric inspection.

The ranking Baseline < +RAG < +Skill < +RAG+Skill obtained under the reported weighting was preserved by all four non-degenerate moderate schemes (Equal, Reported, Domain-heavy and Structural-heavy). Sample-level Spearman rank correlation between the reported S_{adequacy} and the re-aggregated score exceeded 0.88 for every non-degenerate scheme and every condition, indicating that the relative ordering of individual outputs was also stable. The only contrast affected by the choice of weights was Baseline versus +RAG, whose sign flipped under the heavily domain-dominated Critic preset; in that scheme the two conditions remained within 0.6 percentage points of each other, so the flip reflects a near-tie rather than a substantive reversal. The four qualitative conclusions drawn from the main E3 table—that +RAG+Skill is the strongest condition, that skill injection improves over Baseline, that skill injection alone exceeds RAG alone, and that adding RAG on top of skill improves the score further—are therefore stable across the weight settings tested.

4.4.3 Three-Stage Score

The TSS combined all three evaluation stages into a single composite score. Its distribution across conditions is illustrated in Figure 4.8, which shows the violin distribution, per-stage decomposition, Pass@τ robustness curve, empirical Cumulative Distribution Function (CDF), per-sample paired comparison and sensitivity to the chosen stage weights.

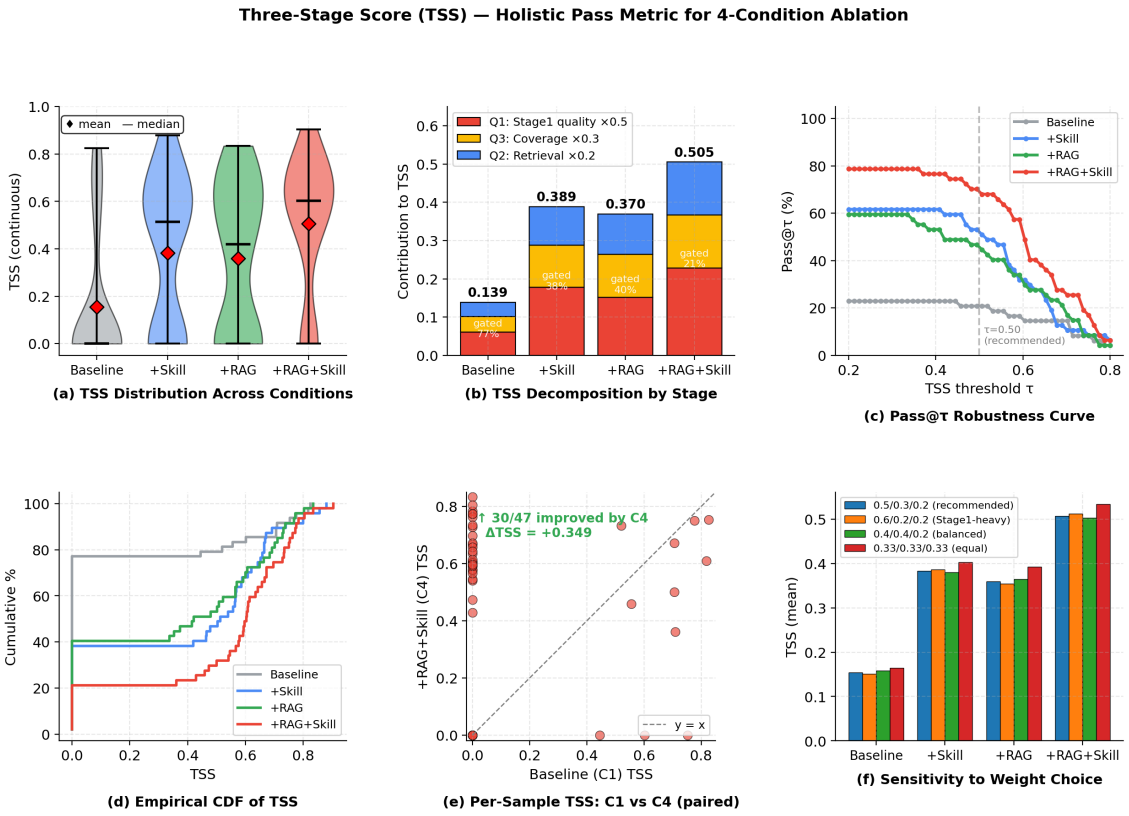


Figure 4.8: TSS analysis for the four E3 conditions. (a) Violin distribution of per-sample TSS (red diamond = mean, black bar = median). (b) Mean TSS decomposed by stage contribution; the percentage in each bar shows the fraction of samples gated to zero by a failing rule check. (c) Pass@τ rate as τ varies from 0.2 to 0.8; the dashed line marks the recommended threshold $\tau = 0.50$. (d) Empirical CDF of TSS. (e) Per-Sample comparison of Baseline against +RAG+Skill (paired); green annotation shows the fraction of samples that improved. (f) Sensitivity of the mean TSS to four alternative stage-weight configurations.

Paired Wilcoxon signed-rank tests on TSS showed that both Skill alone and RAG alone produced significant improvements over Baseline (+Skill: $\Delta\text{TSS} = +0.240$, $p = 0.0006$; +RAG: $\Delta\text{TSS} = +0.201$, $p = 0.0027$). The full combination was the strongest condition overall ($\Delta\text{TSS} = +0.349$, $p < 0.001$). Adding Skill on top of RAG alone produced a further significant gain ($\Delta\text{TSS} = +0.148$, $p = 0.019$), while adding RAG on top of Skill alone did not reach significance ($\Delta\text{TSS} = +0.127$, $p = 0.062$). Panel (f) of Figure 4.8 shows that the ranking of the four conditions was stable across all four alternative weight configurations tested.

The empirical CDF in panel (d) shows a pronounced vertical step at TSS = 0 for every condition, followed by flat segments between subsequent jumps. The step at zero is a direct consequence of the rule-gating mechanism in the scoring rubric: any sample failing the rule check is collapsed to TSS = 0, which concentrates a substantial probability mass on this single value, with the step height matching the per-condition gated fractions reported in panel (b). The flat segments between jumps additionally reflect that TSS is a weighted sum of finite-valued stage scores,

so several intermediate values are simply unattained at the present sample size.

4.4.4 Stage-Level Decomposition and Query-Type Breakdown

Figure 4.9 shows the contribution of each stage to TSS and the pass rate split by query type.

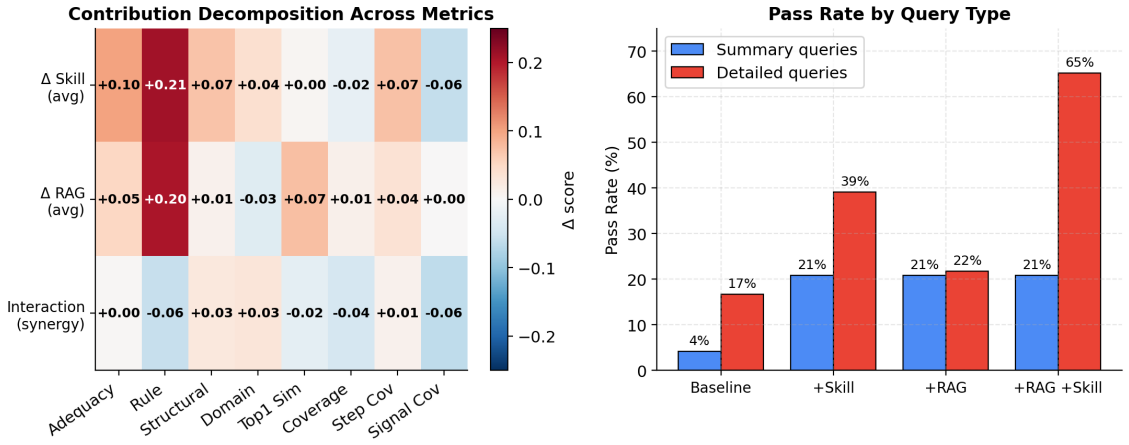


Figure 4.9: Left: mean TSS decomposed into Stage 1 quality (Q_1, weighted 0.5), Stage 3 coverage (Q_3, weighted 0.3) and Stage 2 retrieval (Q_2, weighted 0.2). The label inside each bar shows the proportion of samples gated to zero by a failing rule check. Right: pass rate (%) split by query type (summary and detailed queries).

The decomposition shows that Stage 1 quality (Q_1) accounted for the largest share of the TSS gain in the skill-enabled conditions, reflecting the improvement in rule compliance and structural quality that skill injection produced. The fraction of samples gated to zero by a failed rule check fell from 77% at Baseline to 21% in the +RAG+Skill condition. Stage 2 retrieval quality (Q_2) improved mainly with RAG, consistent with the signal-candidate pool being enlarged by the retrieval step.

The query-type breakdown revealed an asymmetry: detailed queries benefited substantially more from the combined condition (65% pass rate) than summary queries (21%), which were largely unaffected by RAG. Detailed queries provide richer temporal and signal constraints, and these constraints become easier to handle when both skill guidance and retrieval candidates are available.

4.4.5 Coverage Trade-Off

Figure 4.10 places the four conditions in the step-coverage–signal-coverage plane.

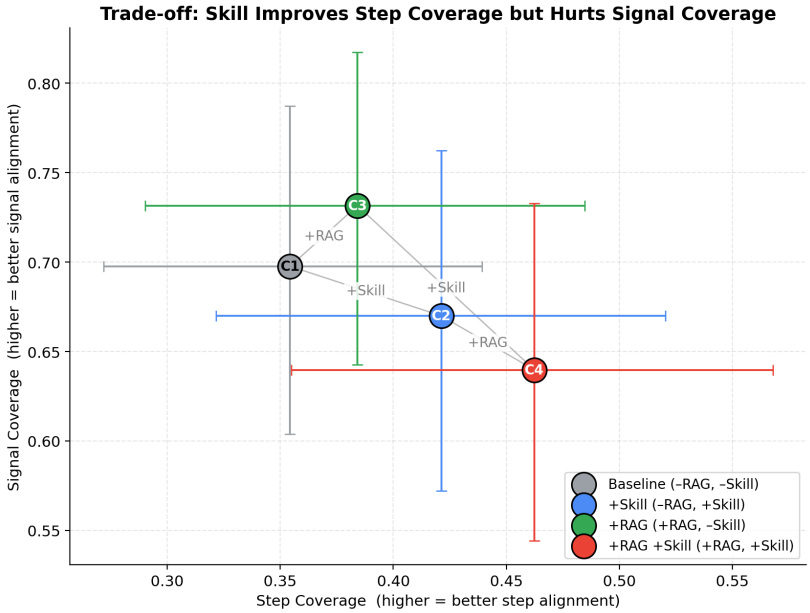


Figure 4.10: Mean step coverage and signal coverage for each condition. Error bars are 95% bootstrap confidence intervals (5000 resamples). Arrows indicate the direction of change when adding Skill (+Skill) or RAG (+RAG) to each condition.

The two coverage dimensions responded differently to the two components. RAG grounding improved signal coverage consistently (+RAG: +0.034 over Baseline), which is expected because the retrieval step supplies more domain-relevant signal candidates. Skill injection increased step coverage in both the no-RAG (+0.067) and the RAG (+0.078) setting, indicating that skill context helps the model plan more complete action sequences. Signal coverage, however, decreased when Skill was added alongside RAG (−0.092 from +RAG to +RAG+Skill), suggesting that skill context shifts the model's attention towards step completeness at the cost of signal specificity. The full combination is therefore a practical trade-off rather than an improvement on both coverage dimensions at once.

4.4.6 Discussion

The E3 results show that Skill injection and RAG grounding each improved overall pipeline quality but addressed different aspects of the generated output. Skill injection drove the largest gains in logical adequacy and rule compliance, reducing the proportion of outputs with fatal planning errors from 77% to 38% in the no-RAG setting. RAG grounding improved grounding in-domainness (top-1 similarity from 0.798 to 0.885) and signal coverage, consistent with a larger and more relevant candidate pool being available during Phase 2.
The combined condition was the strongest overall, but the marginal gain from adding RAG on top of Skill was not statistically significant ($p = 0.062$), whereas adding Skill on top of RAG was ($p = 0.019$). The asymmetry indicates that Skill is the more impactful component. It addressed the main failure mode, fatal rule violations, whether or not retrieval candidates were available, while the extra signal-quality contribution of RAG mattered less once the plan was already well structured.

The query-type breakdown is consistent with this interpretation. Summary queries, which provide fewer explicit constraints, showed little response to RAG augmentation, whereas detailed queries with richer signal and timing constraints benefited markedly from the full combination. The remaining limitation of the pipeline therefore lies in requirements with complex temporal constraints or multiple interacting signals: in those cases, neither skill guidance nor retrieval candidates alone is enough.

4.5 E4: Skill Retrieval Depth Sensitivity Analysis

This section reports the fourth evaluation layer, a skill retrieval-depth sensitivity analysis that complements the end-to-end comparison in Section 4.4. It shows how the evaluated metrics changed when the number of retrieved skills injected into the prompt was varied within the skill-enabled setting.

All results use the non-induction analysis set ($n = 39$ pairs; see Table 3.15). Each pair was evaluated under summary and detailed query formulations; query-level scores were aggregated to requirement–testcase level before the paired comparisons were computed. Three detailed-query samples were excluded from the query-type breakdown due to empty stage-3 output or a judge parse failure (RT_019, RT_045, RT_069), reducing the effective detailed-query sample to $n = 36$ for that analysis.

4.5.1 Experimental Conditions

The E4 conditions varied the skill retrieval depth k, as listed in Table 3.17: a no-skill baseline and three skill-enabled depths injecting the top-1, top-3 and top-5 retrieved skills into the Phase 1 prompt. Each skill-enabled condition was paired with the no-skill baseline on the same requirement–testcase pairs, and Phase 2 grounding and VTT construction were kept fixed across all conditions so that observed differences could be attributed to skill injection depth.

4.5.2 Top-k Dose Response

The analysis compared three skill-injection depths: top-$k = 1$, top-$k = 3$, and top-$k = 5$. Each condition was compared against the corresponding no-skill output at the requirement-testcase level ($n = 39$). Positive values in Table 4.7 therefore indicate an improvement relative to the no-skill setting. In Figure 4.11, each line shows the mean paired change for one metric relative to the matched no-skill output: logical adequacy, in-domain retrieval and coverage. The horizontal zero line marks no change from the no-skill baseline. The shaded bands show bootstrap 95% confidence intervals for the mean paired change.

Table 4.7: Requirement-level change from no-skill generation under different skill retrieval depths. Changes are reported in percentage points except for generated step count.

Skill depth	Logical adequacy	Coverage	Step coverage	Signal coverage	Steps
top-$k = 1$	+2.69	+2.51	+7.50	+0.37	+1.44
top-$k = 3$	+3.96	+1.31	+4.05	+0.13	+1.85
top-$k = 5$	+6.15	-0.52	+5.49	-3.10	+2.06

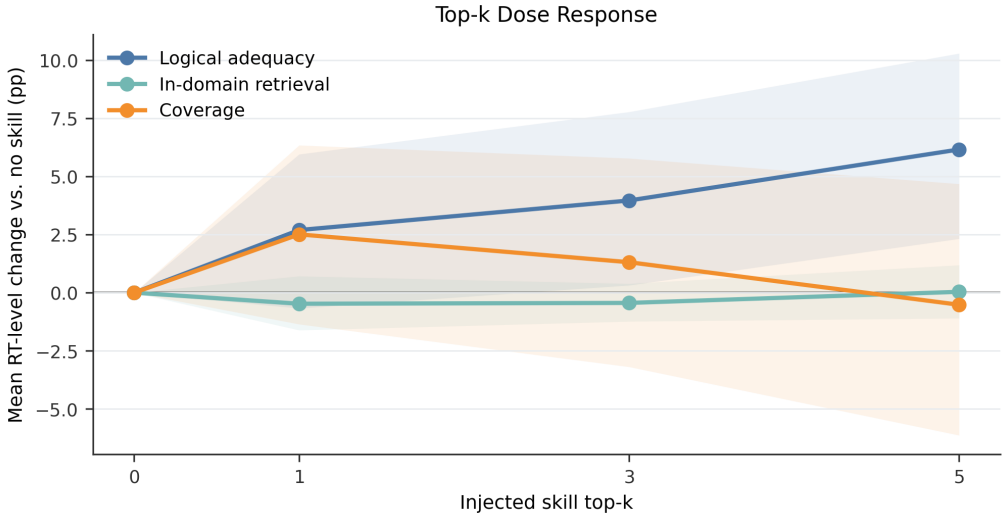


Figure 4.11: Requirement-level dose response when increasing the number of injected skills. Each line shows the mean change relative to the matched no-skill output for one metric: logical adequacy, in-domain retrieval and coverage. The x-axis gives the injected skill depth ($k = 0$ denotes the no-skill baseline), and the y-axis reports mean requirement–testcase-level change in percentage points. The horizontal zero line therefore corresponds to no change from the baseline. Shaded bands show bootstrap 95% confidence intervals for the mean paired change, not standard-deviation bands.

The results show a clear increase in planning-level quality as more skills were injected. Logical adequacy increased from +2.69 percentage points at top-$k = 1$ to +3.96 at top-$k = 3$ and +6.15 at top-$k = 5$. The top-$k = 5$ adequacy improvement was statistically significant under the paired Wilcoxon test ($p = 0.0125$, Cohen's $d_z = 0.48$). The same depth also improved structural quality ($p = 0.0146$, $d_z = 0.40$) and domain fit ($p = 0.0098$, $d_z = 0.39$). Domain fit also reached significance at top-$k = 3$ ($p = 0.0379$, $d_z = 0.34$), suggesting that even moderate skill context is sufficient to improve feature-specific step planning. In contrast, top-$k = 1$ showed its strongest effect on step coverage ($p = 0.0040$, $d_z = 0.45$).

4.5.3 Planning and Grounding Metrics

The effect was not monotonic for downstream grounding and coverage. Overall coverage improved at top-$k = 1$ and top-$k = 3$, but fell slightly at top-$k = 5$.

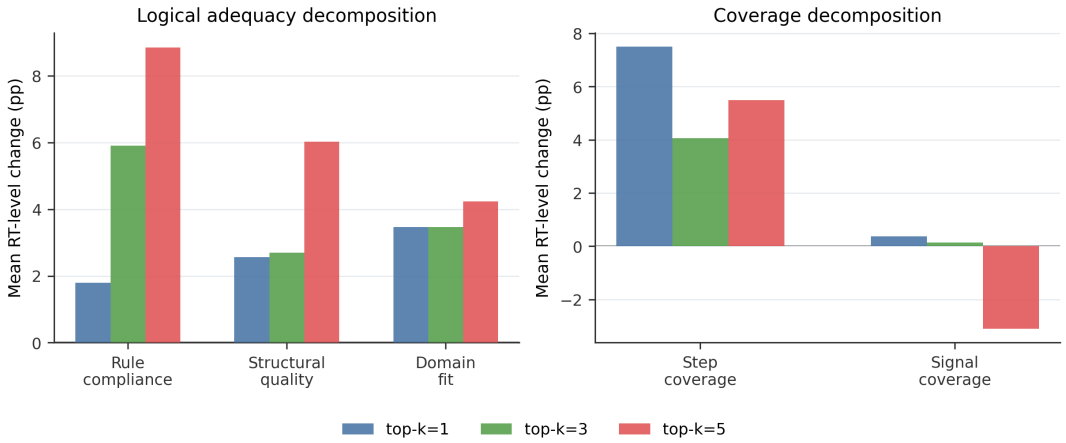


Figure 4.12: Metric decomposition of skill-depth effects across planning, grounding and coverage metrics.

Signal coverage followed the same pattern more strongly, changing by +0.37, +0.13 and -3.10 percentage points for top-$k = 1$, top-$k = 3$, and top-$k = 5$, respectively. The grounding similarity metrics showed little corresponding improvement: top-k similarity changed by -0.47, -0.44 and +0.04 percentage points, while top-1 similarity changed by -0.72, -0.54 and -0.41 percentage points.

4.5.4 Query Form and Plan Complexity

The query-type breakdown further indicates that the benefit was stronger for detailed requirement formulations. For detailed queries, logical adequacy increased by +4.61, +6.72 and +9.00 percentage points for top-$k = 1$, top-$k = 3$, and top-$k = 5$, respectively. The corresponding summary-query gains were +1.44, +1.59 and +4.03 percentage points.

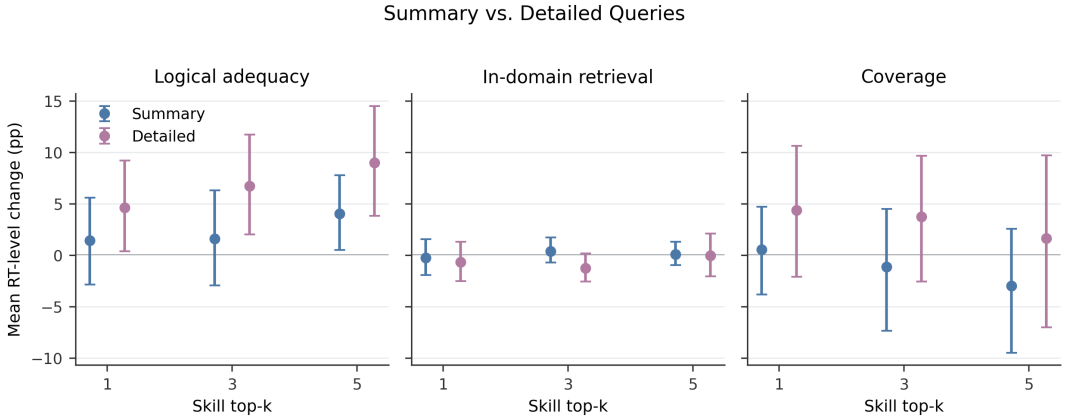


Figure 4.13: Skill-depth sensitivity by query form. Markers show the mean RT-level change in percentage points; error bars are 95% bootstrap confidence intervals (5000 resamples). Detailed queries showed larger logical-adequacy gains than summary queries.

The increased logical adequacy came with higher plan complexity. The mean number of generated steps increased by +1.44 at top-$k = 1$, +1.85 at top-$k = 3$, and +2.06 at top-$k = 5$.

4.5.5 Discussion

The E4 results show different responses across planning and downstream metrics. Higher retrieval depth produced the largest gains in logical adequacy, structural quality and domain fit, whereas grounding similarity and behavioural coverage changed less consistently. This indicates that skill injection mainly affected the planning stage, while logic-to-VTT grounding remained a separate source of variation.

The strongest step-coverage result occurred at top-$k = 1$, while the strongest logical-adequacy result occurred at top-$k = 5$. One interpretation is that a smaller skill context can add missing behavioural steps without introducing as much prompt context, whereas a larger skill context encourages more complete planning but also produces more actions and checks that must be grounded successfully.

The query-form breakdown suggests that retrieved skills were most useful when the input contained richer constraints. Detailed queries showed larger adequacy gains than summary queries at all three retrieval depths, which is consistent with the role of skills as guidance for organising constraint-rich requirements into logical test plans.

One possible reason for the difference between the planning-stage and grounding-stage metrics is that injected skills bring more domain-specific terms into the generated steps. This helps the model follow the rules and produce more logically complete test steps, but it also makes the wording less similar to the original query. The logic-to-VTT grounding stage uses embedding-based retrieval, which is sensitive to wording, so the change in phrasing can reduce signal-level coverage even when the logical plan itself is better. The results match this picture: the planning metrics, including adequacy, rule compliance, and structure, improve as k increases, while signal coverage does not show the same trend. The drop in signal coverage is therefore more likely a side-effect of the interaction between skill injection and retrieval-based signal mapping than a sign of poor skill quality.

4.6 Summary of Results

The retrieval evaluation showed that weighted fusion consistently outperformed single-field retrieval and that BGE-M3 offered the best balance between retrieval quality, local deployment and embedding size. The Phase 1 fine-tuning evaluation showed that the QLoRA-adapted Llama-3.3-70B-Instruct model achieved perfect JSON parseability and schema validity on the held-out set and produced the strongest type-level and domain-style scores, although step-count prediction remained weak. The end-to-end ablation indicated that Skill injection and RAG grounding each improved pipeline quality over Baseline ($\Delta\text{TSS} = +0.240$ and $+0.201$ respectively), with the combined condition achieving the highest TSS of 0.507 and a Pass@τ of 70.2%. Skill injection was the dominant factor, primarily reducing fatal

rule violations, while RAG grounding contributed through improved grounding in-domainness and signal coverage. The skill-depth sensitivity analysis indicated that injecting more retrieved skills improved logical planning quality, with the strongest adequacy, structural and domain-fit gains at top-$k = 5$. Coverage and grounding metrics did not improve monotonically, and their changes were smaller than the planning-metric changes. Both end-to-end evaluations rest on small samples—24 requirement–testcase pairs in E3 and 39 in E4—so these numbers are best read as indicative comparisons between configurations rather than precise estimates of the underlying effects.

5
Conclusion

5.1 Summary of Contributions

This thesis presented a two-phase pipeline for generating vTESTstudio-compatible test artefacts from natural-language automotive requirements. The main design contribution is the separation of the task into Phase 1 (Req2Logic) and Phase 2 (Logic2VTT). Phase 1 converts a requirement into a structured sequence of logical test steps, and Phase 2 then grounds these steps in domain-specific signals and functions before rendering the final VTT artefact. The two-phase split avoids asking the language model to interpret the requirement, select valid domain entities, and produce tool-specific XML in one generation step.

A second contribution is the logical-step intermediate representation used for requirement-driven automotive test generation. This representation makes the generated test logic inspectable before final rendering and allows different parts of the pipeline to be evaluated separately. Around this representation, the work designed and evaluated retrieval-based grounding, supervised fine-tuning for the Req2Logic task, and retrieved skill guidance for improving planning quality. These experiments are empirical rather than algorithmic contributions: they show how the pipeline behaves under different configurations and which components address which failure modes, but they do not introduce new fine-tuning or retrieval algorithms.

The thesis also proposes a multi-layer evaluation setup for this task. Rather than relying only on textual similarity, the evaluation combines retrieval quality, schema validity, step-type structure, logical adequacy, grounding in-domainness, behavioural coverage, and a three-stage end-to-end score.

5.2 Answers to the Research Questions

The research questions are answered at the level of the complete pipeline, because the experiments were designed to study different parts of the same requirement-to-VTT generation process.

RQ1. Natural-language automotive requirements can be automatically transformed into vTESTstudio-compatible test artefacts by decomposing the task into two explicit pipeline stages rather than treating it as a single text-generation problem. In this thesis, Phase 1 (Req2Logic) converted a requirement into a structured sequence of logical test steps with explicit actions, checks, waits and expected outcomes, while Phase 2 (Logic2VTT) grounded these steps in domain-specific signals and functions before rendering the final VTT artefact. The logical-step intermediate representa-

tion made the generated test logic inspectable before final rendering and allowed Phase 1 interpretation errors to be distinguished from later grounding and rendering errors. Together, the two phases show that LLMs can produce vTESTstudio-compatible artefacts when requirement interpretation and tool-specific construction are separated.

RQ2. LLM-generated test artefacts can be grounded in valid domain-specific signals and functions through retrieval-based grounding over the available signal catalogues, function libraries and historical test cases. Field-aware retrieval was important for finding useful historical cases, and weighted fusion outperformed single-field retrieval across the evaluated embedding models. In the end-to-end evaluation, retrieval-based grounding improved grounding in-domainness and signal coverage, showing that retrieval is most useful when the generated logic has to be connected to valid signals and functions.

RQ3. Parameter-efficient fine-tuning improved the structural validity (JSON parse rate, schema valid rate), type consistency (type multiset F_1, type edit similarity), and domain style (domain style accuracy) of the generated intermediate representations. The QLoRA-adapted Llama-3.3-70B model achieved perfect JSON parse ability and schema validity on the held-out test set and produced the strongest type-level and domain-style scores. However, fine-tuning did not fully solve step-count prediction, which suggests that decomposing a requirement into the right number of test actions remains a planning challenge rather than only a formatting problem.

RQ4. Recurring generation errors and implicit domain conventions can be addressed without modifying model weights by injecting retrieved skill guidance into the prompt at inference time. Skill guidance improved planning-oriented aspects of the pipeline, especially logical adequacy, structural quality and domain fit, and reduced fatal planning-rule violations in the end-to-end ablation. The skill-depth experiment showed that larger skill contexts can produce more complete plans, but the gains do not increase monotonically for grounding or behavioural coverage. The pattern points to a trade-off between planning completeness and grounding specificity, and shows that inference-time skill injection can encode implicit domain conventions while still being adjustable per requirement.

RQ5. LLM-generated automotive test artefacts should be evaluated beyond textual similarity because fluent or textually similar outputs may still be invalid as test artefacts. The evaluation therefore combined schema validity, step-type structure, logical adequacy, grounding in-domainness, behavioural coverage and a three-stage end-to-end score. This made it possible to evaluate whether a generated artefact captured the intended behaviour and whether it remained compatible with the available domain resources.

5.3 Limitations

This work has several limitations:

1. The data used in the thesis came from a specific automotive testing context. The signal catalogues, function library, requirement styles and historical test cases reflect one industrial environment, so the results may not transfer directly to other vehicle functions, toolchains or organisations without further

adaptation.

2. The evaluation sets were small. The two end-to-end experiments rest on 24 requirement–testcase pairs in E3 and 39 pairs in E4, which is enough to compare the main pipeline configurations against one another but too few to pin down how the pipeline behaves in general. At this size a handful of outputs can move a reported mean noticeably, so the differences we report are better read as indications than as settled effects, and any stronger claim about robustness across requirement types would need a larger and more varied requirement set.
3. Parts of the evaluation relied on LLM-based judging, especially the logical-adequacy and behavioural-coverage scores that feed into the composite end-to-end results. This was useful because exact string matching is too rigid for requirement-driven test generation, but it also means that these scores are judge-assessed proxies rather than ground-truth measures of correctness. No separate human validation study was conducted to calibrate the judge outputs against test-engineer assessments, so the reported composite scores should be interpreted as structured comparative indicators, not as final evidence that a generated artefact is correct.
4. The pipeline was evaluated as a research prototype rather than as a fully deployed industrial tool. The thesis focused on generation, grounding and artefact-level evaluation. Full integration into a vTESTstudio or CANoe workflow, including execution-time validation and tester-in-the-loop review, remains outside the scope of this work.
5. The experiments showed that some core challenges remain open. Fine-tuning improved format and local conventions but did not fully solve test-step decomposition. Skill guidance improved planning quality but could increase plan complexity. Retrieval improved signal grounding, but summary requirements with few explicit constraints remained difficult.

5.4 Future Work

Several directions can be explored to extend this work:
Future work should first evaluate the pipeline on a larger and more diverse set of requirements, including additional vehicle functions, signal families and test-case styles. This would make it possible to study how well the logical-step representation and grounding strategy generalise beyond the data used in this thesis.
A second direction is stronger execution-level validation. Generated VTT artefacts should be tested in the target vTESTstudio and CANoe workflow, not only compared against reference artefacts. Such evaluation would reveal whether generated tests are syntactically acceptable, executable, and useful to test engineers in practice.
The pipeline could also be extended with explicit validation and repair loops. Rule checks, schema validation, grounding diagnostics and execution feedback could be used to revise generated logical steps or repair grounded scripts before final rendering. This would be a natural extension of the skill-memory mechanism studied in this thesis.
Another direction is adaptive skill retrieval. The results suggest that larger skill contexts improve planning but may hurt downstream grounding. Instead of using

a fixed top-k, future systems could select the number and type of skills based on requirement complexity, query detail and the confidence of the retrieved examples. Finally, the grounding component could be improved with more explicit signal and function constraints. Better candidate ranking, value normalisation and compatibility checks between generated steps and available domain entities may reduce the trade-off between complete planning and signal-specific correctness.

5.5 Ethics and Sustainability

Data handling. The thesis used proprietary requirements, signal catalogues, function libraries and historical test cases from one industrial context. All requirement identifiers, test-case names and signal and function names shown in this report were anonymised. Fine-tuning was restricted to a local open-weight model so that internal requirement and test records did not have to be sent to an external service; only non-sensitive material was exposed to hosted models where they were used.

Automation bias and human oversight. The pipeline is an assistive tool, not a replacement for the test engineer. Because the evaluation reaches only logical-step adequacy and coverage against reference cases—and not syntactic validity or execution—a fluent or high-scoring artefact can still be wrong or non-executable. Presenting generated tests as finished work would risk automation bias, where engineers under-scrutinise plausible output. The two-phase design keeps the intermediate logic inspectable specifically to support human review, and human validation is treated as mandatory before any generated artefact is used.

Safety in the application domain. The target domain is automotive ECU testing, where missed or incorrect checks can let safety-relevant defects pass undetected. The pipeline is therefore positioned as support for test creation, not as a certified safety process, and is not intended to decide on its own whether a vehicle function is safe.

Compute and energy cost. The work relied on energy-intensive models. Fine-tuning and the associated experiments ran on a cloud cluster with two NVIDIA A100-80 GB GPUs, accumulating roughly 21 hours of cluster compute time (about 18.7 hours of logged training, inference and evaluation jobs plus around two hours of interactive use) at a measured cloud cost of about USD 257. Generation and evaluation additionally called hosted models (GPT-4o and DeepSeek-V3), whose own training and serving costs are not visible to this project. Parameter-efficient fine-tuning (QLoRA, 4-bit quantisation, a single training epoch) was chosen partly to limit training cost relative to full fine-tuning, and the local retrieval index avoided a separate vector-database service. Even so, repeated LLM-judge evaluation and large-model inference carry a non-trivial energy footprint, and any deployment should weigh this recurring inference cost against the manual effort the pipeline is intended to reduce.

5.6 Final Remarks

Overall, the thesis shows that requirement-driven automotive test generation becomes easier to control when requirement interpretation, domain grounding and

artefact construction are kept as separate pipeline stages. The two-phase pipeline and the logical-step representation do not remove the need for engineering judgement, but they give a structured basis for inspecting, evaluating and improving LLM-assisted test generation in an industrial automotive testing context. The evaluation in this thesis reaches the generated logical steps and a coverage comparison of the VTT artefacts against reference cases; syntactic validation and execution in vTESTstudio or CANoe were not part of this work and remain for future evaluation.

Bibliography

[1] Vector Informatik GmbH, "vteststudio," 2026, accessed: 2026-05-18. [Online]. Available: https://www.vector.com/int/en/products/products-a-z/software/vteststudio/

[2] E. Hull, K. Jackson, and J. Dick, *Requirements Engineering*, 3rd ed. London, U.K.: Springer, 2011.

[3] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, M. A. Ajagbe, E.-V. Chioasca, and R. T. Batista-Navarro, "Natural language processing for requirements engineering: A systematic mapping study," *ACM Comput. Surv.*, vol. 54, no. 3, pp. 55:1–55:41, 2021.

[4] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (ears)," in *Proc. 17th IEEE Int. Requirements Eng. Conf. (RE)*, 2009, pp. 317–322.

[5] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "NAT2TEST(SCR): Test case generation from natural language requirements based on SCR specifications," *Sci. Comput. Program.*, vol. 95, pp. 275–297, 2014.

[6] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *Proc. Future Softw. Eng. (FOSE)*, 2007, pp. 55–71.

[7] A. Kasoju, K. Petersen, and M. V. Mäntylä, "Analyzing an automotive testing process with evidence-based software engineering," *Inf. Softw. Technol.*, vol. 55, no. 7, pp. 1237–1259, 2013.

[8] K. Juhnke, M. Tichy, and F. Houdek, "Challenges concerning test case specifications in automotive software testing: assessment of frequency and criticality," *Softw. Qual. J.*, vol. 29, pp. 39–100, 2021.

[9] E. Bringmann and A. Krämer, "Model-based testing of automotive systems," in *Proc. 1st Int. Conf. Softw. Testing, Verification, Validation (ICST)*, 2008, pp. 485–493.

[10] K.-W. Shin and D.-J. Lim, "Model-based automatic test case generation for automotive embedded software testing," *Int. J. Automot. Technol.*, vol. 19, no. 1, pp. 107–119, 2018.

[11] Vector Informatik GmbH, "Example how to use xcp in vteststudio," Vector Informatik GmbH, Support Note SN-IND-1-044, 2020. [Online]. Available: https://support.vector.com/sys_attachment.do?sys_id=c59e25cb1b73dc908e9a535c2e4bcbed

[12] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 35, 2022, pp. 22 199–22 213.

[13] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 220:1–220:79, 2024.

[14] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2022. [Online]. Available: https://openreview.net/forum?id=gEZrGCozdqR

[15] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 35, 2022, pp. 27 730–27 744.

[16] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 36, 2023, pp. 10 088–10 115.

[17] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 9459–9474.

[18] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W. Yih, "Dense passage retrieval for open-domain question answering," in *Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP)*. Association for Computational Linguistics, 2020, pp. 6769–6781.

[19] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proc. Conf. Empir. Methods Nat. Lang. Process. 9th Int. Joint Conf. Nat. Lang. Process. (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, 2019, pp. 3982–3992.

[20] D. Milchevski, G. Frank, A. Hätty, B. Wang, X. Zhou, and Z. Feng, "Multi-step generation of test specifications using large language models for system-level requirements," in *Proc. 63rd Annu. Meeting Assoc. Comput. Linguistics (ACL), Vol. 6: Ind. Track*. Vienna, Austria: Association for Computational Linguistics, 2025, pp. 132–146. [Online]. Available: https://aclanthology.org/2025.acl-industry.11/

[21] L. Dong and M. Lapata, "Coarse-to-fine decoding for neural semantic parsing," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics (ACL), Vol. 1: Long Papers*. Melbourne, Australia: Association for Computational Linguistics, 2018, pp. 731–742. [Online]. Available: https://aclanthology.org/P18-1068/

[22] R. Shin, C. H. Lin, S. Thomson, C. Chen, S. Roy, E. A. Platanios, A. Pauls, D. Klein, J. Eisner, and B. Van Durme, "Constrained language models yield few-shot semantic parsers," in *Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP)*. Online and Punta Cana, Dominican Republic:

Association for Computational Linguistics, 2021, pp. 7699–7715. [Online]. Available: https://aclanthology.org/2021.emnlp-main.608/

[23] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics (ACL), Vol. 1: Long Papers*. Vancouver, Canada: Association for Computational Linguistics, 2017, pp. 440–450. [Online]. Available: https://aclanthology.org/P17-1041/

[24] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics (ACL), Vol. 1: Long Papers*. Vancouver, Canada: Association for Computational Linguistics, 2017, pp. 1139–1149. [Online]. Available: https://aclanthology.org/P17-1105/

[25] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing incrementally for constrained auto-regressive decoding from language models," in *Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP)*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021, pp. 9895–9901. [Online]. Available: https://aclanthology.org/2021.emnlp-main.779/

[26] I. Paul, G. Glavaš, and I. Gurevych, "IRCoder: Intermediate representations make language models robust multilingual code generators," in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (ACL), Vol. 1: Long Papers*. Bangkok, Thailand: Association for Computational Linguistics, 2024, pp. 15 023–15 041. [Online]. Available: https://aclanthology.org/2024.acl-long.802/

[27] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 36, 2023, pp. 8634–8652. [Online]. Available: https://openreview.net/forum?id=vAElhFcKW6

[28] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, "Self-refine: Iterative refinement with self-feedback," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 36, 2023, pp. 46 534–46 594. [Online]. Available: https://openreview.net/forum?id=S37hOerQLB

[29] H. Liu, C. Sferrazza, and P. Abbeel, "Chain of hindsight aligns language models with feedback," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=6xfe4IVcOu

[30] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *Trans. Mach. Learn. Res.*, 2024. [Online]. Available: https://voyager.minedojo.org/

[31] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative agents: Interactive simulacra of human behavior," in *Proc. 36th Annu. ACM Symp. User Interface Softw. Technol. (UIST)*, San Francisco, CA, USA, 2023, pp. 2:1–2:22.

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, " Feedback-Directed Random Test Generation ," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*. Los

Alamitos, CA, USA: IEEE Computer Society, May 2007, pp. 75–84. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE.2007.37

[33] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. (ESEC/FSE)*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[34] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Trans. Softw. Eng.*, vol. 50, no. 1, pp. 85–105, 2024.

[35] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proc. 32nd ACM Int. Conf. Found. Softw. Eng. (FSE)*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 572–576. [Online]. Available: https://doi.org/10.1145/3663529.3663801

[36] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *Inf. Softw. Technol.*, vol. 176, no. C, Dec. 2024. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107565

[37] S. Wynn-Williams, R. Tyrrell, V. Pantelic, M. Lawford, C. Menghi, P. Nalla, and H. Artail, "Can generative ai produce test cases? an experience from the automotive domain," in *Companion Proc. 33rd ACM Int. Conf. Found. Softw. Eng. (FSE)*, ser. FSE Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 456–467. [Online]. Available: https://doi.org/10.1145/3696630.3728568

[38] A. Karlsson, E. Lindmaa, S. Sun, and M. Staron, "Ai-based automotive test case generation: An action research study on integration of generative ai into test automation frameworks," in *Companion Proc. 25th Int. Conf. Product-Focused Softw. Process Improvement (PROFES), Ind., Workshop, Doctoral Symp. Papers*. Berlin, Germany: Springer-Verlag, 2024, p. 50–66. [Online]. Available: https://doi.org/10.1007/978-3-031-78392-0_4

[39] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge University Press, 2008.

[40] N. Craswell, "Mean reciprocal rank," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Boston, MA, USA: Springer, 2009, p. 1703.

[41] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics (ACL)*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040/

[42] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Inf. Process. Manage.*, vol. 45, no. 4, pp. 427–437, 2009.

[43] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Sov. Phys. Dokl.*, vol. 10, no. 8, pp. 707–710, 1966.

[44] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Workshop Text Summarization Branches Out*. Barcelona, Spain:

Association for Computational Linguistics, 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013/

[45] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTScore: Evaluating text generation with BERT," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2020. [Online]. Available: https://arxiv.org/abs/1904.09675

[46] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: https://doi.org/10.2307/3001968

A

Appendix 1

A.1 Skill Retrieval Depth Sensitivity Results

Table A.1 reports the requirement-level results for the skill retrieval-depth sensitivity analysis. The values were computed from `combined_rt_level_summary.csv` in `experiment/visualize/results`. Each condition was compared with the corresponding no-skill baseline for the same requirement-testcase pair. Positive values therefore indicate an increase after skill injection. All changes are reported in percentage points, except for generated steps, which are reported as the change in the mean number of steps.

Table A.1: Requirement-level skill retrieval-depth sensitivity results.

Setting	**Metric**	**Change**	p	d_z
top-$k = 1$	Adequacy	+2.69	0.1048	0.25
top-$k = 1$	Structural quality	+2.56	0.1489	0.16
top-$k = 1$	Domain fit	+3.46	0.0559	0.28
top-$k = 1$	Rule validity	+1.79	0.7255	0.07
top-$k = 1$	Coverage	+2.51	0.2087	0.20
top-$k = 1$	Step coverage	+7.50	0.0040	0.45
top-$k = 1$	Signal coverage	+0.37	0.8103	0.02
top-$k = 1$	Top-k similarity	-0.47	0.2491	-0.13
top-$k = 1$	Top-1 similarity	-0.72	0.1831	-0.18
top-$k = 1$	Retrieval margin	-0.30	0.5028	-0.08
top-$k = 1$	Generated steps	+1.44	0.0030	0.52
top-$k = 3$	Adequacy	+3.96	0.1844	0.32
top-$k = 3$	Structural quality	+2.69	0.2476	0.19
top-$k = 3$	Domain fit	+3.46	0.0379	0.34
top-$k = 3$	Rule validity	+5.90	0.2935	0.18
top-$k = 3$	Coverage	+1.31	0.5971	0.09
top-$k = 3$	Step coverage	+4.05	0.1641	0.20
top-$k = 3$	Signal coverage	+0.13	0.9810	0.01
top-$k = 3$	Top-k similarity	-0.44	0.2072	-0.17
top-$k = 3$	Top-1 similarity	-0.54	0.4317	-0.16
top-$k = 3$	Retrieval margin	-0.12	0.9608	-0.04
top-$k = 3$	Generated steps	+1.85	0.0001	0.70
top-$k = 5$	Adequacy	+6.15	0.0125	0.48
top-$k = 5$	Structural quality	+6.03	0.0146	0.40
top-$k = 5$	Domain fit	+4.23	0.0098	0.39
top-$k = 5$	Rule validity	+8.85	0.1078	0.28
top-$k = 5$	Coverage	-0.52	0.8989	-0.03
top-$k = 5$	Step coverage	+5.49	0.0942	0.29
top-$k = 5$	Signal coverage	-3.10	0.4683	-0.15
top-$k = 5$	Top-k similarity	+0.04	0.8710	0.01
top-$k = 5$	Top-1 similarity	-0.41	0.1198	-0.12
top-$k = 5$	Retrieval margin	-0.55	0.0922	-0.17
top-$k = 5$	Generated steps	+2.06	0.0008	0.50

The overall pattern shows that deeper skill retrieval mainly improved planning-oriented metrics. The top-$k = 5$ condition produced the largest gains in adequacy, structural quality and domain fit, while coverage and signal coverage did not improve monotonically. Generated step count increased at all retrieval depths, indicating that injected skills encouraged more complete logical plans but also introduced more steps that had to be grounded downstream.

A.2 Example Data Formats

This section gives worked examples of the data formats described in Section 3.1.5. All identifiers, signal names and text are fictitious; only the structure reflects the original data.

A.2.1 Training Instruction-Response Pair

Each training sample paired a requirement with its logical-step sequence, formatted as an instruction-response pair for supervised fine-tuning (Listing A.1).

Listing A.1: Training instruction-response pair.

```
{
  "instruction": "Generate the logical test steps for the
     requirement.",
  "input": {
    "req_id": "REQ-001",
    "name": "DeactivateDoorLock",
    "purpose": "Verify that the rear door locks are deactivated
       when the vehicle is stationary.",
    "description": "When DeactivateDoorLock is called, the system
       shall deactivate the rear door locks if DoorLockSensorStatus
        = Active and VehicleSpeed = 0."
  },
  "output": {
    "testcase_name": "Deactivate rear door lock at standstill",
    "description": "Checks rear door lock deactivation when the
       vehicle is stationary.",
    "logical_steps": [
      {"type": "call",   "text": "Activate driving usage mode."},
      {"type": "set",    "text": "Set rear-left door lock status to
          Active."},
      {"type": "verify", "text": "Verify rear-left door lock status
          equals Active within 8000 ms."},
      {"type": "intent", "text": "Request door lock deactivation
         ."},
      {"type": "verify", "text": "Verify door lock activation type
         equals Deactivated within 2000 ms."}
    ]
  }
}
```

A.2.2 Historical-Case Index Entry

Each entry of the historical-case index stored the requirement and testcase content together with three precomputed field vectors used in Phase 1 retrieval (Listing A.2). The vector components are truncated for readability.

Listing A.2: Historical-case index entry.

```
{
  "Name": "Deactivate rear door lock at standstill",
  "RT_id": "RT_001",
  "Requirement": {
    "ReqID": "REQ-001",
    "ReqName": "DeactivateDoorLock",
    "Purpose": "Verify rear door lock deactivation at standstill.",
    "Description": "When DeactivateDoorLock is called, ..."
  },
  "Testcase": {
    "TestCaseID": "TC-001",
    "TestCaseName": "Deactivate rear door lock at standstill",
    "logical_steps": [ ... ]
  },
  "VectorPurpose":     [ 0.018, -0.004, 0.022, ... ],
  "VectorDescription": [-0.011,  0.022, 0.010, ... ],
  "VectorSignals":     [ 0.007, -0.031, 0.005, ... ]
}
```

A.2.3 Grounding-Index Entry

Each entry of the three Phase 2 grounding indices stored a candidate name, its readable description, and a single vector computed from that description (Listing A.3). The example shows a `DBSignal` entry; the `SysVar` and function entries follow the same structure.

Listing A.3: Grounding-index entry (`DBSignal`).

```
{
  "name": "RearLDoorLockSts",
  "expand": "rear-left door lock status. Allowed values: 0:Unlk, 1:
     Lk, 2:Err",
  "vector": [-0.077, -0.011, 0.009, 0.026, ... ]
}
```

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se