Can Generative AI Produce Test Cases? An Experience from the Automotive Domain

Stephen Wynn-Williams
McSCert, McMaster University
Hamilton, ON, Canada
wynnwisj@mcmaster.ca

Ryan Tyrrell
McSCert, McMaster University
Hamilton, ON, Canada
tyrrellr@mcmaster.ca

Vera Pantelic
McSCert, McMaster University
Hamilton, ON, Canada
pantelv@mcmaster.ca

Mark Lawford
McSCert, McMaster University
Hamilton, ON, Canada
lawford@mcmaster.ca

Claudio Menghi
University of Bergamo
Bergamo, Italy
claudio.menghi@unibg.it

Phaneendra Nalla
FCA US LLC
Auburn Hills, MI, USA
phaneendra.nalla@stellantis.com

Hassan Artail
FCA US LLC
Auburn Hills, MI, USA
hassan.artail1@stellantis.com

ABSTRACT

Engineers need automated support for software testing. Generative AI is a novel technology for generating new content; however, its applicability for test case generation is still unclear. This work considers the following question: *Can generative AI produce test cases in industrial software applications*? We framed our question in the automotive domain. We performed our evaluation in collaboration with a large automotive manufacturer to assess to what extent generative AI can produce test cases (a.k.a. test scripts) from informal test case specifications. We considered 1) informal test case specifications defined in Rational Quality Manager, an industrial test management tool from IBM, and 2) executable test scripts specified as ecu.test packages supported by the ecu.test tool from Tracetronic. We used generative AI to produce the test scripts from the informal test case descriptions. Our results show that generative AI can produce correct or near-correct test scripts in a reasonable number of cases. We also analyzed the effects of prompt design, choice of generative AI model, and context accuracy on the effectiveness of our solution and reflected on our results.

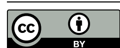
CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Software Testing, LLM, generative AI, Automotive Software

ACM Reference Format:
Stephen Wynn-Williams, Ryan Tyrrell, Vera Pantelic, Mark Lawford, Claudio Menghi, Phaneendra Nalla, and Hassan Artail. 2025. Can Generative AI Produce Test Cases? An Experience from the Automotive Domain. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3696630.3728568



This work is licensed under a Creative Commons Attribution 4.0 International License.

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1276-0/2025/06.
https://doi.org/10.1145/3696630.3728568

1 INTRODUCTION

The automotive industry strongly relies on software testing to verify the presence of software failures [16, 17]. Tests drastically improve failure detection at multiple stages of development [43]. However, developing and maintaining automated tests requires significant effort [43]. Engineers usually start by defining a test plan [27]. This test plan aims to exercise the software requirements by defining a set of *informal test case specifications*. Informal test case specifications are typically abstract since the software-under-test is not yet implemented when the test plan is developed. As the automotive software becomes concrete, the informal test case specifications are converted into *test scripts* defined by *test instructions*. In this work, we consider informal test case specifications specified using Rational Quality Manager (RQM) [23], an industrial test management tool from IBM [22], and test scripts expressed with ecu.test [51], an industrial tool from Tracetronic [50], since these tools are largely used within the automotive domain [16, 17].

Informal test case specifications are typically manually translated into test scripts, a time-consuming and error-prone activity [15]. This activity can be automated using generative AI technologies. Recent advances in generative AI technologies proposed using large language models (LLMs) for several tasks, such as writing of formal specifications [12], hazard analysis [14], goal modeling [10], and software modeling [9], and assurance case development [37, 53–55]. Developing test cases ([47, 52], to name a few) is also a major area of application of LLM and the generative AI technology.

This work evaluates how LLM can help automatically translate informal test case specifications into test scripts within the *automotive context*. Unlike existing works, we consider informal test case specifications defined in RQM and test scripts expressed with ecu.test. Additionally, in our study (a) we could not access a large dataset of informal test case specifications and test scripts that can

be used for training, and (b) we must maintain the confidentiality of the data provided by our automotive industrial partner.

To perform our evaluation we implemented an approach to convert informal RQM test case specifications into test scripts specified in ecu.test. Our solution relies on Retrieval Augmented Generation (RAG) [30] to retrieve relevant background information (a.k.a. *context*) from an auxiliary data source, and *few-shot learning*, a prompting technique for adapting a pre-trained text generation model to a novel task without additional training [8], to generate the test instruction from the a prompt that includes the context and the informal test case specitication.

We assessed the capability of our solution to produce test scripts that correctly implement their informal specification by considering a benchmark of 200 unique pairs of informal test step descriptions and the corresponding executable test instructions. We also analyzed how the effectiveness of our solution depends on the prompt design, the use of different LLM solutions, and the accuracy of the context. Our results show that few-shot learning (properly configured) can produce test scripts that correctly implement their informal specification between 49.5% and 64.5% of the cases. However, when not correct, the specifications are highly similar to the one defined by humans and might be a valid starting point for the implementation of the test cases. Our results also show that changing the ordering between the context and the informal specification does not significantly affect the effectiveness of our solution, LLM variants tuned for code generation outperform their counterparts, and unnecessary and inaccurate context significantly reduces the efficacy of test generation. Our results can benefit practitioners within the automotive domain and developers of LLM-based domain-specific testing tools: the effectiveness of the solution can frame the expectations of engineers, while the results related to the prompt design, LLM variants, and context can help engineers and developers of LLM-based test solutions tune the proposed solution.

Our paper is organized as follows. Section 2 provides the context and assumptions for the work. Section 3 defines our problem and requirements. Section 4 describes the testing approach considered in this work. Section 5 present our evaluation. Section 6 discusses our results and presents threats to validity. Section 7 summarizes the related work. Section 8 provides our conclusions.

2 CONTEXT AND ASSUMPTIONS

Tests for automotive software are typically executed on Hardware-in-the-Loop (HIL) platforms where the software interacts with a hardware platform. HIL platforms allow testers to access variables from the software under test, the plant model and the vehicle interfaces such as a Controller Area Network (CAN) bus.

This paper considers the process for developing automotive test cases under two major assumptions (**A1** and **A2**).

> **Assumption A1**
> Engineers define test case behaviors as *informal test case specifications* in Rational Quality Manager.

Rational Quality Manager (RQM) [23] is an industrial test management tool from IBM [22]. RQM provides an environment for engineers to design test procedures and supports traceability with their associated requirements, executable test scripts, and test execution results. Test procedures in RQM are specified by test *steps* with an informal language. For example, the RQM informal test case specification shown in Figure 1a describes how to test the push-to-start functionality of a vehicle in three steps: setting the presence of the wireless key fob (**Step 1**), simulating the pressing of the start button (**Step 2**), and checking the value of the start command (**Step 3**).

Each step in an RQM test procedure is described using two text fields. The first field (Summary) describes the step using natural language, and is typically written as a short imperative sentence. The second field (Details) describes information needed to execute the step not specified in the summary. While the details are often more concrete than the summary (e.g., including snippets of pseudocode), they are not written in a consistent formal language (notice the inconsistency in the "==" and "=" operators). For example, the summary of **Step 1** from Figure 1a indicates that the step should set the presence of the key, and the details specify that this is to be achieved by assigning a particular value to the KeyPresence signal.

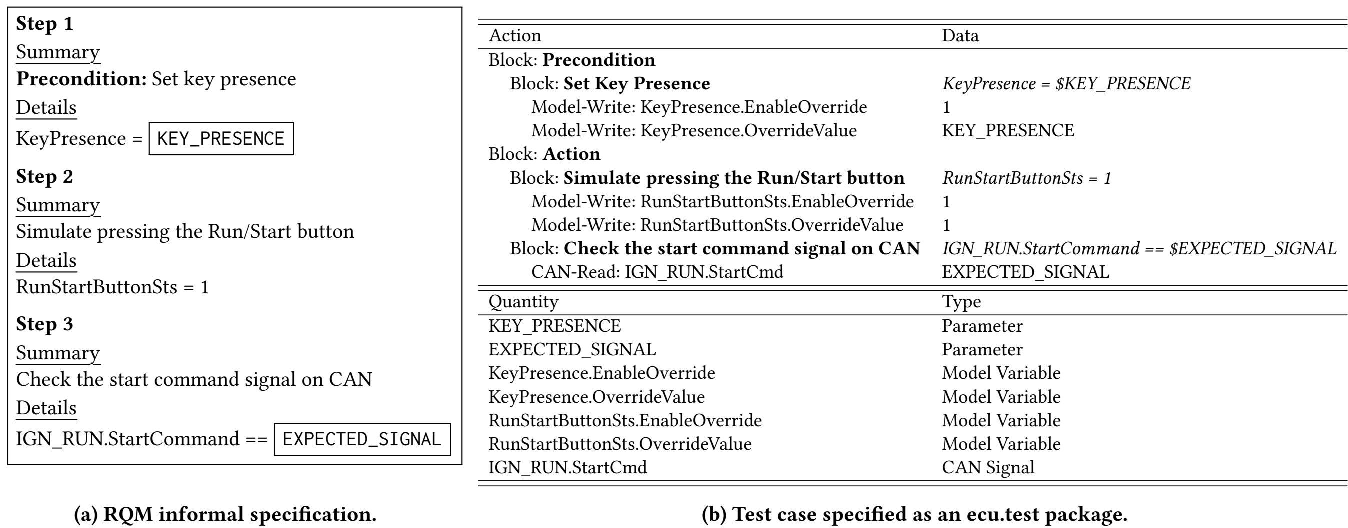
The text fields in RQM test procedures may reference *test parameters*: variables that can take on different values during the execution of a test procedure, allowing it to capture multiple test scenarios. References to test parameters in test procedure specifications are tracked by RQM, and are visualized in the tool by surrounding the names of parameters with a box. For example, the test procedure in Figure 1a uses a test parameter (KEY_PRESENCE) to determine whether the key fob should be present or absent for a particular execution of the test procedure.

Our industrial partner associates each test step with one of three *test procedure stages*: the precondition stage, the action stage, or the postcondition stage. This association is established in RQM through the use of keywords that may be included at the beginning of the summary of a test step. Summaries of test steps associated with the precondition or postcondition stages will start with a keyword indicating the name of the stage, and all other steps are associated with the action stage. For example, the summary of **Step 1** from the test procedure specification in Figure 1a starts with "Precondition," indicating it is part of the precondition stage; **Step 2** and **Step 3** are part of the action stage.

> **Assumption A2**
> *Test scripts* are implemented as ecu.test packages.

The *ecu.test* tool [51] from Tracetronic [50] supports test automation for automotive software. Test scripts are implemented as ecu.test *packages* which execute against a HIL test platform, and are typically produced in a graphical development environment. Figure 1b presents an ecu.test package similar to how it would appear in the ecu.test editor.

An ecu.test package is specified as a sequence of *test instructions*. Test instructions are grouped into hierarchical *blocks* providing a sequence of instructions with a title and (optional) description. For example, the ecu.test package in Figure 1b organizes test instructions into two high-level blocks titled **Precondition** and **Action**. The blocks within each high-level block have been provided with



Step 1
Summary
Precondition: Set key presence
Details
KeyPresence = KEY_PRESENCE

Step 2
Summary
Simulate pressing the Run/Start button
Details
RunStartButtonSts = 1

Step 3
Summary
Check the start command signal on CAN
Details
IGN_RUN.StartCommand == EXPECTED_SIGNAL

(a) RQM informal specification.

Action	Data
Block: **Precondition**	
Block: **Set Key Presence**	*KeyPresence = $KEY_PRESENCE*
Model-Write: KeyPresence.EnableOverride	1
Model-Write: KeyPresence.OverrideValue	KEY_PRESENCE
Block: **Action**	
Block: **Simulate pressing the Run/Start button**	*RunStartButtonSts = 1*
Model-Write: RunStartButtonSts.EnableOverride	1
Model-Write: RunStartButtonSts.OverrideValue	1
Block: **Check the start command signal on CAN**	*IGN_RUN.StartCommand == $EXPECTED_SIGNAL*
CAN-Read: IGN_RUN.StartCmd	EXPECTED_SIGNAL

Quantity	Type
KEY_PRESENCE	Parameter
EXPECTED_SIGNAL	Parameter
KeyPresence.EnableOverride	Model Variable
KeyPresence.OverrideValue	Model Variable
RunStartButtonSts.EnableOverride	Model Variable
RunStartButtonSts.OverrideValue	Model Variable
IGN_RUN.StartCmd	CAN Signal

(b) Test case specified as an ecu.test package.

Figure 1: Example RQM test procedure specification and corresponding ecu.test package.

both a title (e.g., **Set Key Presence**) and a description (e.g., *KeyPresence = $KEY_PRESENCE*). These blocks are further refined into sequences of test instructions: the **Set Key Presence** and **Simulate pressing the Run/Start button** blocks each contain two *Model-Write* instructions, and the **Check the start command signal on CAN** block contains a *CAN-Read* instruction.

Each test instruction performs a particular type of action, and is configured with details about how that action is performed. For example, the *Action* column in Figure 1b indicates that the first instruction in the **Set key presence** block is a *Model-Write* which writes to the `KeyPresence.EnableOverride` variable in the plant model; the value to be written (1) is indicated in the *Data* column.

Each ecu.test package modifies the values of a set of variables from the HIL platform (i.e. plant model variables, etc.), and test package parameters. For example, the bottom half of the ecu.test package illustrated in Figure 1b indicates that the package has been configured with access to two parameters, four model variables, and a CAN signal. As with parameters in RQM test procedure specifications, test package parameters are used to allow a single test package to test multiple scenarios. The variables referenced in ecu.test packages are selected from the ecu.test *workspace* which contains all the variables of the platform under test.

In addition to instructions such as read and write, ecu.test offers standard programming constructs such as loops and subroutines.

Engineers manually convert informal test case specifications into test scripts. For example, the informal specification from Figure 1a is translated into the test script from Figure 1b. It is desirable to automate this process.

3 PROBLEM AND REQUIREMENTS

In this work, we study how generative AI can help converting informal test case specifications into test scripts. Specifically, we consider the following problem.

Problem

Automate the conversion of informal RQM test case specifications into test scripts specified in ecu.test.

To successfully address the needs of our industrial partner, our solution must comply with the following three requirements.

Requirement R1

The produced test scripts shall execute against an existing implementation of the software under test, using quantities available in the testing platform.

Implementing RQM test procedures as test scripts in ecu.test usually requires knowledge of the implementation and target platform. For example, the informal test case specification from the **Step 1** of the test procedure in Figure 1a indicates that the KeyPresence signal should be assigned the value of the `KEY_PRESENCE` parameter. The plant model however does not permit a direct assignment to this signal but requires overriding the signal's normal value. Using this mechanism involves setting the values of two variables from the plant model (`KeyPresence.EnableOverride` and `KeyPresence.OverrideValue`), as depicted in Figure 1b. While the informal test case specification provides details about which variable to modify and its prescribed value, the override mechanism is an implementation detail that must be considered when producing the corresponding test script. Additionally, the names used for quantities may differ between the informal test case specification and the platform, for example, the `IGN_RUN.StartCommand` signal from **Step 3** of Figure 1a becomes `IGN_RUN.StartCmd` in Figure 1b.

Requirement R2

Training the generation approach shall not require a large dataset of informal test case specifications and test scripts.

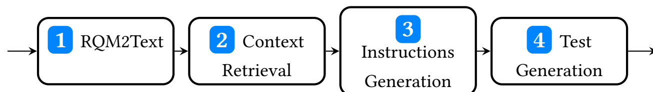
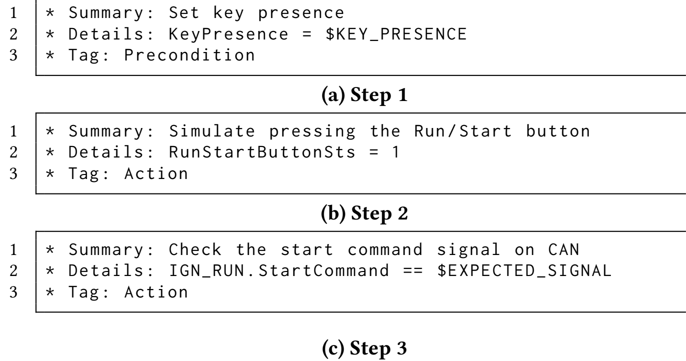


Figure 2: Translating informal test case specifications into test scripts.



```
1 * Summary: Set key presence
2 * Details: KeyPresence = $KEY_PRESENCE
3 * Tag: Precondition
```

(a) Step 1

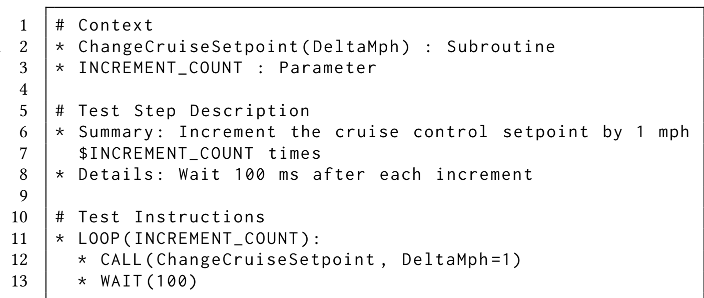
```
1 * Summary: Simulate pressing the Run/Start button
2 * Details: RunStartButtonSts = 1
3 * Tag: Action
```

(b) Step 2

```
1 * Summary: Check the start command signal on CAN
2 * Details: IGN_RUN.StartCommand == $EXPECTED_SIGNAL
3 * Tag: Action
```

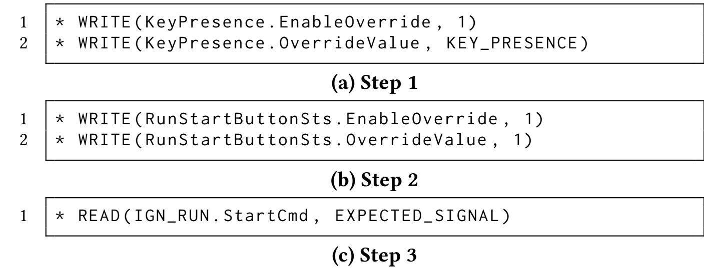
(c) Step 3

Figure 3: Textual specification for the RQM specification from Figure 1a.



```
1  # Context
2  * ChangeCruiseSetpoint(DeltaMph) : Subroutine
3  * INCREMENT_COUNT : Parameter
4
5  # Test Step Description
6  * Summary: Increment the cruise control setpoint by 1 mph
7    $INCREMENT_COUNT times
8  * Details: Wait 100 ms after each increment
9
10 # Test Instructions
11 * LOOP(INCREMENT_COUNT):
12   * CALL(ChangeCruiseSetpoint, DeltaMph=1)
13   * WAIT(100)
```

Figure 4: Prompt example for the instructions generator.



```
1 * WRITE(KeyPresence.EnableOverride, 1)
2 * WRITE(KeyPresence.OverrideValue, KEY_PRESENCE)
```

(a) Step 1

```
1 * WRITE(RunStartButtonSts.EnableOverride, 1)
2 * WRITE(RunStartButtonSts.OverrideValue, 1)
```

(b) Step 2

```
1 * READ(IGN_RUN.StartCmd, EXPECTED_SIGNAL)
```

(c) Step 3

Figure 5: Test instructions for each test step from Figure 3.

During our collaboration, we could access a limited number of informal specifications and test scripts. Therefore, only a limited number of examples can be used to train ML technologies.

Requirement R3

The generation approach shall not expose confidential data to external parties.

To ensure data confidentiality, industrial data must only be used on machines that are in a secured network.

4 TEST CASE GENERATION

Figure 2 presents our approach to translating informal test case specifications into test scripts.

RQM2Text (1) extracts the textual specification from an RQM file. For example, Figures 3a to 3c illustrate how the textual specification from the RQM informal test case specification in Figure 1a is represented. The text representation uses a simple human-readable cue to label the summary and details texts and indicate which quantity names refer to parameters (i.e., using the dollar sign). Additionally, unlike RQM, the textual representation explicitly indicates their associated test procedure stage using a separate field ("Tag"). In the following, we will seamlessly indicate as informal test case specification the RQM informal test case specification and its textual specification.

Context Retrieval (2) analyzes the informal specification step-by-step and retrieves relevant information from the ecu.test workspace. We rely on Retrieval Augmented Generation (RAG) [30] to identify contextual information. The retriever identifies the platform quantities, test parameters, and subroutines that are necessary to implement the informal test case specification in ecu.test. Collectively, the retrieved information provides the *context* necessary to produce the desired test package. For example, when presented with the informal specification from Figure 3a, the retriever should identify the variables `KeyPresence.EnableOverride` and `KeyPresence.OverrideValue`, and the parameter `KEY_PRESENCE`.

Instructions Generation (3) processes the informal test case specification step-by-step and their context using a generative AI model that outputs the corresponding sequence of test instructions. Figure 4 presents an example of a prompt containing the informal test case specification step-by-step and their context. Table 1 describes the instructions for the test cases considered in this paper. Figure 5 contains the instructions outputted by the instructions generator component for the informal specification in Figure 3 generated by the instructions generator.

Few-shot learning is used for text generation, i.e., a few examples are provided to the generative model to learn the translation paradigm. The prompt also incorporates (i) a natural language overview of the task; (ii) a list of all ecu.test instructions that can be used, along with their parameters; (iii) instructions for how to interpret the informal specification (e.g. using dollar signs to indicate parameters); and (iv) a list of conventions and design patterns to follow (e.g. using `EnableOverride` and `OverrideValue` variables to override signals in the plant model).

Test Generation (4) combines the responses from the text generator into a single ecu.test test script. The contexts used for generating the responses are also used in the production of the test script, as they are needed to add references to the elements used.

Our approach satisfies Requirement **R1** by leveraging the information available from the ecu.test workspace (*Context Retrieval* 2) to generate instructions that reference existing quantities from the

Table 1: Plain text representations of ETIs

Instruction Syntax	Description
WRITE(QUANTITY, VALUE)	Writes a value to a model variable, ECU variable, or CAN signal.
READ(QUANTITY, VALUE)	Reads a quantity (variable or signal) and checks its value. Failing this check fails the test case.
READ(QUANTITY, VALUE, timeframe=TIMEFRAME)	Check the variable's value within the given timeframe.
WAIT(TIME)	Waits a specified number of milliseconds before proceeding with the next step.
CALL(SUBROUTINE, **PARAMETERS)	Calls a subroutine and assigns its parameters.
LOOP(COUNT)	Loops over a sequence of instructions a specified number of times.

testing platform. It satisfies **R2** by using few-shot learning to adapt a pretrained text generation model without the need for a training dataset (*Instructions Generation* 3). Finally, it satisfies **R3** by considering text-generation models that can be executed locally by machines on a secured network.

5 EVALUATION

The success of our approach depends on the performance of the *Context Retrieval* (2) and *Instructions Generation* (3). In this work, we study the instructions generator component since our solution would be ineffective if a perfect context retrieval procedure is present, but the instruction generator is ineffective. Developing (and assessing) effective context retrieval procedures is future work.

To assess the effectiveness of the text generation we considered the following research questions.

- **Correctness (RQ1)**. Can our solution produce test scripts that correctly implement their informal test case specification? (Section 5.4)
- **Effects of the Prompt Design (RQ2)**. How does the prompt design affect the test generation results? (Section 5.5)
- **Effects of LLM Variants (RQ3)**. How does the selection of LLM variant affect the test generation results? (Section 5.6)
- **Effects of Context Accuracy (RQ4)**. How sensitive is generation to the accuracy of context retrieval? (Section 5.7)

Before answering our research questions, we present the industrial benchmark considered in this study (Section 5.1), our methodology (Section 5.3), and evaluation metrics (Section 5.3).

5.1 Benchmark

Our industrial partner provided a set of 36 test pairs including an informal test case specification and the corresponding test script under development. The test case specification contained an average of 10.6 informal test step descriptions (*min*=5, *max*=21, *IQR*=5.25).

We cleaned our benchmark as follows. We fixed inconsistencies between the informal test case specification and the corresponding test script. For example, the test scripts sometimes contained executable test instructions not described in the informal test case specification or did not contain executable test instructions implementing part of the described behavior. We fixed the convention. Our industrial partner follows a convention of providing the value for named constants when they are mentioned in their informal test step description, but these values were not always present, relying on the fact that the engineers implementing the test scripts could look them up. We fixed text formatting issues. For example, there

```
1  # Context
2  * KeyFobLockUnlockCmd.EnableOverride : ModelVariable
3  * KeyFobLockUnlockCmd.OverrideValue : ModelVariable
4  * KeyPresence.EnableOverride : ModelVariable
5  * KeyPresence.OverrideValue : ModelVariable
6  * KEY_PRESENCE : Parameter
7
8  # Test Step Description
9  * Summary: Set key presence
10 * Details: KeyPresence = $KEY_PRESENCE
11 * Tag: Precondition
12
13 # Test Instructions
```

```
1  * WRITE(KeyPresence.EnableOverride, 1)
2  * WRITE(KeyPresence.OverrideValue, KEY_PRESENCE)
```

Figure 6: Example format used for the few-shot examples used to train the AI model

were parameters missing dollar sign prefix and unclosed parentheses. Finally, we removed pairs with the same informal test step description and executable test instruction.

After processing, our benchmark contained 200 unique pairs of informal test step descriptions and the corresponding executable test instructions. The test scripts contained an average of 4.4 executable test instructions (*min*=1, *max*=21, *IQR*=4).

5.2 Methodology

We trained our model by including four examples for the few-shot learning instructions generator component. Across these examples, each test instruction is used at least once. Figure 4 illustrates the format of the examples. The few-shot examples are formatted to label the context, the informal specification, and test instructions using markdown section header syntax (lines beginning with hash characters). Each entry in the retrieved context is included in the prompt, and the type of context is indicated (i.e. whether it is a model variable, etc.) For subroutines, the parameters are listed in parentheses following the name. The final element in the prompt is the informal test case specification to be considered by the instructions generator component and its context. These inputs are formatted similarly to the examples, but the text ends with the text *# Test Instructions*. The generative model is expected to generate the corresponding test instructions.

To answer our research questions, we executed text-generation models on a secured machine to ensure the confidentiality of our industrial data (Requirement R3). We considered LLM from three families of open lightweight text generation models: Google's Gemma family [48], Meta's Llama2 family [49], and MistralAI's Mistral family [25] since we could not use larger proprietary models. From

these families, we evaluated base models, i.e., gemma-7b (g-7b), Llama-2-7b (L-2-7b), Mistral-7B-v0.3 (M-7B-v03), variants tuned for instruction following, i.e., gemma-7b-it (g-7b-it), Llama-2-7b-chat (L-2-7b-c), Mistral-7B-Instruct-v0.3 (M-7B-I-v03), variants tuned for generating code, i.e., codegemma-7b (cg-7b), CodeLlama-7b (CL-7b), and variants tuned for both, i.e., codegemma-7b-it (cg-7b-it), CodeLlama-7b-Instruct (CL-7b-I). We selected the largest model we could execute using our on-site hardware for each model series (two NVIDIA GeForce RTX 3090 graphics cards, totalling 48 GB VRAM [38]). This variety of models is used to provide a comprehensive answer to RQ1 and RQ3. To answer RQ2, we evaluated different *prompt orders*: Whether the context or the informal specification comes first in the prompt. The labels "Cont,Inf" and "Inf,Cont" from Table 2 and Table 3 respectively mark rows that refer to experiments where the context precedes (or follows) the informal specification. To answer RQ4, we evaluated different *context order*, i.e., the ordering (alphabetically vs shuffled) of items in the context list. The labels "Alph" and "Shuf" respectively mark rows that refer to experiments where the entries within the context are ordered alphabetically or shuffled.

A *configuration* consists of a prompt order, context order, and generative model. We conducted two experiments:

Exp 1: We considered a proxy for the retrieved context that mimic the outputs of an ideal retrieval process for each informal specification and contains only the elements referenced in the corresponding test packages.

Exp 2: We mimic the outputs of an imprecise retrieval process which includes all context entries referenced in the test scripts and some unrelated entries (9.8 times as much unnecessary context as necessary context).

Table 2 and Table 3 present the results of the two experiments.

5.3 Evaluation Metrics

To evaluate the effectiveness of our solution we compare the sequence of *test instructions* automatically generated by our approach (the *predictions*) and manually defined by engineers (the *references*). To measure their similarity we considered three metrics: *exact match rate*, *line match rate*, and *ChrF*. These metrics are complementary: The first two metrics consider predictions as successful or unsuccessful depending on whether or not they *exactly* match the reference, the last metric considers the *similarity* between a prediction and the reference. These metrics are summarized as follows:

Exact Match Rate (EMR): measures the proportion of predictions that exactly match the associated reference.

Line Match Rate (LMR): measures the proportion of predictions that contain exactly the same lines as the associated reference, ignoring their order.

ChrF: The ChrF score [41] measures the similarity between a prediction and a reference by splitting each line into substrings and calculating an F_β score measuring the proportion of substrings in the prediction that have a counterpart in the reference, and visa-versa. The metric is parameterized by two values: the recall weighting factor (β) and the maximum substring length (N). We use $\beta = 2$ and $N = 6$ based on recent research results [18, 42].

The exact match rate provides a *pessimistic bound* on the effectiveness of our solution: it captures predictions that exactly match the reference. However, other functionally equivalent predictions (e.g., by reordering some instructions) may exist. The line match rate provides an *optimistic bound* on the effectiveness of our solution: It estimates the effectiveness assuming that reordering the instructions does not affect their functionality. While this assumption may be true in some cases, it is generally false. The ChrF provides a *complementary measure* that estimates the overall similarity between predictions and references: It reflects the human judgment of similarity between code snippets [18], but does not imply its correctness [18, 40]. Our metric is the mean ChrF score.

Table 2 and Table 3 report the EMR, LMR, and ChrF scores for our experiments: The highest scores for each prompt variant in bold; the highest scores across all prompt variants are underlined.

5.4 RQ1 — Correctness

To answer RQ1, we analyze the results of the experiments from Exp 1 (Table 2), i.e., the one using an ideal context proxy.

The mean EMR is 30.6 (*min*=8, *max*=49.5, *IQR*=10.125) showing that the best configuration generates predictions that exactly match the reference in at most 49.5% of the cases. However, the EMR is a pessimistic bound for the solution's effectiveness: Other predictions that are functionally equivalent (e.g., by reordering some of the instructions) may exists.

The mean LMR is 41.8 (*min*=10.5, *max*=64.5, *IQR*=13.5) showing that the best configuration generates predictions that exactly match the reference in 64.5% of the cases. However, the LMR is an optimistic bound for the solution's effectiveness: Reordering the lines may generate not functionally equivalent programs.

The mean ChrF score is 85.2 (*min*=69.4, *max*=94, *IQR*=9.25) showing that the best configuration generates predictions that are for 94% similar to the one defined by humans. This result shows that the test scripts are highly similar to the ones defined by humans (even if not perfectly functionally equivalent).

On our benchmark, we can conclude that few-shot learning (properly configured) can produce test scripts that correctly implement their informal specification for at least 49.5% (and at most 64.5%) of the cases. Despite not being correct, the ChrF score shows that the test scripts are highly similar to the ones defined by humans and, therefore, they can provide draft solutions engineers can use to develop their test scripts.

RQ1 — Correctness

Few-shot learning (properly configured) can produce test scripts that correctly implement their informal specification between 49.5% and 64.5% of the cases. However, when not correct, the test scripts are highly similar to the ones defined by humans and might be a valid starting point for the implementation of the test cases.

5.5 RQ2 — Prompt Design

We considered the results from Table 2 and Table 3 and analyzed how the prompt design impacts the effectiveness of our solution.

We compared the results related to configurations that differ in the prompt and context ordering. For example, the results of the two configurations from Table 2 identified with a gray background

Table 2: Prompt Order (Prompt), Context Order (Cont), generative model (LLM), Mean ChrF (ChrF), EMR, and LMR for Exp 1.

Prompt	Cont	LLM	ChrF	EMR	LMR	Prompt	Cont	LLM	ChrF	EMR	LMR
Cont, Inf	Alph	g-7b	89.7	37.5	49.0	Inf, Cont	Alph	g-7b	91.8	43.0	57.5
Cont, Inf	Alph	g-7b-it	73.8	26.5	34.0	Inf, Cont	Alph	g-7b-it	78.8	28.5	36.0
Cont, Inf	Alph	cg-7b	**93.4**	**49.5**	**64.5**	Inf, Cont	Alph	cg-7b	**93.1**	**47.0**	**59.5**
Cont, Inf	Alph	cg-7b-it	88.9	36.5	49.5	Inf, Cont	Alph	cg-7b-it	86.5	35.0	48.5
Cont, Inf	Alph	L-2-7b	80.2	18.0	23.5	Inf, Cont	Alph	L-2-7b	81.1	18.5	24.5
Cont, Inf	Alph	L-2-7b-c	73.3	26.5	17.5	Inf, Cont	Alph	L-2-7b-c	74.3	12.5	16.0
Cont, Inf	Alph	CL-7b	86.3	37.0	50.0	Inf, Cont	Alph	CL-7b	86.2	36.0	46.0
Cont, Inf	Alph	CL-7b-I	86.3	38.5	53.0	Inf, Cont	Alph	CL-7b-I	86.9	38.5	49.0
Cont, Inf	Alph	M-7B-v03	89.6	37.0	49.0	Inf, Cont	Alph	M-7B-v03	90.8	36.5	48.5
Cont, Inf	Alph	M-7B-I-v03	87.0	30.5	40.0	Inf, Cont	Alph	M-7B-I-v03	87.4	33.0	42.5
Cont, Inf	Shuf	g-7b	89.7	34.5	46.0	Inf, Cont	Shuf	g-7b	92.4	36.5	56.0
Cont, Inf	Shuf	g-7b-it	69.4	20.0	29.5	Inf, Cont	Shuf	g-7b-it	79.1	24.5	37.5
Cont, Inf	Shuf	cg-7b	**93.4**	**44.0**	**63.0**	Inf, Cont	Shuf	cg-7b	**94.0**	**45.0**	**64.5**
Cont, Inf	Shuf	cg-7b-it	87.7	32.5	47.0	Inf, Cont	Shuf	cg-7b-it	87.5	32.5	48.5
Cont, Inf	Shuf	L-2-7b	80.3	18.0	24.5	Inf, Cont	Shuf	L-2-7b	79.9	15.5	21.0
Cont, Inf	Shuf	L-2-7b-c	76.5	8.5	14.5	Inf, Cont	Shuf	L-2-7b-c	74.7	8.0	10.5
Cont, Inf	Shuf	CL-7b	86.3	29.5	44.0	Inf, Cont	Shuf	CL-7b	86.5	26.5	40.5
Cont, Inf	Shuf	CL-7b-I	86.0	32.0	46.0	Inf, Cont	Shuf	CL-7b-I	86.9	30.0	45.0
Cont, Inf	Shuf	M-7B-v03	88.4	32.0	47.0	Inf, Cont	Shuf	M-7B-v03	89.5	30.0	45.5
Cont, Inf	Shuf	M-7B-I-v03	86.1	29.0	40.0	Inf, Cont	Shuf	M-7B-I-v03	88.3	28.0	45.0

Table 3: Prompt Order (Prompt), Context Order (Cont), generative model (LLM), Mean ChrF (ChrF), EMR, and LMR for Exp 2.

Prompt	Cont	LLM	ChrF	EMR	LMR	Prompt	Cont	LLM	ChrF	EMR	LMR
Cont, Inf	Alph	g-7b	30.9	0.5	1.5	Inf, Cont	Alph	g-7b	29.5	0.0	0.0
Cont, Inf	Alph	g-7b-it	**59.2**	3.5	4.5	Inf, Cont	Alph	g-7b-it	**49.6**	0.0	0.0
Cont, Inf	Alph	cg-7b	48.7	0.5	0.5	Inf, Cont	Alph	cg-7b	43.4	1.0	1.5
Cont, Inf	Alph	cg-7b-it	53.1	**5.5**	**5.5**	Inf, Cont	Alph	cg-7b-it	46.7	**3.0**	**3.0**
Cont, Inf	Alph	L-2-7b	41.2	0.5	0.5	Inf, Cont	Alph	L-2-7b	31.5	0.5	0.5
Cont, Inf	Alph	L-2-7b-c	40.9	0.0	0.0	Inf, Cont	Alph	L-2-7b-c	40.0	1.0	1.0
Cont, Inf	Alph	CL-7b	47.1	4.0	4.0	Inf, Cont	Alph	CL-7b	41.5	2.5	2.5
Cont, Inf	Alph	CL-7b-I	50.4	3.5	4.0	Inf, Cont	Alph	CL-7b-I	43.3	2.5	2.5
Cont, Inf	Alph	M-7B-v03	43.9	0.0	0.0	Inf, Cont	Alph	M-7B-v03	40.8	**3.0**	**3.0**
Cont, Inf	Alph	M-7B-I-v03	44.0	0.5	2.5	Inf, Cont	Alph	M-7B-I-v03	41.6	0.0	0.0
Cont, Inf	Shuf	g-7b	30.9	0.5	2.0	Inf, Cont	Shuf	g-7b	27.7	0.0	1.0
Cont, Inf	Shuf	g-7b-it	**56.8**	1.5	2.5	Inf, Cont	Shuf	g-7b-it	**50.1**	0.5	3.0
Cont, Inf	Shuf	cg-7b	53.1	**6.5**	**9.5**	Inf, Cont	Shuf	cg-7b	46.4	**3.5**	**4.0**
Cont, Inf	Shuf	cg-7b-it	53.3	**6.5**	8.5	Inf, Cont	Shuf	cg-7b-it	44.2	1.5	1.5
Cont, Inf	Shuf	L-2-7b	34.7	1.0	1.0	Inf, Cont	Shuf	L-2-7b	28.6	0.0	0.0
Cont, Inf	Shuf	L-2-7b-c	41.0	0.5	0.5	Inf, Cont	Shuf	L-2-7b-c	35.4	0.0	0.0
Cont, Inf	Shuf	CL-7b	41.0	1.5	3.0	Inf, Cont	Shuf	CL-7b	36.6	0.5	0.5
Cont, Inf	Shuf	CL-7b-I	43.1	1.5	2.5	Inf, Cont	Shuf	CL-7b-I	38.7	1.0	1.0
Cont, Inf	Shuf	M-7B-v03	37.8	1.5	2.0	Inf, Cont	Shuf	M-7B-v03	32.8	1.0	1.5
Cont, Inf	Shuf	M-7B-I-v03	42.9	1.5	3.0	Inf, Cont	Shuf	M-7B-I-v03	36.6	0.5	0.5

are compared since they only differ in context ordering (Alphabetical vs Shuffle). Therefore, we performed 40 comparisons in total (20 comparisons for each experiment). A significant difference in measured values for a metric across a pair of configurations would support the hypothesis that the prompt variants have an effect on the quality of the generated text, with the prompt variant with the higher metric value performing better. We considered a variant to

Table 4: Impact of prompt variations on each metric.

Exp	Comparison	Mean ChrF	EMR	LMR
Exp 1	"Cont,Inf" vs "Inf,Cont"	0:18:2	1:18:1	1:15:4
	Alphabetical vs Shuffled	0:20:0	11:9:0	4:15:1
Exp 2	"Cont,Inf" vs "Inf,Cont"	13:7:0	1:19:0	2:18:0
	Alphabetical vs Shuffled	6:14:0	0:19:1	0:19:1

Table 5: Impact of LLM variants on each metric.

Exp	Comparison	Mean ChrF	EMR	LMR
Exp 1	Code Generation Tuning	0:4:12	0:1:15	0:1:15
	Instruction Tuning	10:10:0	12:7:1	15:5:0
Exp 2	Code Generation Tuning	2:5:9	0:14:2	0:14:2
	Instruction Tuning	0:12:8	0:19:1	0:19:1

perform better if a metric measures an improvement of at least five points[1] in more than half of the analyzed pairs of configurations.

Table 4 reports the results of our analysis. The upper and lower parts of the table show the results for Exp 1 and 2. The rows in the table show, for each metric, the proportion of configurations that measured a preference for the left-hand variant (the leftmost number), a preference for the right-hand variant (the rightmost number), or no preference (the center number).

For Exp 1, the configurations using prompts that ordered context alphabetically yielded higher EMR scores than their counterparts with shuffled context in 11 out of the 20 cases, with the remaining 9 pairs measuring no significant difference. For Exp 2, the configurations using prompts that presented the context before the informal specification (that is, context-first prompts) measured significantly higher ChrF scores than their counterparts in 13 out of the 20 pairs of configurations, with the remaining 7 pairs measuring no significant difference. For the remaining cases, the results show that the prompt order and the context order do not significantly affect the effectiveness of the proposed solution. Therefore, despite the fact that in the majority of the cases the prompt order and context order do not affect the effectiveness of the solution, our results suggest to order the context entries alphabetically and to put the context before the informal specification.

RQ2 — Prompt Design

Even though in most cases the prompt order and context order do not affect the effectiveness of our solution, our results suggest ordering the context entries alphabetically and putting the context before the informal specification.

5.6 RQ3 — Effects of LLM Variants

To determine the impact of LLM variants on the effectiveness of our solution, we compare the performance of models fine-tuned for *code-generation* and *instruction-following* against the corresponding models that were not. We use the same paired-comparison strategy used to measure the impact of the prompt variants in Section 5.5.

To determine the effects of code-tuning, the analysis compares pairs of configurations using the same prompt and model family, but differ only in whether the model variant was tuned for code-generation (e.g., gemma-7b vs. codegemma-7b or gemma-7b-it vs. codegemma-7b-it). Since no code-generation variants of the Mistral models were evaluated, the analysis of code-generation variants only included experiments using models from the Gemma and Llama2 families. Therefore, we performed 32 comparisons in total (16 comparisons for each experiment).

To determine the effects of instruction-following, the analysis compares pairs of configurations using the same prompt and model family, but differ only in whether the model variant was tuned for instruction-following. Therefore, we performed 40 comparisons in total (20 comparisons for each experiment).

Table 5 reports the results of our analyses. The comparisons column in Table 5 contain the number of experiments which measured a preference for the base model (leftmost number), the tuned model (rightmost number), or no preference at all (the center number).

Code Generation. For Exp 1, the LLM fine-tuned for code generation outperformed the not code-tuned models for most of the cases. For Exp 2, the code-generation variants did generate ChrF scores that were significantly better than their counterparts for 9 out of 16 comparisons, but 2 of those 16 pairs supported a significant preference for a model variant that was *not* tuned for code generation. Furthermore, there is no consistent significant preference according to EMR and LMR measures.

Instruction Following. For Exp 1, the LLM tuned for instruction following performed significantly worse than their counterparts in most of the experiments: the ChrF, EMR and LMR measured a significant decrease in performance (in respectively 10 out of 20, 12 out of 20, and 15 out of 20 cases). For Exp 2, most of the comparisons showed that instruction tuning did not make a significant impact. Only one pair of experiments measured a significant difference in EMR and LMR values, but the ChrF scores showed a preference for instruction-tuned models in 8 out of 20 comparisons (the remaining comparisons showed no preference). The ChrF scores indicate that some effect may be present, but that there is insufficient evidence to support a generalizable result since most experiment pairs (i.e. 12 out of 20) could not establish a statistically significant alternative to the null hypothesis of their being no generalizable effect.

RQ3 — Effects of LLM Variants

Our results suggest that LLMs tuned for code generation outperform their counterparts. Instruction tuning decreased the effectiveness when the ideal context was used, and did not provide improvements for the unfiltered context.

5.7 RQ4 — Effects of Context Accuracy

We analyze if unnecessary context in the prompt negatively impacts the generation results, and if the results degrade as the amount of unnecessary context increases. Understanding if and when this

[1]This threshold correlates with statistical significance (with significance level $alpha$ = 0.05) for code generation [18].

Table 6: Impact of unused context on ChrF, EMR, and LMR.

Unused Context	ChrF	EMR	LMR
0 (Ideal)	93.4	49.5	64.5
1	87.8	20.5	24.5
3	84.4	21.0	25.5
33%	88.2	28.5	33.0
50%	84.1	25.5	28.0
66%	77.4	20.5	24.5
Unfiltered	69.0	12.0	12.5

degradation occurs is important for future investigations as it informs the parameters for deciding whether a particular context retrieval approach can be successful.

We compared the results of the ideal proxy experiments (Exp 1) to their counterparts in the unfiltered context proxy experiments (Exp 2). For Exp 2, the average EMR is 1.6 (min=0.0, max=6.5, IQR=2.0), the average LMR is 2.1 (min=0.0, max=9.5, IQR=2.5, and the average ChrF is mean=42.0 (min=27.7, max=59.2, IQR=10.2). Compared to the results from Exp 1 from Section 5.4, the decrements of the average EMR, LMR, and ChrF are respectively ↓28.7 (min=↓8.0, max=↓49.0, IQR=↓12.5), ↓39.3 (min=↓10.5, max=↓64.0, IQR=↓15.0) and ↓43.2 (min=↓12.6, max=↓64.7, IQR=↓11.5).

The results of these experiments indicate a significant impact on performance from the unnecessary entries present in the context (recall that on average the unnecessary entries outnumber the necessary entries by a factor of 9.8). To determine the amount of unnecessary context that can be tolerated, we conducted an additional series of experiments using contexts containing varying amounts of unnecessary context. We run experiments using context proxies containing 1 or 3 unnecessary context entries, and a particular ratio of unnecessary context entries (33%, 50%, or 66%). For these experiments we updated the few-shot example contexts to contain the same number/proportion of unnecessary entries as the input context. We run these experiments by considering the prompts with the context first, order the context entries alphabetically, and evaluate against codegemma-7b (cg-7b) since this configuration was generally preferable compared to the others (see sections 5.4, 5.5, and 5.6). Table 6 reports the results of these experiments.

The results show a fairly rapid decrease in performance as unnecessary entries were added. Each metric measured a decrease in the quality of the results after only a single entry of unnecessary context was introduced, with EMR and LMR falling the most; adding a total of three unnecessary context entries yielded results similar to the experiment where only one entry was added. When 33% of the context was unnecessary, each metric measured better performance than adding a static number of entries, with performance falling as a higher proportion of unnecessary entries were added.

RQ4 — Effects of Context Accuracy

Unnecessary and inaccurate context significantly reduces the effectiveness of the test generation.

6 DISCUSSION AND THREATS TO VALIDITY

We report several *lessons learned* from our evaluation.

Lesson L1. Our solution yielded many ready-to-use test instructions (RQ1). However, the scripts produced using generative AI tools should be reviewed by engineers due to incorrect test instructions which may lead to erroneous test cases.

Lesson L2. Our results (RQ1, RQ2, RQ3, and RQ4) show that it is possible to adapt a pre-trained text generation model to perform the test case generation task without collecting data for fine-tuning by using few-shot learning (i.e., enriching the prompt with a few examples that can be used to guide the text generation model).

Lesson L3. We evaluated our solution using an in-house platform, i.e., without sending data to third-party cloud-hosted generative AI platforms. Setting up this platform required significant effort. For example, we had to assess the publicly available models to understand which fitted the computational capabilities of our hardware (i.e., we could not use the most powerful models available).

Lesson L4. Despite our results confirming that the selection of the prompt design (RQ2), LLM variant (RQ3), and context accuracy (RQ4) affect the effectiveness of our procedure, the results are still affected by the non-deterministic nature of the generative models. For example, for RQ4, shifting from 3 to 33% of unused context increased the effectiveness of our solution. We do not have a precise explanation for this result: It might be that for models with unnecessary context, adding more unnecessary context makes the model realize that it should be ignoring some of it, or it can be caused by some noise generated by the non-deterministic nature of the generative AI.

The automated generation of test cases is a research problem *relevant to industry* [1, 3, 20, 60]. Our research has been conducted in collaboration with a large automotive company to assess the potentiality of generative AI for test case generation in an industrial context. Using Rational Quality Manager (Assumption A1) and ecu.test packages (Assumption A2) were assumptions defined as part of this collaboration. Our industrial partner required us to maintain existing test cases confidential (Requirement R3). For this reason, while our experiments are replicable and we have a complete replication package, we can not share it publicly. However, to foster experiment replication and open science policies, we precisely indicated the versions of the LLM models we considered in this work.

Our results significantly improve the *state of practice* by (a) confirming that generative AI can be used to formulate initial test case specifications that can be improved by engineers [26], (b) presenting results that are less optimistic than the one reported in previous studies (e.g., [58]) and unlikely support the hypothesis that LLMs outperform skilled testing engineers, and (c) confirming that, currently, we cannot rely on LLMs to replace manual testing for detecting software errors [32]. Our results provide a valuable resource for other engineers from the automotive domain looking into the effectiveness of generative AI for test case generation. They are also valuable to other CPS engineers and the research community that needs precise data about the effectiveness of LLM.

Threats to Validity. The dataset we used to evaluate our solution threatens the *external validity* of our results. We can not claim that our results *generalize to other testing scenarios and domains* (i.e., not

automotive): The peculiarities of each CPS domain and the tools used in each domain likely affect the effectiveness of the generative AI platforms. However, our focus is the automotive domain. Within this domain, the work, approach, and lessons learned apply to practitioners outside our study group.

The benchmark cleaning and the metrics used for our evaluation threaten the *internal validity* of our results. For the benchmark cleaning, we worked with our industrial partner to determine their expectations for (and inform their guidelines on) how the benchmark should be cleaned, and which ecu.test features would be commonly used, ensuring their practices are aligned with the fixes we made and ensure that our dataset was representative. For the evaluation metrics, using multiple metrics mitigates the limitations of each single metric providing complementary information about the effectiveness of our solution.

Finally, our results can be improved by considering techniques such as tuning generation hyperparameters [45] and using model-specific features such as chat templates for instruction following models [19]. The results from our study provide a suitable baseline to conduct such activities as future work.

7 RELATED WORK

Test case and code generation approaches, and using LLMs for semantic parsing are related to our study.

Test Case Generation Approaches. Test generation derives test procedures from various sources, including functional requirements [39], use cases [57], behavioral models [61], and source code of the software under test [47, 52]. For CPS, there is often an abstraction gap between the environment in which the test procedures are developed and executed which is often manually addressed (e.g. [57, 61]). Recently generative AI and LLM have been investigated as instruments to generate test cases (e.g., [6, 7, 11, 13, 26, 28, 29, 34, 59]). Wang et al. [57] use NLP techniques such as Parts of Speech (POS) labelling [44] in their workflow to bridge the abstraction gap. Their process consumes use cases specified in a restricted natural language (RUCM [24]) and requires manually created domain models and mapping tables to relate the abstract specification to the concrete test platform. Other approaches (e.g. [47, 52]) require less manual effort, leveraging large language models to generate unit tests directly from source code. Lin et al. [33] use LLM agents to solve a programming problem by performing the individual tasks that would be performed by different roles in a software development team (including a "tester" role).

Code Generation Approaches. The gap in abstraction between specification and implementation has been observed in the application of LLMs to the task of project-level code generation [5, 21, 31]. Studies have observed that LLMs must be provided enough *project-level context* to generate code that executes against the desired methods, classes, etc. in a pre-existing project. RAG-based approaches [5, 35, 46, 62] address this problem. The RepoCoder approach [62] uses an existing semantic retriever to find passages from the project repository and performs iterative retrieval and generation steps, using the generated results as input to the retriever to refine the retrieved context. The CoCoGen approach [5] extends the iterative retrieval concept by incorporating compiler feedback and structured SQL queries into the approach of [62]. Another study [46] proposes employing an extended RAG model (DRAG) that dynamically extends the vocabulary of the text generator with tokens for the entities fetched by the retriever rather than providing the retrieved context as part of the input to the generator model. Although the results from [46] showed that the DRAG approach could improve the generation results, preparing a DRAG model requires training activities and cannot use pre-trained models directly off the shelf. The capabilities of these RAG-based approaches were studied using various pre-trained models and benchmarks. They demonstrated a significant performance improvement when the project-level context is available to the text generation model.

LLMs for Semantic Parsing. Our work also relates to studies that use LLMs to generate outputs that conform to the grammar of DSLs. The task of converting natural language to a machine-readable format (i.e., DSL) is known as semantic parsing. Several approaches leverage the definition of the target grammar, either by including it in the prompt [36] or by constraining the LLM to ensure the results are well formed (e.g. [4, 56]). One study [2] shows that LLMS can generalize from few-shot examples and generate machine-readable outputs without a grammar definition.

Unlike existing studies, we consider the test case generation problem within the automotive domain, we performed our study in collaboration with a large automotive company, and we used industrial test cases to assess the effectiveness of our solution.

8 CONCLUSIONS

This paper explores the use of generative AI to automatically convert informal test case specifications into executable test scripts in the automotive domain. By integrating large language models with few-shot learning and retrieval-augmented generation, the proposed approach highlighted the potential of generative AI to support industrial software testing processes. Our solution assumes test case specifications defined in Rational Quality Manager and test scripts specified in ecu.test. We evaluated our solution by considering an industrial benchmark. Our results show that generative AI can produce correct or near-correct test cases in many scenarios, the quality of results depends significantly on prompt design, large language model selection, and the accuracy of context retrieval. Our study underscores the need for human oversight to address subtle errors in logic sequencing and value assignments, ensuring functional correctness. Future research should prioritize improving retrieval mechanisms, expanding dataset diversity, and exploring hybrid human-AI workflows to enhance generative AI's scalability, reliability, and larger applicability in industrial settings.

ACKNOWLEDGMENTS

This work was partly supported by the European Union - Next Generation EU, "Sustainable Mobility Center (Centro Nazionale per la Mobilità Sostenibile - CNMS)", M4C2 - Investment 1.4, Project Code CN_00000023, by projects SERICS (PE00000014) under the NRRP MUR program, and GLACIATION (101070141).

REFERENCES

[1] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23 (2018), 1959–1981.

[2] Nils Baumann, Juan Sebastian Diaz, Judith Michael, Lukas Netz, Haron Nqiri, Jan Reimer, and Bernhard Rumpe. 2024. Combining retrieval-augmented generation and few-shot learning for model synthesis of uncommon DSLs. In *Modellierung 2024 Satellite Events*. Gesellschaft für Informatik eV, 10–18420.

[3] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103.

[4] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. *arXiv preprint arXiv:2403.06988* (2024).

[5] Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Xuanhua Shi, and Hai Jin. 2024. Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback. *arXiv preprint arXiv:2403.16792* (2024).

[6] Mohamed Boukhlif, Nassim Kharmoum, and Mohamed Hanine. 2024. LLMs for Intelligent Software Testing: A Comparative Study. In *International Conference on Networking, Intelligent Systems and Security (NISS '24)*. ACM, Article 42, 8 pages.

[7] Mohamed Boukhlif, Nassim Kharmoum, Mohamed Hanine, Mohcine Kodad, and Souad Najoua Lagmiri. 2024. Towards an Intelligent Test Case Generation Framework Using LLMs and Prompt Engineering. In *International Conference on Smart Medical, IoT & Artificial Intelligence*. Springer, 24–31.

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

[9] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* 22, 3 (2023), 781–793. https://doi.org/10.1007/S10270-023-01105-5

[10] Boqi Chen, Kua Chen, Shabnam Hassani, Yujing Yang, Daniel Amyot, Lysanne Lessard, Gunter Mussbacher, Mehrdad Sabetzadeh, and Dániel Varró. 2023. On the use of GPT-4 for creating goal models: an exploratory study. In *International Requirements Engineering Conference Workshops (REW)*. IEEE, 262–271.

[11] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the International Conference on the Foundations of Software Engineering (FSE)*. ACM, 572–576.

[12] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*. Springer, 383–396.

[13] Yifei Deng, Renzhi Chen, Chao Xiao, Zhijie Yang, Yuanfeng Luo, Jingyue Zhao, Na Li, Zhong Wan, Yongbao Ai, Huadong Dai, and Lei Wang. 2024. LLM - TG: Towards Automated Test Case Generation for Processors Using Large Language Models. In *IEEE International Conference on Computer Design (ICCD)*. 389–396.

[14] Simon Diemert and Jens H Weber. 2023. Can large language models assist in hazard analysis?. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 410–422.

[15] dSpace. 2024. Writing High-Quality Test Scripts: Taming the Chaos. https://www.lhpes.com/blog/writing-high-quality-test-scripts-taming-the-chaos.

[16] dSpace. 2025. Compound Tests Volkswagen: Virtualizing ECU integration tests in response to the rapidly increasing complexity in software-defined vehicles. https://www.dspace.com/en/pub/home/applicationfields/stories/vw-compound-tests.cfm.

[17] dSpace. 2025. SIL for Early Validation (Stellantis/FCA accelerates software testing with agile practices and virtualization). https://www.dspace.com/en/pub/home/applicationfields/stories/stellantis-fca-sil-validation.cfm.

[18] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software* 203 (2023), 111741. https://doi.org/10.1016/J.JSS.2023.111741

[19] Hugging Face. [n. d.]. Chat Templates. https://huggingface.co/docs/transformers/v4.48.0/en/chat_templating. Accessed: 2025-01-15.

[20] Vahid Garousi, Michael Felderer, Marco Kuhrmann, and Kadir Herkiloglu. 2017. What industry wants from academia in software testing?: Hearing practitioners' opinions. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Emilia Mendes, Steve Counsell, and Kai Petersen (Eds.). ACM, 65–69. https://doi.org/10.1145/3084226.3084264

[21] Sandra Greiner, Noah Bühlmann, Manuel Ohrndorf, Christos Tsigkanos, Oscar Nierstrasz, and Timo Kehrer. 2024. Automated Generation of Code Contracts: Generative AI to the Rescue?. In *SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24)*. ACM, 1–14.

[22] IBM. 2023. IBM. https://www.ibm.com.

[23] IBM. 2023. Rational Quality Manager (RQM). https://www.ibm.com/docs/en/etm/4.0.6.

[24] Sumit Jain and Mohsin Sheikh. 2014. RUCM: A Measurement Model for detecting the Most Suitable Code Component from Object Oriented Repository. *International Journal of Computer Applications* 103, 2 (2014).

[25] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[26] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, et al. 2024. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the International Conference on the Foundations of Software Engineering*. ACM, 492–496.

[27] Katharina Juhnke, Matthias Tichy, and Frank Houdek. 2021. Challenges concerning test case specifications in automotive software testing: assessment of frequency and criticality. *Software Quality Journal* 29 (2021), 39–100. https://doi.org/10.1007/S11219-020-09523-0

[28] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2312–2323.

[29] Heiko Koziolek, Virendra Ashiwal, Soumyadip Bandyopadhyay, and Chandrika K R. 2024. Automated Control Logic Test Case Generation using Large Language Models. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–8.

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[31] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2404.00599* (2024).

[32] Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W Eric Wong. 2025. Evaluating large language models for software testing. *Computer Standards & Interfaces* 93 (2025), 103942.

[33] Feng Lin, Dong Jae Kim, et al. 2024. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852* (2024).

[34] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *IEEE/ACM International Conference on Software Engineering*. 1–13.

[35] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[36] My M. Mosthaf and Andrzej Wasowski. 2024. From a Natural to a Formal Language with DSL Assistant. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 541–549.

[37] Claudio Menghi, Torin Viger, Alessio Di Sandro, Chris Rees, Jeff Joyce, and Marsha Chechik. 2023. Assurance case development as data: A manifesto. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 135–139.

[38] NVIDIA. 2025. GeForce RTX 3090 Family: Specs. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti/#specs. Accessed: 2025-01-15.

[39] Ana CR Paiva, Daniel Maciel, and Alberto Rodrigues da Silva. 2020. From requirements to automated acceptance tests with the RSL language. In *Evaluation of Novel Approaches to Software Engineering (ENASE)*. Springer, 39–57.

[40] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. *arXiv preprint arXiv:2406.12655* (2024).

[41] Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *workshop on statistical machine translation*. 392–395.

[42] Maja Popović. 2016. chrF deconstructed: beta parameters and n-gram weights. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*. 499–504.

[43] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *International workshop on automation of software test (AST)*. IEEE, 36–42.

[44] Nihar Ranjan, Kaushal Mundada, Kunal Phaltane, and Saim Ahmad. 2016. A Survey on Techniques in NLP. *International Journal of Computer Applications* 134, 8 (2016), 6–9.

[45] Matthew Renze and Erhan Guven. 2024. The effect of sampling temperature on problem solving in large language models. *arXiv preprint arXiv:2402.05201* (2024).

[46] Anton Shapkin, Denis Litvinov, and Timofey Bryksin. 2023. Entity-augmented code generation. *arXiv preprint arXiv:2312.08976* (2023).

[47] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *International Conference on Evaluation and Assessment in Software Engineering*. 313–322.

[48] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette

Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).
[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[50] Tracetronic. 2023. Tracetronic. https://www.tracetronic.com/.
[51] Tracetronic. 2025. ecu.test. https://www.tracetronic.com/products/ecu-test/. Accessed: 2025-01-15.
[52] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
[53] Torin Viger, Logan Murphy, Simon Diemert, Claudio Menghi, and Marsha Chechik. 2024. Supporting Change Impact Assessment with LLMs. In *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 203–204.
[54] Torin Viger, Logan Murphy, Simon Diemert, Claudio Menghi, Alessio Di, and Marsha Chechik. 2023. Supporting Assurance Case Development Using Generative AI. In *SAFECOMP 2023, Position Paper*.
[55] Torin Viger, Logan Murphy, Simon Diemert, Claudio Menghi, Jeff Joyce, Alessio Di Sandro, and Marsha Chechik. 2024. AI-Supported Eliminative Argumentation: Practical Experience Generating Defeaters to Increase Confidence in Assurance Cases. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 284–294.
[56] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2024. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems* 36 (2024).
[57] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C Briand. 2020. Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering* 48, 2 (2020), 585–616.
[58] Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024. LLM4Fin: Fully Automating LLM-Powered Test Case Generation for FinTech Software Acceptance Testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1643–1655.
[59] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1127–1139.
[60] Xiao Yu, Lei Liu, Xing Hu, Jacky Keung, Xin Xia, and David Lo. 2024. Practitioners' Expectations on Automated Test Generation. In *SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 1618–1630.
[61] Muhammad Nouman Zafar, Wasif Afzal, Eduard Paul Enoiu, Athanasios Stratis, and Ola Sellin. 2021. A model-based test script generation framework for embedded software. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 192–198.
[62] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).