# Acceleration of Automotive Software Development by Retrieval Augmented Integration Test Script Generation

Masashi Mizoguchi
*Research and Development Group*
*Hitachi, Ltd.*
Hitachi, Japan
masashi.mizoguchi.re@hitachi.com

Kentaro Yoshimura
*Research and Development Group*
*Hitachi, Ltd.*
Hitachi, Japan
kentaro.yoshimura.jr@hitachi.com

Keita Nakazawa
*SDV Solution Business Unit*
*Astemo, Ltd.*
Hitachinaka, Japan
keita.nakazawa.yy@hitachiastemo.com

Yasuomi Sato
*SDV Solution Business Unit*
*Astemo, Ltd.*
Hitachinaka, Japan
yasuomi.sato.sb@hitachiastemo.com

Takahiro Iida
*SDV Solution Business Unit*
*Astemo, Ltd.*
Hitachinaka, Japan
takahiro.iida.pf@hitachiastemo.com

Fumio Narisawa
*SDV Solution Business Unit*
*Astemo, Ltd.*
Hitachinaka, Japan
fumio.narisawa.ks@hitachiastemo.com

***Abstract*—Improving the efficiency of software integration testing is a critical challenge in the automotive industry, particularly as Electronic Control Unit (ECU) architectures become increasingly complex. This paper addresses the automation of integration test script generation by leveraging Large Language Models (LLMs) with Retrieval Augmented Generation (RAG). Specifically, we target the phase in which test engineers translate natural language test case specifications into executable scripts for integration test environments containing hardware debug interfaces. To bridge the knowledge gap between LLMs and domain-specific test tool APIs, we construct a task-oriented vector store that incorporates both API manuals and supplemental, workflow-centric information. By combining these with prompts containing code prefixes, our method enables LLMs to generate robust and correct integration test scripts. We evaluated our approach on typical test scenarios reflecting industry practices for multi-core ECUs. While the test cases used were not directly taken from a specific development project, they closely mirror those routinely employed across numerous automotive ECU development initiatives. The proposed method successfully generated executable scripts for all cases and reduced total test execution man-hours by 43% compared to a realistic baseline of manual execution. These results demonstrate the practical benefit of context-enriched LLMs in accelerating specialized software engineering tasks within the automotive domain, and it also identifies remaining challenges in extending automation to broader aspects such as test coverage, maintainability, and seamless process integration.**

***Index Terms*—automotive software, integration test, large language model, retrieval augmented generation, test script**

## I. Introduction

The scale and complexity of automotive software are escalating rapidly due to innovations such as autonomous driving and electrification [1], [2]. Consequently, the efficiency of software development—especially in integration testing—has become a pressing issue for both OEMs and suppliers in the automotive industry.

Automotive software development commonly adopts the V-model, with Automotive SPICE[1] [3] serving as a widely-adopted process assessment framework [4]. In particular, the Software Integration and Integration Test phase (SWE.5) now requires extensive resources and man-hours. In integration testing, test engineers develop test case specifications, construct complex multi-core environments, and create scripts to automate test execution—often using hardware debugger interfaces.

Although test automation infrastructure is generally available, the overhead of specifying and developing test scripts—including hardware- and tool-specific workflows—remains substantial and is becoming a bottleneck. For test cases that are relatively simple or have limited runs, the cost of script development may even outweigh manual execution, a nontrivial consideration in project resource trade-offs.

Large Language Models (LLMs) have shown promise in code and document generation [5], [6], but their application to engineering tasks in the automotive industry faces significant hurdles. Generic LLMs lack detailed knowledge of proprietary APIs, specialized workflows, and the tacit practices of automotive integration test engineers. Retrieval Augmented Generation (RAG) [7] seeks to mitigate this by supplying LLMs with target domain knowledge, but its practical application within regulated, safety-critical industrial contexts is not yet well understood [8]–[11].

**Contributions.** This paper presents a method for automating the generation of integration test scripts from natural language test case specifications by combining LLMs, a RAG-based vector store containing both API documentation and workflow-

[1]Automotive SPICE® is a registered trademark of VDA Verband der Automobilindustrie e.V.

centric information, and task-specific prompts. Our key findings are:

- LLMs alone cannot generate valid test scripts for automotive integration unless supplied with task- and tool-specific information.
- The vector store must combine API documentation and workflow (supplemental) knowledge reflecting practical test engineering processes.
- Using structured prompts with code prefixes, rather than relying on the LLM's prior knowledge, greatly improves robustness and script reliability.
- Evaluation with typical, representative test cases for multi-core ECU integration demonstrates a 43% reduction in test execution man-hours over manual execution.

Importantly, this work highlights the significance of bridging the gap between LLMs and industrial development practice, which is vital for realizing practical LLM utilization in regulated safety-critical domains.

## II. Industrial Context and Motivation

Integration test automation for automotive ECUs is typically carried out using dedicated hardware debuggers (e.g., TRACE32[2] [12]). In practice, these environments require engineers to write Python or proprietary scripts to initialize hardware and evaluate application behavior.

While industrial test frameworks exist, much of the script development work is non-trivial: it requires reading and interpreting both vendor tool API manuals and various unwritten or project-specific conventions (e.g., how to initialize hardware/processors, set breakpoints via addresses or symbols, or take snapshots for evidence).

Integration test specifications are usually documented in natural language by engineers, and translating these into correct, executable scripts forms a labor-intensive task. The inefficiency is exacerbated in recent projects with rapidly changing software on complicated hardware configurations.

It is important to clarify that, industry-wide, *automated execution* of integration tests is the prevailing practice, realized via scripts that operate tester hardware according to pre-defined sequences. However, the *development* of these scripts remains a significant overhead: in some cases, particularly for simple, seldom-executed tests, the effort of developing scripts exceeds that of direct manual execution—hence, manual runs still persist in a part of practical projects.

Our work takes direct aim at this engineering "pain point": reducing the labor and specialized knowledge required to translate test case intent into reliable scripts.

## III. Integration Test Process Overview

Figure 1 summarizes the workflow for SWE.5 as specified by Automotive SPICE. First, the integration sequence of software units is determined based on the basic design specifications. Next, the specifications for the integration testing of each software unit are determined so that the integrated

[2]TRACE32® is a registered trademark of Lauterbach GmbH.

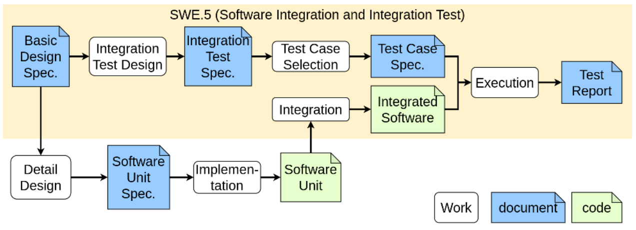

Fig. 1. Summarized workflow in SWE.5.

units meet the basic design specifications. In the next step, start, end, and pass/fail criteria are determined, and test cases are designed based on these criteria and documented in the integration test case specification document. After integrating the software units, the test cases described in the integration test case specification document are executed. For hardware-based verification, both the start/end conditions and criteria for pass/fail must be operationalized into scripts that automate actual board operations (e.g., loading firmware, setting breakpoints, collecting signals, checking memory/register states). Finally, the results of the test case execution are recorded in the test report.

Figure 2 shows our evaluation environment: an RH850 evaluation board [13] with two CPU cores, connected via TRACE32 to a host PC running test scripts and debugger software. The RH850 evaluation board has the target software preinstalled prior to test execution. Since the debugger software shows the values of CPU registers and memories, it is possible to determine whether the integration test case specifications are satisfied.

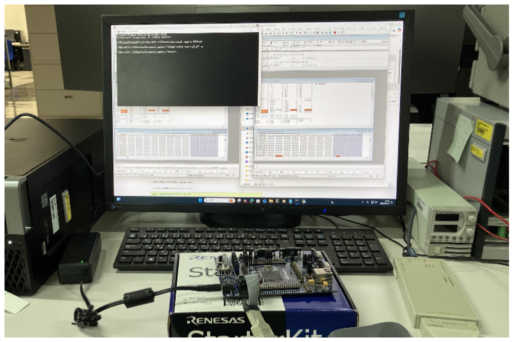

Fig. 2. Integration test environment.

## IV. Test Script Generation Method

Our approach consists of three main components (Fig. 3):

1) Construction of a vector store (for RAG) containing both the API manuals and supplemental workflow-centric documentation.
2) Design of a robust *code prefix*: a code fragment that is prepended to the LLM query and initialized according to the current hardware configuration.

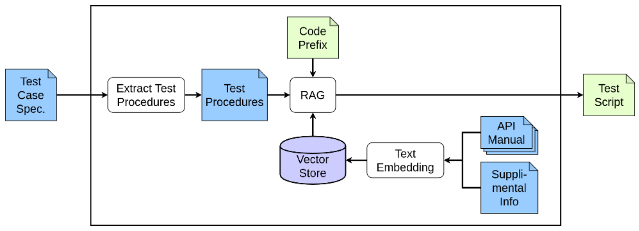

Fig. 3. Summary of the proposed method.

3) Augmenting LLM prompts by retrieving relevant vector store chunks and passing them together with the code prefix for the test script generation.

**Importance of the Code Prefix.** Initialization of the hardware debug interfaces must be performed based on the hardware configuration of the evaluation board. For the test environment shown in Fig. 2, two debug interfaces are to be declared and bound to the two CPU cores on the board. By including such preprocessing in the code prefix, the LLM can focus on converting test cases described in natural language in the specification document into executable scripts. Listing 1 shows an example of the code prefix given to the LLM. The code prefix only depends on the hardware configuration of the microcontrollers. Therefore, the same code prefix can be reused across integration test cases in the same development project.

Listing 1. Code prefix for hardware debug interface initialization
```
import lauterbach.trace32.rcl as t32

debugger_PE0 = t32.connect(node='
    localhost', port=20000, protocol="TCP"
    , timeout=10.0)
debugger_PE1 = t32.connect(node='
    localhost', port=20001, protocol="TCP"
    , timeout=10.0)
```

**Supplemental Workflow Knowledge.** The vector store must combine not only official API documentation but also "tacit" workflow knowledge (see Listing 2). For instance, setting breakpoints by symbol, taking trace screenshots, or starting execution through TRACE32 all require sequences or API idioms not covered directly in manuals.

Listing 2. Examples of supplemental knowledge in the vector store.
```
###Breakpoint
##Set by symbol
#Sample code
symbol = dbg.symbol.query_by_name("
    function_name")
address = symbol.address
bp = dbg.breakpoint.set(address=address)

###Take screenshot.
import pyautogui
img1 = pyautogui.screenshot('filename.png
    ')

###Start execution
dbg.cmd("Go")
```

**Prompt Construction.** A sample high-level prompt is as follows, leveraging the code prefix and supplementary chunks:

> #Command
> You are developing a Python script as a test script developer. Generate a script to realize the operations shown in "Operation to be implemented", definitely add the code by adding it at the end of the code prefix, and output the entire code. When you generate the script, refer to context. Do not use your prior knowledge. You can use your prior knowledge only in case the context is not useful. When you receive the instruction of starting execution, execute PE0 only.
> #Operation to be implemented
> - Set a breakpoint at the beginning of main function.
> #Code prefix
> <Code prefix shown in Listing 1>

The script generated is then executable and board-specific as shown in Listing 3.

Listing 3. The script generated with the vector store containing the API manuals and the supplemental information.
```
import lauterbach.trace32.rcl as t32

debugger_PE0 = t32.connect(node='
    localhost', port=20000, protocol="TCP"
    , timeout=10.0)
debugger_PE1 = t32.connect(node='
    localhost', port=20001, protocol="TCP"
    , timeout=10.0)

# Obtain the address of the main function
symbol = debugger_PE0.symbol.
    query_by_name("main")
address = symbol.address

# Set a breakpoint at the beginning of
    the main function
breakpoint = debugger_PE0.breakpoint.set(
    address=address)
```

**Example of Complex Test Script Generation.** For a more comprehensive test case, the approach correctly wires together multi-core operations, breakpoint settings by symbol/address, and reporting actions. Listing 4 is the generated script with the following prompt:

> #Operation to be implemented
> - Set a breakpoint at the beginning of mainPE0 of PE0.
> - Set a breakpoint at the beginning of mainPE1 of

PE1.
- Set a breakpoint at the following address of PE1: "P:0xFEF00AAA".
- Start execution, take a screenshot after waiting for 5 seconds, save an image as "a.png".
- Set a breakpoint at function "func0" of PE0.
- Start execution, take a screenshot after waiting for 5 seconds, save an image as "b.png".

Listing 4. Generated script for representative test case.

```
import lauterbach.trace32.rcl as t32
import time
import pyautogui

debugger_PE0 = t32.connect(node='
    localhost', port=20000, protocol="TCP"
    , timeout=10.0)
debugger_PE1 = t32.connect(node='
    localhost', port=20001, protocol="TCP"
    , timeout=10.0)

# Set breakpoint at mainPE0 of PE0
symbol_mainPE0 = debugger_PE0.symbol.
    query_by_name("mainPE0")
address_mainPE0 = symbol_mainPE0.address
bp_mainPE0 = debugger_PE0.breakpoint.set(
    address=address_mainPE0)

# Set breakpoint at mainPE1 of PE1
symbol_mainPE1 = debugger_PE1.symbol.
    query_by_name("mainPE1")
address_mainPE1 = symbol_mainPE1.address
bp_mainPE1 = debugger_PE1.breakpoint.set(
    address=address_mainPE1)

# Set breakpoint at address "P:0xFEF00AAA
    " of PE1
address_PE1 = debugger_PE1.address.
    from_string("P:0xFEF00AAA")
bp_address_PE1 = debugger_PE1.breakpoint.
    set(address=address_PE1)

# Start execution for PE0 only
debugger_PE0.cmd("Go")
time.sleep(5)
pyautogui.screenshot('a.png')

# Set breakpoint at function "func0" of
    PE0
symbol_func0 = debugger_PE0.symbol.
    query_by_name("func0")
address_func0 = symbol_func0.address
bp_func0 = debugger_PE0.breakpoint.set(
    address=address_func0)

debugger_PE0.cmd("Go")
time.sleep(5)
pyautogui.screenshot('b.png')
```

## V. Evaluation

We conducted the evaluation by generating scripts for nine representative integration test cases. Table I shows an overview of them and the results of the test script generation. These are typical cases encountered in actual multicore ECU development, covering process launch, synchronization, and multicore orchestration. While these test cases were not directly sourced from a commercial development project, they faithfully reflect practices common across various ongoing industry projects.

The evaluation board in Fig. 2 used features of two CPU cores and employed a typical asymmetric multiprocessing configuration. For all test cases, the scripts generated by our system were verified on the actual hardware environment.

Table II reports the man-hours required for test case execution using (a) conventional manual execution (i.e., step-wise test tool operation as performed by experienced engineers) and (b) the proposed LLM+RAG-based test script generation and execution workflow[3].

TABLE II
Comparison of test execution man-hours between manual operation and script-based (proposed).

| No. | Proposed | Manual | No. | Proposed | Manual |
|---|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 6 | 14 |
| 2 | 5 | 10 | 7 | 5 | 5 |
| 3 | 4 | 4 | 8 | 4 | 9 |
| 4 | 3 | 4 | 9 | 5 | 7 |
| 5 | 5 | 12 | Total | 40 | 70 |

*Unit: [min]

The results demonstrate a substantial reduction (43%) in required test execution man-hours. While test script generation itself incurs some cost, the overall efficiency gain justifies the investment—a key consideration in resource-constrained industrial settings.

## VI. Discussion

Our industrial case study exemplifies both the promise and the challenges of applying LLM+RAG-based automation to automotive integration testing.

**Bridging the Practice–LLM Gap.** Although earlier literature recognizes the code- and document-generating strength of LLMs, their out-of-the-box performance is insufficient for highly specialized industrial tasks. In regulated domains, the critical gap is not solely the tool API knowledge, but the combination of process conventions, hardware idiosyncrasies, and organizational workflows. Our results underscore the importance of supplementing both documented API and "implicit" workflow knowledge, often sourced from experienced test engineers, to the RAG vector store.

[3]This comparison focuses specifically on test execution man-hours, as is standard in industry when quantifying efficiency gains for integration test processes.

TABLE I. Representative integration test cases and results of the test script generation.

| No. | Test case specification | Generation |
|---|---|---|
| 1 | Verify that each core successfully gets its own core number in the start up code. | Success |
| 2 | Check processing contents of `GHS_startup` in Core 0. | Success |
| 3 | Verify that Core 0 successfully gets its own core number in `main` function. | Success |
| 4 | Verify that Core 1 successfully gets its own core number in `main` function. | Success |
| 5 | Check that `StartOtherCores` function and `WaitForCore0Ready` function are synchronized between cores. | Success |
| 6 | Verify that `SyncCores` function is synchronized between cores. | Success |
| 7 | Verify that the processing of `mainPE0` function is working. | Success |
| 8 | Verify that the processing of `mainPE1` function is working. | Success |
| 9 | Confirm that the processing of `mainPE0` and `mainPE1` functions continue with no errors. | Success |

**Prompts Containing Code Prefixes Enhance Reliability.** The adoption of prompts containing a code prefix, tailored manually to the hardware and system configuration, was key to generating robust scripts. Unlike generic code snippets or boilerplate, these prompts can in principle be auto-generated by parsing hardware or tool documentation—a promising direction for further automation. At present, any hardware change (such as replacing the microcontroller) requires code prefix adaptation, which is a prime target for systematization in future work.

**On Evaluation and Significance.** Efficiency in terms of man-hours saved is an expected basic advantage of automation. However, in practice—especially in safety-critical sectors—the main bottleneck is the reliability, accuracy, maintainability, and traceability of generated artifacts. While our evaluation focused on execution effort, future studies must address coverage, accuracy (script correctness), and maintainability—such as the ability to adapt scripts to requirement or hardware changes over the project lifecycle.

**Practical Impact and Limitations.** Importantly, the test cases for evaluation, while not copied from any specific project, were designed as "archetypes" reflecting common scenarios in contemporary multi-core ECU software integration. The same types of test procedures, as confirmed by industrial collaborators, are executed across several commercial development efforts. However, the current evaluation was bounded in terms of project diversity, scale, and test coverage.

**Scope and Future Work.** Several promising avenues exist. First, measuring and improving script maintainability and test coverage will further validate the industrial value. Second, automating code prefix generation for arbitrary hardware configurations (possibly via LLM-driven parsing of tool manuals) may eliminate one current barrier. Third, extending the approach beyond test execution, e.g., to test specification authoring, traceability analysis, or even system-level validation and requirements engineering, aligns with observed industrial needs.

## VII. Conclusion

We presented a practical approach to automate the generation of integration test scripts for automotive software using LLMs with RAG. By bridging the knowledge gap between LLMs and industrial test workflows—via a carefully structured vector store and use of high-level prompts containing code prefixes—our method enables the automatic production of executable test automation scripts for complex multi-core environments. In evaluation, all representative scripts could be generated and executed, lowering test execution man-hours by 43% for typical ECU development cases.

Our work demonstrates the importance of context/supplemental knowledge in LLM-based automation for regulated domains, and highlights the critical role of process- and hardware-aware prompt engineering. For future research, we plan to quantify potential gains in script maintainability, test coverage, and lifecycle adaptation, and to explore the use of automated code prefix construction for arbitrary hardware and toolchains.

## References

[1] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 33–42.

[2] T. Litman, "Autonomous vehicle implementation predictions," 2017.

[3] VDA QMC Working Group 13 / Automotive SIG, "Automotive SPICE Process reference model process assessment model version 3.1," 2023, (accessed on Feb 28, 2025). [Online]. Available: https://vda-qmc.de/wp-content/uploads/2023/02/Automotive_SPICE_PAM_31_EN.pdf

[4] R. Messnarz, H.-L. Ross, S. Habel, F. König, A. Koundoussi, J. Unterrreitmayer, and D. Ekert, "Integrated automotive spice and safety assessments," *Software Process: Improvement and Practice*, vol. 14, no. 5, pp. 279–288, 2009.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[7] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, 2023.

[8] X. Hu, J. Lu, Q. Cheng *et al.*, "Large language model-enable knowledge exploration for automotive assembly process using ontology."

[9] L.-B. Hernandez-Salinas, J. Terven, E. A. Chavez-Urbiola, D.-M. Córdova-Esparza, J.-A. Romero-González, A. Arguelles, and I. Cervantes, “Idas: Intelligent driving assistance system using rag,” *IEEE Open Journal of Vehicular Technology*, vol. 5, pp. 1139–1165, 2024.

[10] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software testing with large language models: Survey, landscape, and vision,” *IEEE Transactions on Software Engineering*, 2024.

[11] N. Choi, G. Byun, A. Chung, E. S. Paek, S. Lee, and J. D. Choi, “Trustworthy answers, messier data: Bridging the gap in low-resource retrieval-augmented generation for domain expert systems,” *arXiv preprint arXiv:2502.19596*, 2025.

[12] Lauterbach, “TRACE32.” [Online]. Available: https://www.lauterbach.com/

[13] Renesas Electronics, “RH850 Automotive Microcontrollers.” [Online]. Available: https://www.renesas.com/en/products/microcontrollers-microprocessors/rh850-automotive-mcus