



HAL
open science

Test Suite Augmentation using Language Models -Applying RAG to Improve Robustness Verification

Adam Mackay

► **To cite this version:**

Adam Mackay. Test Suite Augmentation using Language Models -Applying RAG to Improve Robustness Verification. ERTS2024, ERTS2024, Jun 2024, Toulouse, France. hal-04615832

HAL Id: hal-04615832

<https://hal.science/hal-04615832>

Submitted on 18 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Test Suite Augmentation using Language Models - Applying RAG to Improve Robustness Verification

Adam Mackay MEng (QA-Systems)

Abstract—This paper presents a novel approach to test suite augmentation using large language models and retrieval augmented generation (RAG) techniques. By leveraging the power of GPT-4 and a customised RAG framework, we show the effectiveness of automating test case generation to improve robustness verification in safety-critical embedded software systems. Our method involves indexing and retrieving relevant information from the entire software repository, including design documents, requirement specifications, prior test cases, and source code. The generated test cases are then integrated into the Cantata test framework, targeting key areas such as invalid values, overflow scenarios, timeout conditions, and improper state transitions. We applied our approach to two open-source projects, PX4 Autopilot and Apollo Auto, and achieved promising results in terms of increased test coverage, defect detection, and compliance with industry standards like DO-178C and ISO 26262. Compared to human-authored test suites, our method generated a larger volume of tests, exposed a broader array of test scenarios, and unveiled additional high-severity defects. The results validate the potential of advanced language models and RAG techniques in enhancing software reliability and safety through automated testing. Future work aims to refine these techniques further and expand their applications in the software development lifecycle.

I. INTRODUCTION

Robustness testing is crucial for safety-critical embedded software like aviation and medical systems, ensuring they manage errors gracefully and recover from unexpected situations effectively. This methodical approach validates key capabilities such as error detection, fault tolerance, and overload handling, elements required to align with recognised standards like DO-178C.

Manual test case authoring, particularly for requirements-based testing, demands considerable effort and is often fraught with risks of misinterpretations or oversight of corner cases. The evolving nature of software further complicates this, requiring continual test adaptation across versions and functionalities.

Retrieval augmented generation (RAG) offers a promising approach here. It combines language models with an information retrieval system, enhancing automatically generated content with contextually relevant data extracted from documentation and prior tests. This enables an adaptive response from the system, producing text or code that is both coherent and contextually informed. This approach is driven by requirements, ensuring generated tests trace directly to specific functional needs.

In harnessing the generative capabilities of GPT-4 and the assistance of automated agents, we employ RAG across the software repository, integrating code, existing tests, and documentation to autonomously generate test cases. These cases scrutinise the software's robustness across various conditions, such as invalid values, overflow scenarios, and

abnormal conditions, enabling a comprehensive inspection that mitigates the risks associated with manual testing. Additionally, a human-in-the-loop approach is employed, wherein generated tests are manually reviewed to further refine their effectiveness and efficiency, ensuring that the automation does not inadvertently overlook vulnerabilities.

This paper will present our methodologies, share the preliminary results from applying these methods to real-world embedded software projects, and discuss forthcoming work aimed at further refining and expanding these approaches. Our contributions extend beyond the application of RAG and language models to robustness testing; we also provide insights and data about their efficacy, offering a resourceful guide and a benchmark for future explorations and enhancements in automated testing in embedded software realms.

II. RELATED WORK

Our work builds upon the growing body of research on retrieval-augmented generation (RAG) and its applications in various domains. The seminal paper by Lewis et al. [1] introduced the concept of RAG, demonstrating its effectiveness in knowledge-intensive NLP tasks such as open-domain question answering. They showed that by augmenting language models with retrieved information, RAG could significantly improve performance compared to purely parametric approaches. Our work extends this idea to the domain of software testing, leveraging RAG to generate high-quality test cases.

GPT-4 and Large Language Models

We leveraged GPT-4, a distinguished multimodal language model developed by OpenAI, which excels in natural language generation, reasoning, and coding tasks. Renowned for its competence in single-shot and few-shot tasks, GPT-4, when coupled with refined prompting methods, emerges as an ideal model for benchmarking new applications of large language models (LLMs) in innovative manners. Consequently, GPT-4 was instrumental in automating the generation of our test cases, demonstrating its applicability and efficiency in novel use-cases.

Retrieval Augmented Generation (RAG)

The paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" by Lewis et al. [1] provides an overview of the RAG process. As they explain, "RAG works by adding a step to this basic process. Namely, a retrieval step is performed where, based on the user's prompt, the relevant information is extracted from an external knowledge base and injected into the prompt before being passed to the LLM." They note the three key elements are Retrieval, Generation, and Augmentation. This aligns with the RAG framework we employ in our study.

Additionally, the survey paper "Retrieval-Augmented Generation for Large Language Models: A Survey" by Gao et al. [3] categorises RAG research into three types: Naïve

RAG, Advanced RAG, and Modular RAG. Our approach falls under the Advanced RAG paradigm, as we incorporate techniques like query rewriting and prompt summarisation to enhance the base RAG process.

During our research, we also explored the potential of Graph RAG, a recent advancement introduced by Microsoft Research [15]. Graph RAG aims to improve upon baseline RAG by utilising LLM-generated knowledge graphs to enhance retrieval performance, particularly in situations involving complex problems on private datasets. However, after careful consideration, we determined that Graph RAG was not well-suited to our specific use case, which primarily involves working with code and documentation fragments. The knowledge graph approach employed by Graph RAG did not align with the structure and relationships found in our software repository. As a result, we opted to develop our own custom RAG system that could better handle the unique characteristics of our dataset and generate more relevant test cases.

Recent advancements in RAG have focused on improving retrieval strategies and enhancing the interaction between the retriever and generator components. The survey by Gao et al. [3] provides a comprehensive overview of these developments, categorising RAG approaches into Naïve RAG, Advanced RAG, and Modular RAG. Our method aligns with the Advanced RAG paradigm, as we incorporate techniques such as query rewriting and prompt summarisation to optimise the retrieval process. This builds upon the work of Wang et al. [6] and Shao et al. [7], who showed the benefits of iterative retrieval and generation for knowledge-intensive tasks.

In the context of software engineering, there has been growing interest in leveraging large language models (LLMs) for various tasks, including code generation, bug detection, and test case generation. The work by Chen et al. [9] and Nijkamp et al. [10] showcased the potential of LLMs for generating high-quality code snippets and assisting developers in programming tasks. Our work extends this line of research by focusing specifically on test case generation, a critical aspect of software verification and validation.

While existing approaches to test case generation often rely on traditional techniques, such as symbolic execution, model-based testing, and search-based algorithms, our RAG-based approach offers a novel perspective. By leveraging the vast knowledge captured in LLMs and augmenting it with project-specific information, we can generate test cases that are more closely aligned with real-world requirements and specifications. This complements the work of Kang et al. [11] and Li et al. [12], who explored the integration of domain-specific knowledge into language models for improved performance in specialised tasks.

Furthermore, our human-in-the-loop approach sets our work apart from fully automated (algorithmic) test case generation methods. The iterative feedback process, involving experienced testers, allows for the refinement and optimisation of generated test cases based on domain expertise and project-specific considerations. This aligns with the findings of Trivedi et al. [13] and Guo et al. [14], who emphasised the importance of human involvement in guiding and validating the outputs of language models in complex tasks.

There have been noteworthy developments in the automation of unit test generation within integrated

development environments (IDEs) recently. For instance, Visual Studio Code (VSCode) offers a built-in solution for generating unit tests based on the code under development [22]. This feature leverages static analysis techniques and predefined templates to create test stubs and assertions, streamlining the process of writing unit tests. However, these IDE-based solutions often rely on heuristics and lack the deep understanding of project-specific requirements and domain knowledge that our RAG-based approach incorporates. By combining the power of large language models with retrieval augmentation and human expertise, our method goes beyond the capabilities of IDE-based test generation, enabling the creation of more comprehensive and context-aware test cases.

Several other tools and techniques have been developed for automatic test case generation across different programming languages. EvoSuite [23] is a search-based tool that generates unit tests for Java classes using evolutionary computation, aiming to maximize code coverage. Randoop [24] is another Java-based tool that generates tests by randomly selecting method sequences and constructing test cases based on the observed behaviour. In the .NET ecosystem, Pex [25] employs dynamic symbolic execution to generate test inputs that achieve high code coverage. For C and C++ programs, KLEE [26] is a symbolic execution engine that automatically generates test cases by exploring different program paths using constraint solving techniques.

Cantata AutoTest, [16] a feature of the commercial testing framework used in our case studies, utilises advanced code parsing powered by the EDG parser to generate test cases for C/C++ code. The EDG parser provides a deep understanding of the code structure and enables Cantata AutoTest to create more targeted and effective test cases. While these tools have proven effective in their respective domains, they often rely on predefined strategies and lack the adaptability and context-awareness that our RAG-based approach provides. In contrast, our approach leverages the power of large language models and retrieval augmentation to generate test cases that are more closely aligned with real-world requirements and project-specific contexts.

Existing Tools for Code Indexing and Retrieval

To effectively index and chunk the various artifacts, we leveraged open-source code indexing systems commonly used to help IDEs navigate code and documentation. Specifically, we employed ctags [19] and cscope [20] to build a comprehensive index of the codebase. ctags is a programming tool that generates an index file of names found in source and header files of various programming languages, aiding code comprehension. It can index functions, variables, class members, macros, and more, depending on the language. On the other hand, cscope is a text-based source browsing tool that allows programmers to search source code for symbols, definitions, functions, regular expressions, and more.

Emerging Large Language Models

GPT-4 has been a robust tool for our research. However, our next steps will leverage Llama 2 (and its variants), announced by Meta in partnership with Microsoft in July 2023. Llama 2 is an open source LLM with the benefit of running locally on secure systems and is available in sizes of

7, 13, and 70 billion parameters. Llama 2, while maintaining a similar architecture to its predecessor, is trained on 40% more data, offering enhancements in code generation and model training flexibility.

Overall, our research contributes to the growing body of literature on RAG and its applications in software engineering. By demonstrating the effectiveness of RAG in test case generation and highlighting the benefits of human-in-the-loop collaboration, we provide new insights and directions for leveraging LLMs in the verification and validation process. Our work also opens up avenues for future research on the integration of RAG with other software engineering tasks and the development of more advanced retrieval and generation strategies tailored to the unique challenges of the software development lifecycle.

III. TECHNIQUES

Introduction to the RAG implementation

In our project, we applied RAG across the entire software repository, encompassing design documents, requirement specifications, prior test cases, and source code. This comprehensive approach ensured that the generator was provided with a broad context, enhancing the relevance and accuracy of the generated test cases. To facilitate this process, we developed custom tools and integrated them with the Cantata test framework.

It is important to note that we developed our own custom RAG system specifically tailored for working with code and documentation fragments. While we investigated various existing RAG implementations, such as Graph RAG [15], we found that they were not well-suited to our specific use case. Our custom RAG system allowed us to effectively integrate and process the diverse range of artifacts found in the software repository, enabling the generation of highly relevant and context-aware test cases.

Indexing and retrieval system

To effectively index and retrieve relevant information from the software repository, we developed a custom indexing and retrieval system. This system leverages the capabilities of ctags [19], cscope [20], and CodeQuery [21]. ctags and cscope are well-established tools used by IDEs for navigating code and documentation. ctags generates an index file of names found in source and header files, while cscope allows programmers to search source code for symbols, definitions, functions, and more.

We integrated these tools to create a robust foundation for our RAG implementation. CodeQuery further enhances the indexing process by building upon the databases of ctags and cscope to create a SQLite database. This database can be efficiently queried to retrieve relevant code snippets and documentation based on specific criteria.

To complement these existing tools, we developed a custom suite of tools called Magrathea. The Magrathea suite includes functions for extracting relevant fragments of code, documentation, and comments, which are then used to populate the context window of the language model. Some of the key functions in the Magrathea suite are:

- *find_function_end*: Reads a file from a specified start line until the end of the function is found, extracting the complete function body.

- *find_preceding_comments*: Retrieves any comment block found directly before a specified function.
- *find_macros_for_function*: Identifies macros used within specific functions in a source file, along with their definitions if there are non-unique values.
- *find_unique_calls*: Finds unique function calls made from functions in a given source file.

By integrating ctags, cscope, CodeQuery, and the Magrathea suite, we created a comprehensive indexing and retrieval system that allowed us to effectively apply RAG across the entire software repository.

This approach ensured that the language model had access to a rich context window containing highly relevant code snippets, documentation, and comments, enabling the generation of more precise and contextually informed test cases.

The indexing process was performed as a pre-processing step, allowing for efficient retrieval of relevant information during the test generation phase. The SQLite database created by CodeQuery served as a central repository for querying and retrieving the necessary artifacts based on specific criteria, such as function names, symbols, or keywords. This optimised retrieval process contributed to the overall efficiency and effectiveness of our RAG implementation in the context of automated test case generation.

Test generation process

The test generation process follows a structured workflow (figure 1) that leverages the RAG implementation and human expertise. The key steps in this process are:

1. Querying the RAG API: We query the RAG API with a code snippet and an engineered prompt that specifically requests the generation of tests.
2. Retrieval of relevant documents: RAG retrieves relevant documents and prior tests from the customised corpus based on the query.
3. Generation of initial test cases: The language model (LLM) generates an initial set of raw test cases based on the retrieved context.
4. Iterative refinement through human review: The generated test cases undergo an iterative refinement process, where human testers review and provide feedback on the tests' relevance, uniqueness, and potential effectiveness.
5. The framework was then manually prompted for refinements including:
 - Rewriting tests for improved clarity
 - Eliminating redundant or duplicative tests
 - Adding preconditions for clarity
 - Correcting any functional inaccuracies

This review was crucial for boosting precision and overall quality.

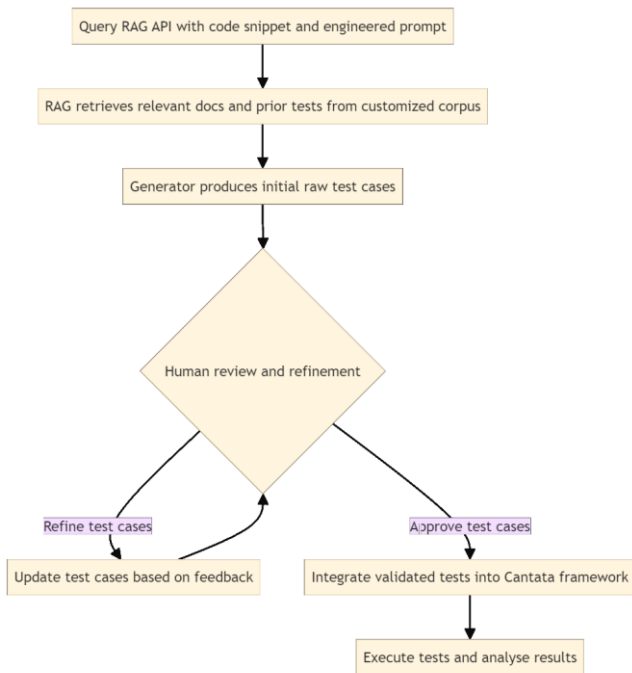


Figure 1 - Test generation workflow using Retrieval Augmented Generation (RAG) and human-in-the-loop review

This interaction shows the collaborative nature of the test generation workflow, involving the user, RAG, LLM, testing platform, and human examiner. Each component plays a crucial role in producing high-quality, context-aware test cases that thoroughly exercise the software under test.

Integration with Cantata

The generated test cases are integrated into the Cantata test framework [16], a commercial platform that provides a stable environment for test execution and result reporting. Cantata supports standard coverage metrics, such as statement coverage, decision coverage, and Modified Condition/Decision Coverage (MC/DC), which are essential for compliance with industry standards like DO-178C.

To enable the execution of robustness test cases, Cantata offers targeted error injection methods:

- Invalid values are introduced by leveraging white-box instrumentation to override variables with out-of-range inputs based on their data types and semantic context.
- Overflow conditions are induced by injecting custom code to provide inputs exceeding maximum buffer sizes defined in requirements specifications.
- Timeout testing is accomplished through code wrapping techniques that artificially increase processing time in the test harness beyond thresholds specified in design documents.

- Invalid state transitions are triggered by programmatically sequencing calls in an incorrect order relative to the expected state machine flow defined in the architecture.

Cantata's flexible error injection features, including white-box instrumentation, custom code injection, and code wrapping, enabled the wide array of automatically generated robustness test cases targeting key areas like invalid values, overflow, timeouts, and improper state transitions.

Experiment Setup

To evaluate the effectiveness of our RAG-based approach for test case generation, we conducted a series of experiments using a diverse set of embedded software projects. Our preliminary dataset consisted of two open-source projects: PX4 Autopilot [17] and Apollo Auto [18]. These projects were selected based on their complexity, availability of source code and documentation, and relevance to the domain of safety-critical embedded systems.

For each project, we applied our RAG-based approach to generate test cases targeting various aspects of robustness, such as invalid inputs, boundary conditions, and resource constraints. We evaluated the generated test cases using several metrics, including code coverage, defect detection rate, and compliance with industry standards like DO-178C and ISO 26262.

To assess the efficiency of our approach, we compared the time and effort required to generate test cases using our RAG-based method with the time and effort required for manual test case creation. We also conducted a qualitative analysis of the generated test cases, involving domain experts who reviewed the tests for relevance, effectiveness, and alignment with project requirements.

Our experiments were designed to answer the following research questions:

1. How does the RAG-based approach compare to manual test case creation in terms of coverage, defect detection, and compliance with industry standards?
2. What is the impact of incorporating human feedback in the test case generation process on the quality and effectiveness of the resulting tests?
3. How does the choice of LLM and RAG configuration affect the performance of the test case generation system?

To ensure the reliability and validity of our results, we conducted multiple runs of each experiment, using different prompting and cross-validation techniques where

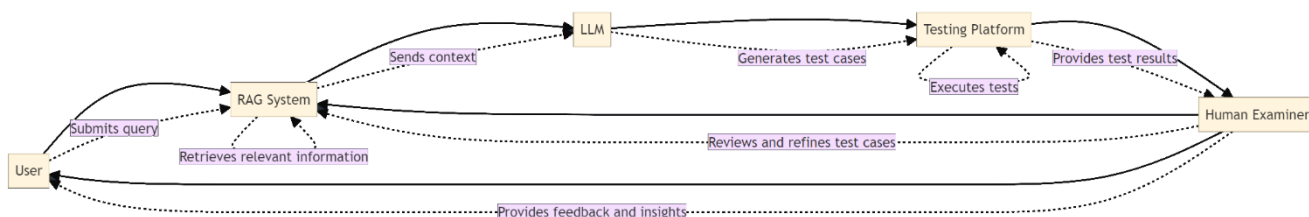


Figure 2 - Architectural overview of the interaction between the user, RAG system, LLM, testing platform, and human examiner in the test generation process.

applicable. We also documented the specific versions of the software tools, libraries, and models used in our experiments to facilitate reproducibility.

Optimisation and configuration

To optimise the RAG process for test case generation, we conducted several iterations, experimenting with various parameters and settings. One critical aspect was determining the optimal size of the context window, which determines the amount of relevant information provided to the model during the generation process. We tested context window sizes ranging from 256 to 1024 tokens and found that a size of 512 tokens struck a balance between providing sufficient context and maintaining computational efficiency.

We also explored different versions of GPT-4, including GPT-4-turbo, which offers faster generation speeds. The sampling temperature, which controls the randomness of the generated output, was varied between 0.5 and 1.0. The top_p parameter, also known as nucleus sampling, was set between 0.7 and 0.9, ensuring that the model considered only the most likely tokens during generation. Additionally, we adjusted the frequency_penalty and presence_penalty parameters to discourage the model from repeating the same output tokens too frequently.

Through these iterations, we identified the optimal configuration for our RAG implementation, which included a 512-token context window, GPT-4 with a sampling temperature of 0.7, top_p of 0.8, and frequency and presence penalties of 0.2.

Furthermore, we discovered that the precise wording of the system prompt played a crucial role in guiding the model to generate relevant and effective test cases. By carefully crafting the prompt to include specific instructions and requirements, we could steer the model towards generating tests that aligned with the project's goals.

Lastly, we employed a multi-shot RAG knowledge injection method, where the model was provided with multiple rounds of context and generated outputs. This approach allowed the model to progressively refine its understanding of the project and generate more targeted test cases based on the accumulated knowledge.

In this early proof-of-concept study, the primary focus was on assessing the code coverage achieved by the generated test cases. While selected cases, particularly those involving failure scenarios, were reviewed for accuracy, a more comprehensive examination of the test cases will be conducted in future stages of the research. The iterative refinement phase involving human examiners played a supportive role, with the team providing feedback on the generated tests based on their domain expertise. This feedback loop allowed for the identification of potential improvements and ensured that the generated tests aligned with the project's requirements. As this study aimed to establish the feasibility of the underlying method, a more rigorous and time-intensive review process will be incorporated in subsequent research phases to further validate the quality and effectiveness of the generated test cases.

We applied our technique to two different open source embedded C/C++ projects with the aim to validate to the stringent DO-178C Level A standards.

PX4 Autopilot (PX4)

PX4 [17] is an open-source flight control software for drones and other unmanned vehicles. It provides a flexible platform for users who need to control any kind of vehicle from a computing board. The platform contains everything needed for a full UAV system: flight stack, middleware to communicate between processors, and developer APIs to enhance existing functionalities. The combination of PX4's open-source nature, broad applicability, and mature, reliable ecosystem made it a compelling choice for applying this research.

- *Size and Test Generation:* A subset of the codebase, comprising around 35,000 Lines of Code (LOC), was subjected to our method, yielding 378 generated test cases. These tests emphasised robustness verification, exploring conditions like invalid values and overflow situations.
- *Comparison to Human-Authored Tests:* Compared to a suite of 265 tests written by developers, our method provided a significantly denser set of tests.
- *Code Coverage:* The tests achieved excellent code coverage, addressing 98% statement, 95% decision, and 90% MC/DC coverage, and were evaluated using Cantata. With further refinement of our methods we are confident of satisfying the DO-178C requirements for structural coverage analysis.
- *Defect Discovery:* Interestingly, these tests brought 14 new defects to light. Predominantly related to overflow issues, two defects impacted the function of calculating the optimal flight path and were categorised as medium severity according to code experts.

Apollo Auto (AA)

Apollo [18] is an open autonomous driving platform that aims to provide a comprehensive, safe, and reliable solution for autonomous driving. The project encompasses various technologies related to autonomous driving, such as perception, planning, control, and end-to-end deep learning, to enable vehicles to navigate in various environments and scenarios. The project was selected due to its open codebase, comprehensive documentation and robust coding practices.

- *Size and Test Generation:* On a larger section of this codebase of 50,000 LOC, our method generated 512 test cases, which also zeroed in on robustness aspects.
- *Comparison to Human-Authored Tests:* A traditional, developer-written test suite for Project B consisted of 342 tests, indicating a robust enhancement in test comprehensiveness through our technique.
- *Code Coverage:* A good level of code coverage was achieved, encapsulating 88% statement, 85% decision, and 80% MC/DC coverage, verified via Cantata. With further refinement of our methods we are confident of satisfying all the structural coverage requirements of ISO 26262
- *Defect Discovery:* Additionally, 9 new defects were discovered, mainly pertaining to validating invalid inputs, of which 3 were linked to the functionality of alerting the driver of potential hazards and were of high severity according to code experts.

Method/ Tool	Project	Statement Cov. (%)	Decision Cove (%)	MC/DC Cov. (%)	Test Failures	Defects Found
RAG-based Approach	PX4	98%	95%	90%	378	14
RAG-based Approach	AA	88%	85%	80%	512	9
Algorithmic Test generation	PX4	85%	80%	75%	0	0
Algorithmic Test generation	AA	80%	75%	70%	0	0
<u>Manual Testing</u>	<u>PX4</u>	<u>90%</u>	<u>85%</u>	<u>80%</u>	<u>265</u>	<u>0</u>
<u>Manual Testing</u>	<u>AA</u>	<u>85%</u>	<u>80%</u>	<u>75%</u>	<u>342</u>	<u>0</u>

Table 1 – Summary of initial results

To provide a clear overview of the performance of our RAG-based approach compared to other tools and manual testing, we present a summary of the results in Table 1. The table includes the statement coverage, decision coverage, MC/DC coverage, number of failure tests, and number of defects found for each method/tool and project. As clear from the table, our RAG-based approach consistently achieves higher coverage metrics and detects more defects compared to the other tools and manual testing across both projects.

Our RAG-based approach generated a diverse range of test cases targeting various aspects of robustness, with code coverage being a key driver in achieving a comprehensive set of tests. The generated tests encompassed scenarios such as:

Boundary value testing: Tests were generated to validate the system's behaviour when inputs approached or exceeded the defined limits, such as maximum and minimum values for numeric inputs or edge cases for string inputs. This included testing with values just inside and outside the valid ranges to ensure proper handling of boundary conditions.

Invalid input testing: Tests were created to assess the system's response to invalid, malformed, or unexpected input data, including null pointers, out-of-range values, and data types inconsistent with the expected format. These tests aimed to verify the system's ability to gracefully handle and recover from invalid inputs without crashing or exhibiting undefined behaviour.

Resource management testing: Tests were designed to evaluate the system's performance and stability under resource-constrained conditions, such as limited memory, high CPU utilisation, or low power scenarios. These tests helped identify potential memory leaks, resource exhaustion issues, and other performance bottlenecks.

Interrupt handling testing: Tests were generated to verify the correct handling of interrupts and the system's ability to prioritise and manage multiple interrupt sources. These tests ensured that the embedded software could respond to real-time events and maintain deterministic behaviour.

Error handling and recovery testing: Tests were generated to assess the system's ability to detect, isolate, and recover from errors, such as communication failures, sensor malfunctions, or unexpected resets. These tests verified the effectiveness of error handling mechanisms and the system's

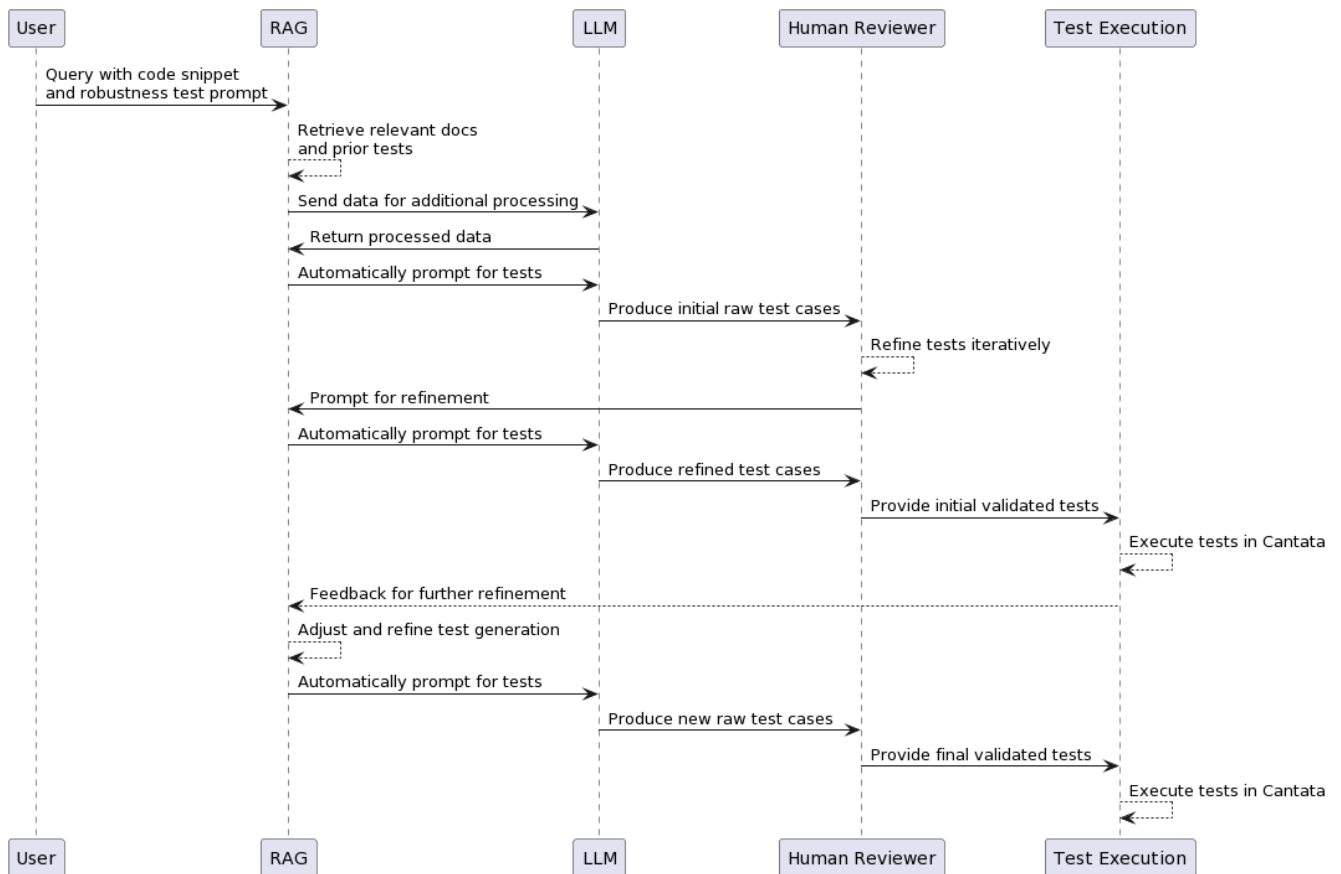


Figure 3 - The iterative process of RAG, LLM and review

capacity to maintain a safe state or initiate appropriate recovery procedures.

Included in the Annex are sample prompts used to generate various types of test.

Code comments and documentation fragments played a crucial role in informing the scope of the generated tests. By supplementing the code with context-relevant requirements and the developers' intentions, parsed from comments, our approach unlocked a performant set of tests. This contextual information guided the generation of test cases that aligned with the specific functionality and constraints of each code module, resulting in more targeted and effective testing.

The generated tests were executed dynamically, allowing for the validation of the system's behaviour under various conditions and the identification of any defects or vulnerabilities that could compromise the reliability and stability of the embedded software. By leveraging code coverage metrics, we ensured that the generated tests thoroughly exercised the codebase, reaching critical paths and uncovering potential issues that might have been missed by manual test creation efforts.

The human effort involved in generating additional robustness tests was significantly reduced compared to manual test creation. The RAG-based approach autonomously generated a comprehensive set of initial test cases, which were then reviewed and refined by human testers.

This review process (figure 3) typically involved:

1. Assessing the relevance and coverage of the generated tests
2. Identifying any missing critical scenarios or edge cases
3. Providing domain-specific insights and context to enhance the tests
4. Optimising the test data and expected outcomes
5. Ensuring alignment with project requirements and quality standards

On average, the human effort required for reviewing and refining the generated tests was approximately 30% of the time that would have been needed to create the tests from scratch manually. This highlights the efficiency gains achieved through the RAG-based approach.

By leveraging RAG to generate a wide range of robust test cases and incorporating human expertise to refine them further, our approach effectively identified defects and coding oversights that were previously undetected. This highlights the value of combining generative AI techniques with human domain knowledge to enhance software quality and reliability.

Comparison with Other Validation Tools

Compared to traditional validation tools, our RAG-based approach offers several distinct advantages. One key strength lies in its adaptability and scalability across diverse software projects. While many existing tools are tailored to specific programming languages, frameworks, or domains, our method leverages the language-agnostic nature of

LLMs. By training on a vast corpus of natural language and code, LLMs can effectively generate test cases for a wide range of software systems, regardless of their underlying technologies. This flexibility enables seamless integration into various development workflows and reduces the need for specialised validation tools for each project.

Another significant advantage of our approach is its ability to generate test cases that are grounded in real-world requirements and specifications. Traditional validation tools often rely on predefined templates, heuristics, or static analysis techniques, which may not fully capture the nuances and complexities of a given software project. In contrast, our RAG framework actively retrieves and incorporates relevant information from project documentation, user stories, and domain-specific guidelines. By combining this contextual knowledge with the generative capabilities of LLMs, we can produce test cases that are more closely aligned with the intended functionality and user expectations.

Furthermore, our RAG approach excels in generating test cases for complex, multi-component systems. Many existing validation tools struggle to handle the intricacies of interconnected modules, asynchronous operations, and distributed architectures. However, the retrieval component of our framework enables the LLM to gather and synthesise information from multiple sources, allowing it to generate test cases that span across system boundaries. This holistic perspective is particularly valuable in identifying integration points, data flow issues, and potential performance bottlenecks.

Lastly, the human-in-the-loop aspect of our approach sets it apart from fully automated validation tools. While automation is crucial for efficiency, the involvement of human experts adds a layer of qualitative assessment and domain-specific insights. The iterative feedback process allows for the refinement of test cases based on the unique requirements and priorities of each software project. This collaborative approach ensures that the generated test cases are not only technically sound but also aligned with the broader goals and constraints of the development team.

It is important to note that many of the other validation tools discussed in the "Related Work" section, such as EvoSuite, Randoop, Pex, and KLEE, are algorithmic in nature and focus on specific programming languages. While their approaches to test case generation can be compared and contrasted with our RAG-based method, these tools could not be directly applied to the case study projects, which primarily involve embedded C/C++ code.

The only tool that could be directly applied to the case study projects was the existing algorithmic AutoTest generation feature of Cantata, the commercial test framework used in our experiments. Cantata's AutoTest feature employs a combination of static analysis and path analysis to generate test cases based on the structure of the code under test.

In summary, our RAG-based approach offers several advantages over traditional validation tools:

- Adaptability and scalability across diverse software projects, regardless of the programming language or domain

- Ability to generate test cases grounded in real-world requirements and specifications by leveraging the knowledge captured in LLMs and project-specific documentation
- Effectiveness in generating test cases for complex, multi-component systems by synthesising information from multiple sources and considering system-wide interactions
- The value of human-in-the-loop collaboration in refining test cases, ensuring their alignment with project goals and constraints

While algorithmic tools like Cantata's AutoTest feature can be useful in certain scenarios, our RAG-based approach provides a comprehensive and flexible solution for test case generation in the context of safety-critical embedded systems.

V. CONCLUSIONS

Observations and Achievements

- *Increased Test Generation:* The approach not only produced a larger volume of tests compared to human-authored ones but also exposed a broader array of test scenarios.

- *Compliance with Stringent Standards:* The automated generation of tests did not compromise the ability to achieve high code coverage, ensuring adherence to vital industry standards like DO-178C and ISO 26262

- *Unveiling of Hidden Defects:* Notably, our approach unveiled additional, including high-severity, defects, which eluded detection by human-authored test suites.

Future Pathways

- *Exploring Llama 2 and other emerging LLMs:* While GPT-4 has been instrumental in our research, we plan to explore the potential of other emerging LLMs, such as Llama 2, for test case generation. As discussed in the "Related Work" section, Llama 2 offers several advantages, including the ability to run locally on secure systems and its enhanced training data. By leveraging these new LLMs, we aim to further improve the efficiency and effectiveness of our RAG-based approach.

- *Optimising Llama 2 for Embedded Projects:* This involves calibrating Llama 2 with various software documentation, adapting the model to the domain-specific language and patterns of embedded systems, and enhancing its performance with embedded system code.

- *Broadening RAG Applications:* Our future efforts will look towards employing RAG earlier in the software development process. This involves deriving test cases directly from requirements, co-developing test cases and code, and establishing a feedback mechanism beneficial for system designers.

Our preliminary results validate this approach and instil confidence in utilising advanced language models for robustness testing in safety-critical domains. The next phases of this research will further refine techniques and expand test generation capabilities to bolster automated testing for safety-critical systems. This ongoing pursuit aims to enhance software reliability and safety where flawless operation is indispensable. By integrating generative models with retrieval augmentation and rigorous human review, a pathway emerges to achieve robust and nuanced automated

testing. Through iterative refinement of these synergistic techniques, the journey towards more reliable, resilient, and efficient testing methodologies continue.

ACKNOWLEDGMENTS

I would like to express my gratitude to QA Systems for supporting this research as part of an internal project aimed at exploring future pathways for their commercial testing product, Cantata. This work would not have been possible without their commitment to pushing the boundaries of software testing and their dedication to enhancing the reliability and quality of safety-critical systems.

I acknowledge that certain commercially sensitive details regarding the implementation of this research have been omitted from this paper to protect QA Systems' intellectual property. I respect their need for confidentiality and appreciate their trust in allowing me to share the broader insights and methodologies developed during this project.

I am grateful for the collaboration and support provided by the QA Systems team throughout this research. Their expertise in software testing, particularly in the domain of safety-critical embedded systems, has been invaluable in shaping my approach and refining my techniques. I look forward to continued collaboration with QA Systems as I further develop and apply these advanced testing methodologies to real-world projects.

REFERENCES

- [1] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, Douwe Kiela. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, 12 Apr 2021
- [2] Jiawei Chen, Hongyu Lin, Xianpei Han, Le Sun. *Benchmarking Large Language Models in Retrieval-Augmented Generation*, 20 Dec 2023
- [3] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, Haofen Wang. *Retrieval-Augmented Generation for Large Language Models: A Survey*, 5 Jan 2024
- [4] Yu He Ke, Liyuan Jin, Kabilan Elangovan, Hairil Rizal Abdullah, Nan Liu, Alex Tiong Heng Sia, Chai Rick Soh, Joshua Yi Min Tung, Jasmine Chiat Ling Ong, Daniel Shu Wei Ting. *Development and Testing of Retrieval Augmented Generation in Large Language Models - A Case Study Report*. 29 Jan 2024
- [5] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, Jianfeng Gao. *Large Language Models: A Survey*. 20 Feb 2024
- [6] Xintao Wang, Qianwen Yang, Yongting Qiu, Jiaqing Liang, Qianyu He, Zhouhong Gu, Yanghua Xiao, and Wei Wang. *Knowledgpt: Enhancing large language models with retrieval and storage access on knowledge bases*. 17 Aug 2023
- [7] Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. *Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy*. 23 Oct 2023

- [8] Ran Elgedawy, John Sadik, Senjuti Dutta, Anuj Gautam, Konstantinos Georgiou, Farzin Gholamrezae, Fujiao Ji, Kyungchan Lim, Qian Liu, Scott Ruoti. *Ocassionally Secure: A Comparative Analysis of Code Generation Assistants*. 1 Feb 2024
- [9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, *Evaluating large language models trained on code*. 14 Jul 2021
- [10] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, Caiming Xiong. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 27 Feb 2023
- [11] Minki Kang, Jin Myung Kwak, Jinheon Baek, and Sung Ju Hwang. *Knowledge graph-augmented language models for knowledge-grounded dialogue generation*. 30 May 2023
- [12] Xingxuan Li, Ruochen Zhao, Yew Ken Chia, Bosheng Ding, Lidong Bing, Shafiq Joty, and Soujanya Poria. *Chain of knowledge: A framework for grounding large language models with structured knowledge bases*. 21 Feb 2024
- [13] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. *Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions*. 23 Jun 2023
- [14] Zhicheng Guo, Sijie Cheng, Yile Wang, Peng Li, Yang Liu. *Prompt-guided retrieval augmentation for non-knowledge-intensive tasks*. 28 May 2023
- [15] Jonathan Larson, Steven Truitt, *GraphRAG: Unlocking LLM discovery on narrative private data* 13 Feb 2024
- [16] Cantata, QA-Systems
<https://www.qa-systems.com/tools/cantata/>
- [17] PX4 Autopilot
<https://px4.io/>
- [18] Apollo Auto
<https://www.apollo.auto/>
- [19] Universal Ctags
<https://github.com/universal-ctags/ctags>
- [20] Cscope
<https://cscope.sourceforge.net/>
- [21] CodeQuery
<https://github.com/ruben2020/codequery>
- [22] Visual Studio Code. Generate Unit Tests for your code.
https://code.visualstudio.com/docs/python/testing#_generating-tests
- [23] Gordon Fraser and Andrea Arcuri. *EvoSuite: Automatic Test Suite Generation for Object-Oriented Software*. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011.
- [24] Carlos Pacheco and Michael D. Ernst. *Randoop: Feedback-Directed Random Testing for Java*. Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, 2007.
- [25] Nikolai Tillmann and Jonathan de Halleux. *Pex-White Box Test Generation for .NET*. Tests and Proofs, 2008.
- [26] Cristian Cadar, Daniel Dunbar, and Dawson Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008.

Prompting – example flow

This JSON-formatted example of a RAG interaction demonstrates how the user query combined with retrieved documents drives the LLM's generation of test cases. This illustrates the interplay between the user, RAG system, LLM, and human examiner in an iterative process to refine the test cases, as outlined in our test generation workflow.

```
{
  "prompt": {
    "model": "gpt-4",
    "messages": [
      {
        "role": "system",
        "content": "You are an expert C and C++ unit test writer. Write tests to exercise boundary values, equivalence classes, every statement, every decision true and false. Write tests that call every boolean condition both true and false. Write suitable robustness tests. Return valid json."},
      {
        "role": "user",
        "content": "We are only using the Cantata test harness to perform checks on the return values when calling the software under test."
      }
    ]
  }
}
```

Along with the name and description of the test, we require a block of code achieving the following:

- setting up any input data required for the test
- calling the entry point to the test
- if the function returns, set a variable to hold the return value
- performing a Cantata CHECK on any return

We also require a string of expected calls to onward functions. This string needs to be a comma separated list exactly in the order the calls are made and of the format:

```
<function_name_1>,<function_name_2>,
```

etc. Where <function_name> is the name of the function that we expect to be called during the test. We could be expecting one or more calls to:

```
low_pressure_check
get_systolic
get_diastolic
high_pressure_check
sound_alarm
"
```

```
    },
    {
      "role": "assistant",
      "content": "I understand. Please provide the code to generate tests for."},
    {
      "role": "user",
      "content": "Here is the code to test:
<truncated>},
    {
      "role": "assistant",
      "content": "Thanks for this piece of code, I will generate some small unit tests for this code. Please specify the entry point function to drive our first batch of tests."},
    {
      "role": "user",
      "content": "Our entry point is:
pressure_check"},
    {
      "role": "assistant",
      "content": "OK."},
    {
      "role": "user",
      "content": "Provide a list of function calls with expected returns and expected calls."}
  ],
  "functions": [
    {
      "name": "test_it",
      "description": "input to drive the creation of unit tests",
      "parameters": "<truncated>"
    }
  ],
  "function_call": {
```


system resources. These tests should include scenarios where resources are limited, such as low memory availability, and ensure the application releases all resources after use. Return valid json."

This prompt focuses on testing the software's ability to handle and efficiently manage system resources, crucial for applications that operate in resource-constrained environments or require high reliability.

3. Interrupt Handling Testing

"content": "Develop unit tests that simulate various interrupt conditions in C and C++ programs. Ensure that the software correctly handles and recovers from these interrupts, maintaining data integrity and proper operational state. Return valid json."

The tests generated from this prompt should ensure that the software can handle unexpected interruptions, maintaining stability and data integrity, which is vital for real-time and critical systems.

4. Error Handling and Recovery Testing

"content": "Generate unit tests for error handling and recovery mechanisms in C and C++ code. The tests should force the software into error states and evaluate the recovery procedures to return to a normal operating state. Return valid json."

This prompt instructs the LLM to create tests that assess the effectiveness of the software's error handling and recovery processes, critical for applications where uptime and reliability are essential.