Issues in Information Systems
Volume 27, Issue 1, pp. xx-xx, 2026

DOI: https://doi.org/10.48009/1_iis_2021_xx-xx (Note: The header will be completed by IIS editor, keep line space)

Sky Unit: An LLM-driven unit test generator for embedded C

Gerald Rigdon
Fellow, Software Engineering, Boston Scientific Corporation, St. Paul, USA
gerald.rigdon@bsci.com

Abstract

Sky Unit is an automated unit test generation system that combines Abstract Syntax Tree (AST) analysis with Large Language Model (LLM) capabilities to produce test cases for embedded C firmware that properly execute within the commercial VectorCAST unit test tool framework. This paper examines the Sky Unit architecture and the AST indexed Retrieval Augmented Generation (RAG) pipeline which enables test generation for complex embedded systems firmware. In testing on a real-world medical device firmware project, Sky Unit produced executable tests for 76 out of 83 files and achieved 71.4% statement coverage and 63.9% branch coverage. These results demonstrate the effectiveness of Sky Unit in a Class III medical device software engineering environment.

Keywords: unit test generation, VectorCAST, LLMs, embedded C, retrieval augmented generation, medical device firmware

Glossary

AI – Artificial Intelligence
AST – Abstract Syntax Tree
CLI – Command Line Interface
LLM – Large Language Model
MC/DC – Modified Condition / Decision Coverage
OTSS – Off the Shelf Software
RAG – Retrieval Augmented Generation
VectorCAST – Commercial OTSS Unit Test Tool

Introduction

The medical device software development lifecycle as shown in Figure 1 and described in the international standard IEC 62304 (International Electrotechnical Commission [IEC], 2015) identifies testing a unit of software as an important part of an overall testing strategy and included in the testing process step. Moreover, technology companies that are not bound by standards such as IEC 62304 recognize unit testing. Per Amazon Web Services (n.d.), unit testing is defined as the process where you test the smallest functional unit of code and is identified as being useful for bug discovery and documentation.

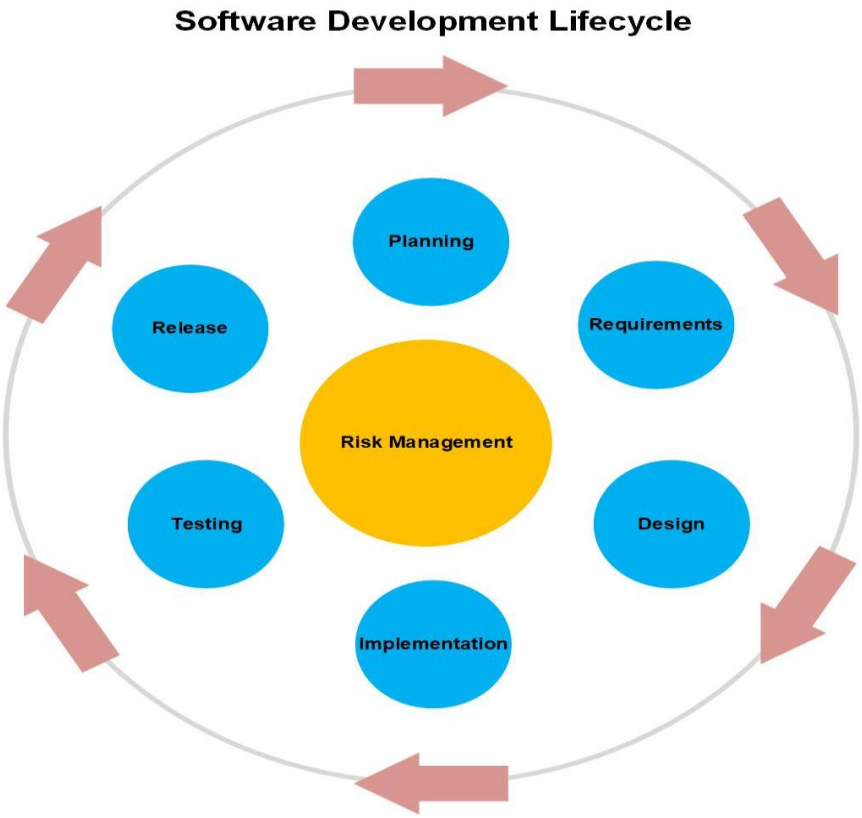

Figure 1. IEC 62304 Medical Device Software Development Lifecycle

In the domain of medical devices and especially embedded systems associated with implanted devices, the C programming language remains dominant (Beningo, 2024) at over 60% market share. Commercial testing tools are available from several vendors with various offerings and capabilities. Often, "automated testing" is a marketing ploy where the automation lies in the execution of test suites created by engineers rather than the automated creation of those suites from raw source code. However, the emergence of LLMs presents new opportunities for exploration in task automation to evaluate whether these AI language models can effectively generate tests from the source code.

This research presents Sky Unit, an AI-powered CLI tool component within an AI-augmented engineering ecosystem known as Blue Sky, that automatically generates, imports, validates, and iteratively fixes VectorCAST unit test scripts for embedded C firmware by combining LLM-based test generation with AST-indexed code context and deterministic post-processing. Unit testing embedded C code presents unique challenges: hardware-dependencies, complex firmware type systems with custom definitions,

pointer-heavy interfaces, etc. Traditional test generation approaches struggle with these complexities. Sky Unit addresses these challenges through a novel architecture that:

- Pre-indexes source code into a searchable AST database
- Extracts complete type systems
- Analyzes decision points using tree-sitter (open-source parse generator library) AST to extract conditions and generate test case suggestions
- Provides LLMs with rich contextual metadata to generate semantically meaningful tests
- Applies deterministic fixes to ensure syntactic correctness
- Validates through VectorCAST with iterative error correction

With this design in place this paper focuses on answering two research questions: 1) How effective is Sky Unit when applied to a medical-grade firmware project? 2) What limitations and failures emerge testing complex embedded firmware?

Related Work

The systematic literature review of LLM use in unit test generation presented by Zapkus and Slotkiene (2024) analyzed 37 articles from 2019 to 2023. Since then, two papers of note (Liu et al., 2025; Chowdhury et al., 2026) focused on unit test generation for the C programming language. Liu et al. (2025) presented STRUT, a novel unit test generator for C that utilizes structured test cases as a bridge to LLMs. Recognizing the challenge of automated unit test generation for C, Chowdhury et al. (2026) introduced SPARC, a neuro-symbolic, scenario-based framework that employs control flow graph analysis, an operation map, path targeted test synthesis, and self-correction loops.

Sky Unit is not just a research level prototype unit test generator. It is an end-to-end automated workflow. It combines AST aware code context extraction, branch coverage, VectorCAST-compatible script output, and iterative repair execution feedback loops. The environment-specific orchestration includes deterministic syntax/semantic fix layers, report parsing, and generated test artifacts that support an existing medical device unit test infrastructure. Table 1 compares Sky Unit with the two most closely related recent C language focused systems, namely STRUT (Liu et al., 2025) and SPARC (Chowdhury et al., 2026). Sky Unit differs across the following four dimensions:

1. AST-indexed vector RAG pipeline over the complete firmware type system
2. VectorCAST-compatible script output rather than generic test code
3. Multi-layer deterministic repair stage applied before validation
4. Validation loop driven by real import and execution feedback inside a regulated tool chain

Table 1. Feature Comparison of Sky Unit, STRUT, and SPARC

Dimension	**Sky Unit (this work)**	**STRUT (Liu et al., 2025)**	**SPARC (Chowdhury et al., 2026)**
Target language / domain	Embedded C firmware; medical device (IEC 62304)	C; industrial embedded and open-source	C; real-world and algorithmic subjects
Code-context method	AST-indexed vector RAG (tree-sitter + Azure Cognitive Search) with full type-system extraction and relevance filters	Structured seed test cases as the LLM bridge	Control-flow-graph analysis plus RAG

LLM used	Claude Sonnet 4.5 (AWS Bedrock) with AutoGen agents	GPT-4o	DeepSeek V3.2 (with LLM-portability analysis)
Test output format	VectorCAST-compatible test scripts	Executable C tests via SunwiseAUnit (rule-based conversion)	Standalone C unit test files
Validation and repair	VectorCAST import and execution feedback with iterative error correction plus multi-layer deterministic fixes	Compile and run with coverage-guided regeneration iteration	Iterative self-correction loop using compiler and runtime feedback
Production toolchain integration	Yes; deployable into an existing VectorCAST/IAR/ARM pipeline	Plugin to the SunwiseAUnit commercial tool	Research prototype
Evaluation scale	83 files / 5,609 generated tests (production firmware)	1,709 focal methods across multiple projects	59 projects
Reported results	91.2% test pass rate; 71.4% statement / 63.9% branch coverage	96.01% execution pass rate; 77.67% line / 63.60% branch coverage	+31.36% line and +26.01% branch coverage vs. baseline; 94.3% test retention

Research Methodology

The research methodology centers on a three-phase architecture approach. The AST index provides type-safe context that prevents LLM hallucinated identifiers, the relevance filters manage token budget constraints, and the VectorCAST validation loop ensures generated tests are both syntactically importable and semantically meaningful. The following subsections detail each phase.

Phase 1: Offline Structural Analysis of the Code Base (See Figure 2)

- Parse C language source files with tree-sitter.
- Extract enhanced type system:
    - Filename and path for source code identification
    - Full source code content in text for vector embedding
    - Functions with names, parameters, return type, static/inline qualifiers
    - Structs with bitfield support and array dimensions
    - Unions with field definitions
    - Enums with values
    - Typedef names and base type resolution
    - Macros
    - Globals with canonical metadata (name, type, size, is_static, is_const, is_array)
    - Include chains per file (transitive header resolution)
    - Call graphs and dependency relationships
- Upload to Azure Cognitive Search to create an AST index and support RAG.

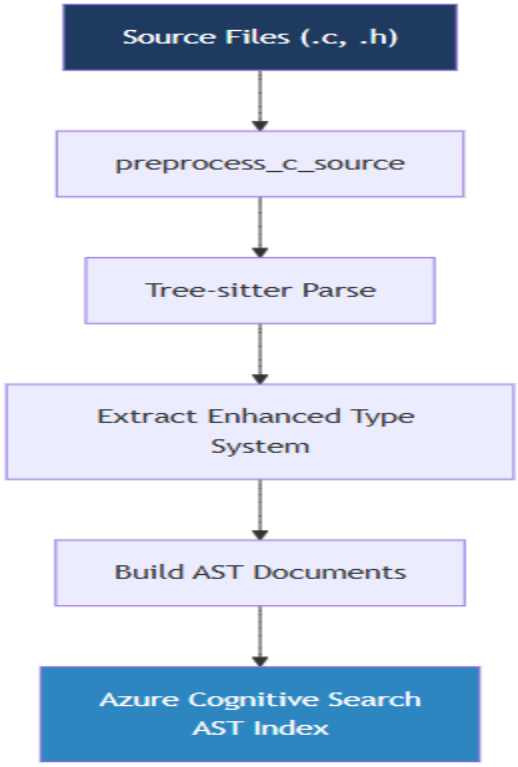

Figure 2. Phase 1 — Offline Structural Analysis of the Code Base

Phase 2: Online Context Assembly with Relevance Filtering (See Figure 3)

- Query AST search index for target file metadata
- Filter type system with relevance filters to fit token budget for context window sizing
- Extract function bodies with tree-sitter
- Construct LLM prompt with analysis, examples, syntax reference, and type system

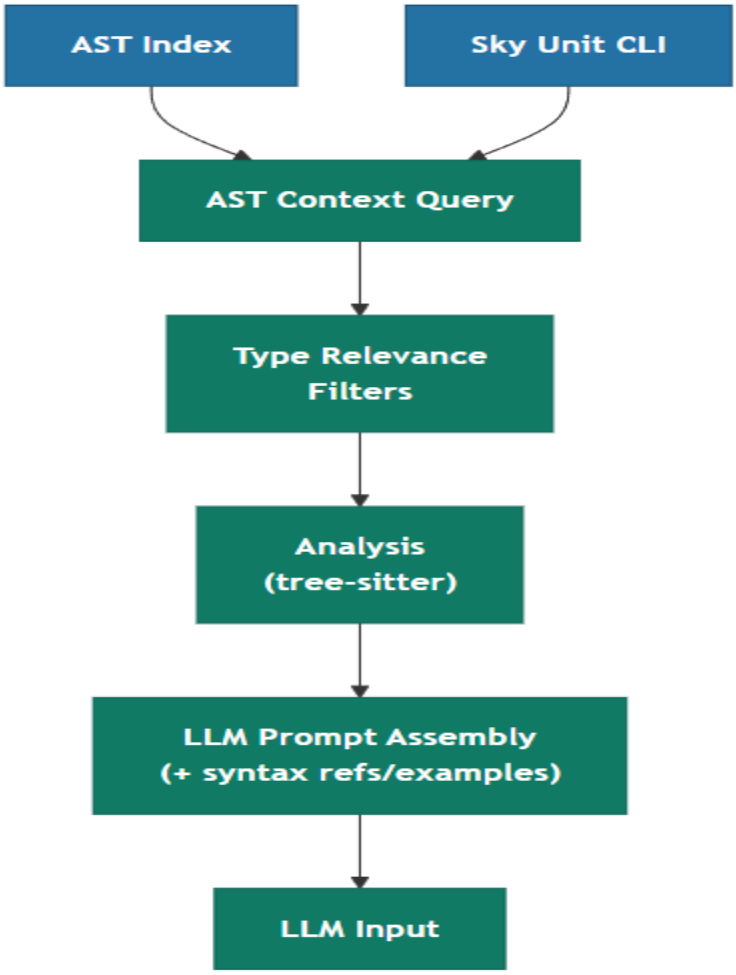

Figure 3. Phase 2 — Online Context Assembly with Relevance Filtering

Type System Filter Strategies

The full type system of a C file can be very large when you follow all the transitive headers it pulls in. Sky Unit runs on Claude Sonnet 4.5, which gives it a 200,000-token context window (the total amount of text, measured in tokens, that the LLM can read and reason over in a single request). That budget fills up fast, so before assembling each prompt Sky Unit applies four relevance filters that trim the type system down to what a given function actually needs. Each filter uses its own depth limits to stay within budget while still handing the LLM enough context to write meaningful tests.

Struct filter. This filter walks the struct dependency graph using a breadth-first approach, starting from the types a function actually touches. It seeds the walk from the function's parameters and return type and expands three levels deep. It seeds from any globals used by a function and expands two levels. For structs pulled in indirectly it keeps just the top-level fields to keep the context from blowing up.

Enum filter. An enum is included if its type name shows up in the function body or in the global type declarations. As a safety net, it is also included if any of its member values appear in the function body, which catches the common case where code uses an enum constant without ever naming the enum type.

Union filter. A union is included when its name appears in the function source, globals, or in the fields of a struct that already passed through the struct filter. This matters most for register/bit-field unions which are big enough on their own to overflow the context window.

Macro filter. This filter builds a set of word tokens from the function source and the global types and then matches macro names against that set on whole-word boundaries to prevent tripping on substrings. The number of #defines it pulls in is capped to keep the context window from overflowing.

All four filters share a common safeguard such that if a filter ends up with nothing, it falls back to returning the full unfiltered set. The driving rationale is that it is better to hand the LLM everything and risk some truncation rather than to strip out the type context it depends on entirely.

Phase 3: Iterative Generation with Validation Feedback (See Figure 4)

- Generate tests via LLM
- Apply multi-layer deterministic fixes including orchestrator-level accessor fixes
- Validate with VectorCAST to support both Linux and Windows environments
- Execute iterative feedback loop

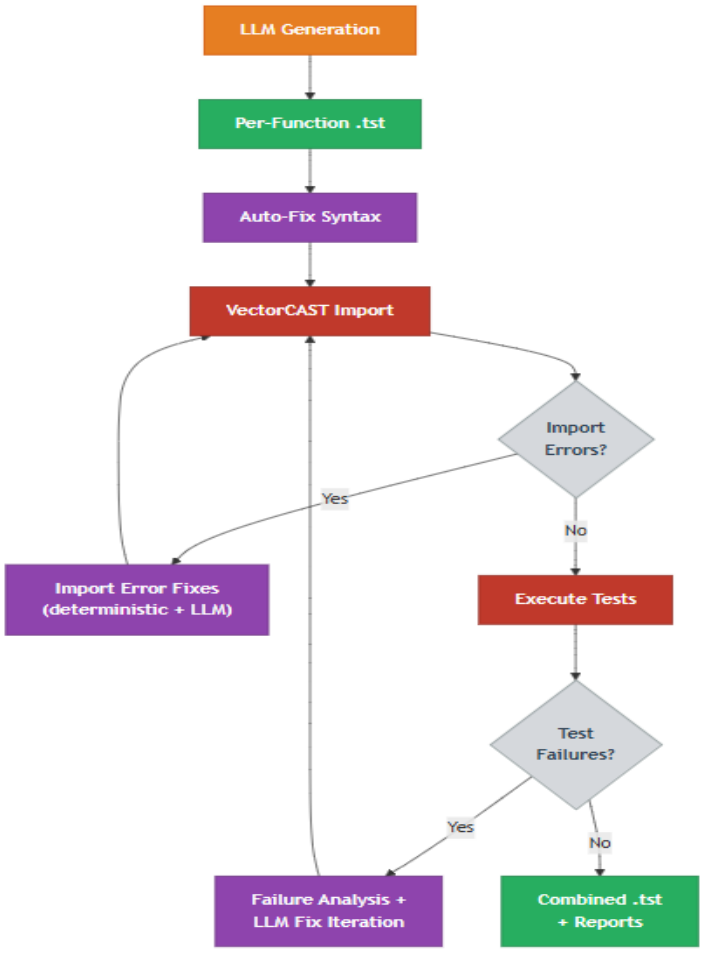

Figure 4. Phase 3 — Iterative Generation with Validation Feedback

The AST Search Index: Foundation of Code Understanding
The AST search index is a pre-computed database containing parsed representations of C source code. Unlike simple text search, the AST index understands code structure which results in better data for LLMs to digest. Below is a brief comparison of search capability examples that illustrate the difference:

Traditional text search:

- Finds the InputId string anywhere in source files
- Cannot distinguish whether ModuleConfigT is a variable name, a type, or a function
- No understanding of which functions call which other functions

AST index search:

- Knows InputId is a parameter of type InputIdT* belonging to module_init
- Knows ModuleConfigT is a struct with specific fields like timeout, count, and mode
- Knows the function module_init calls validate_input_id, which in turn calls check_range

When generating a test, the LLM needs to set up parameters, globals, and expected values using exact type information. This would include correct field names, valid enum values, proper pointer allocation, and accurate array sizes. The AST index provides all of this in a structured, queryable form. Without it, the LLM would have to guess at names and types from raw source text, leading to import errors and invalid test scripts.

Experimental Evaluation

Dataset
The evaluation dataset comprises a complete production application firmware code base that was developed under IEC 62304 (IEC, 2015). A total of 83 source files were targeted across 10 test batches. Of these, 7 files produced no usable results due to build failures, missing workspace data, or complete execution failure (File9, File18, File44, File46, File60, File69, File81), leaving 76 files that built and produced test execution results.

The dataset exhibits characteristics that are typical of C code in safety-critical embedded systems. There are extensive custom typedefs, pointer-heavy interfaces including void* parameters, hardware register abstractions using unions and bitfields, complex state machines, and deep include header chains producing large transitive type systems. File complexity ranges from simple utility functions (File59, 7 statements) to large modules (File4, 1,444 statements).

Experimental Setup
All test generation was performed using the configuration listed in Table 2.

Table 2. Experimental Configuration

Component	Configuration
LLM	Claude Sonnet 4.5 via AWS Bedrock
Context Window	200,000 tokens
Agent Framework	AutoGen 0.4+ (TestGenerator + ErrorAnalyzer agents)
AST Parser	Tree-sitter (C grammar)
Search Index	Azure Cognitive Search with text-embedding-3-large
Test Framework	VectorCAST (Windows execution)
Compiler	IAR iccarm.exe (ARM embedded target)
Coverage Type	Statement + Branch

The generation pipeline was executed in batch mode across 10 batches and selective batch re-runs. Each batch processed source files end-to-end with AST index queriers, type system extraction and filtering,

prompt assembly, LLM generation, deterministic post-processing, VectorCAST import validation with iterative error correction, and test execution with result parsing.

Metrics

The evaluation employs the following primary metrics:

Test Pass Rate (test-level): The ratio of individual test cases that pass execution to total test cases generated.

File Pass Rate (file-level): The proportion of source files where ALL generated test cases pass. Any single failure results in a FAIL classification.

Structural Coverage: Statement and branch coverage achieved by the generated test suite, as reported by VectorCAST.

Results and Discussion

Aggregate Results

Table 3 summarizes the aggregate results across the 76 files that built and ran successfully.

Table 3. Aggregate Results — Batches 1–10 (76 Built and Ran)

Metric	Value
Total Source Files	83 (76 built & ran)
Total Test Cases Generated	5,609
Test Cases Passing	5,114
Test Pass Rate	91.2%
Files with All Tests Passing	34 / 76
File Pass Rate	44.7%
Statements Covered (76 ran)	11,311 / 15,831
Statement Coverage (76 ran)	71.4%
Branches Covered (76 ran)	4,518 / 7,067
Branch Coverage (76 ran)	63.9%
Statement Coverage (all 83)	11,311 / 17,162 = 65.9%
Branch Coverage (all 83)	4,518 / 7,621 = 59.3%

Two pass-rate measures are reported, and they answer different questions. The test pass rate (test-level) is the fraction of individual generated test cases that execute and pass; it measures the correctness of the generated tests. The file pass rate (file-level) is the fraction of files in which every generated test passes; it is an all-or-nothing gate in which a single failing test reclassifies the entire file as FAIL. The two measures allow for a richer understanding of the metrics.

The 91.2% test pass rate shows that Sky Unit predominantly generates valid executable test cases. The 44.7% file pass rate reflects how often a file is completely free of any failing test. For example, File49 passes 487 of 491 tests (99.2%) yet is classified FAIL because of its 4 failures. The test pass rate is a better indicator of generation quality while the file pass rate is a readiness gate of file level completeness.

Test Pass Rate Distribution

Table 4 reports how the 76 files that built and ran are distributed across test-pass-rate bands.

Table 4. Test Pass Rate Distribution (76 Built and Ran)

Pass Rate Range	Files	% All	Representative
100%	34	44.7%	File17 (100%), File20 (100%), File80 (100%)
90–99%	22	28.9%	File38 (97.7%), File49 (99.2%), File82 (98.7%)
80–89%	8	10.5%	File4 (85.8%), File6 (89.6%), File7 (89.9%)
70–79%	7	9.2%	File30 (79.5%), File37 (77.2%), File53 (77.0%)
50–69%	4	5.3%	File10 (69.2%), File63 (63.4%), File76 (50.0%)
Below 50%	1	1.3%	File48 (44.1%)

73.7% of files (56/76) achieve 90% or higher test pass rates. 84.2% (64/76) achieve a rate of 80% or higher.

Coverage Analysis

Table 5 summarizes statement and branch coverage grouped by file pass status.

Table 5. Coverage Summary by File Pass Status

Category	Files	Statement Coverage	Branch Coverage
All Tests Pass (PASS)	34	2,920 / 3,618 (80.7%)	989 / 1,374 (72.0%)
Some Tests Fail (FAIL)	42	8,391 / 12,213 (68.7%)	3,529 / 5,693 (62.0%)
Built & Ran	76	11,311 / 15,831 (71.4%)	4,518 / 7,067 (63.9%)
All 83 files	83	11,311 / 17,162 (65.9%)	4,518 / 7,621 (59.3%)

Files with 100% test pass rate also tend to score higher on structural coverage which suggests the LLM writes more complete test suites when the AST index gives it a clear picture of the type system, and the function interfaces it is working with. 9 files reached 100% statement coverage: File28, File29, File31, File39, File50, File52, File59, File72, and File75.

It is important to understand what a PASS status conveys. It tells you that every test Sky Unit generated for that file ran and passed but does not tell you that those tests covered most of the code. So, a file can pass yet still post low coverage because Sky Unit only produced a small number of tests which happened to be valid. Some examples of this are found in File35 (6 of 6 tests passing, but only 7% statement coverage), File45 (1 of 1, 21%), and File27 (31 of 31, 51%). In each case every test passed while the test suite only covered a small slice of the entire file. The takeaway is that coverage and pass rate measure different things. Both need to be understood in context because you cannot infer one from the other.

Notable Results

File17 produced 142 test cases at a 100% pass rate, covering 96% of statements and 95% of branches. File80 was close behind with 138 test cases at a 100% pass rate, at 88% statement and 81% branch coverage. File49 generated 491 test cases and passed 487 of them (99.2%), reaching 90% statement and 82% branch coverage across 788 statements which are strong numbers for a suite that large.

File28 and File29 each hit 100% statement and 100% branch coverage with just four test cases for each. File52 did the same across 118 statements, though it needed 63 test cases to get there. Even the FAIL files do well. The 42 files classified as FAIL still passed 89.0% of their tests overall (3,992 of 4,487). Hence, FAIL generally means a few tests did not pass, which is better framing than stating the generated tests were bad.

Failure Analysis and Limitations

The failures we saw cluster into three main types.

1. **Hardware dependencies**. 7 files (File9, File18, File44, File46, File60, File69, and File81) produced nothing usable. All of them lean heavily on hardware such as registers, DMA controllers, or interrupt handlers which the LLM cannot realistically exercise on its own.
2. **Running out of room on complex logic.** Larger modules such as File4 (85.8%) and File24 (94.0%) tend to stumble on cases that require tracking multi-step state transitions, where the LLM struggles to reason about everything that has accumulated along the way.
3. **Stubbing limits.** Some tests fail because VectorCAST cannot stub an internal or static function. While Sky Unit manages these unstubbable functions, that knowledge cannot be applied retroactively to tests already generated earlier in the same batch.

Validity Concerns

Internal validity. Because the iterative fix loop is part of the pipeline, the final pass rates reflect more than just the LLM's first draft. These pass cases include deterministic fixes and LLM-based error correction. Reporting first-attempt rates separately from post-iteration rates would give a cleaner read on the raw quality of what the LLM generates.

External validity. Every result comes from a single product line, one compiler (IAR), and one test framework (VectorCAST) on a single ARM target. We have not tested other embedded platforms, compilers, RTOSes, or test frameworks. Therefore, we cannot assume these numbers carry over to other environments.

Construct validity. Statement and branch coverage are only proxies for test quality. These generated tests confirm what the LLM predicts the code will do, which is not necessarily what the requirements say it should do.

Conclusion

This paper presented Sky Unit which is an end-to-end system for automated unit test generation targeting embedded C firmware. By combining AST-indexed code analysis with LLM-based generation and multi-layer deterministic post-processing, Sky Unit generated 5,609 test cases across 76 production firmware files that built and executed successfully, achieving a 91.2% test pass rate and 71.4% statement coverage. An additional 7 files produced no usable results due to build failures or deep hardware dependencies. The file-level pass rate of 44.7% highlights remaining challenges, though 73.7% of files achieve 90%+ test pass rates. Future work includes MC/DC pairs coverage analysis, cross-platform evaluation, LLMs with larger context windows, and first-attempt vs post-iteration metrics to isolate generation quality from fix loop effectiveness. In a regulated context, the generated tests are intended to augment, not replace, engineer-authored verification. All output requires human review before acceptance as a regulated artifact.

Acknowledgements

The author thanks the following contributors to this work: Daniel Oyadeji (Software Engineer and Analyst), Eric Enrooth (Project Lead), and Paul Christensen (Engineering Manager).

References

Amazon Web Services. (n.d.). What is unit testing? https://aws.amazon.com/what-is/unit-testing/

Beningo, J. (2024, October). The best embedded programming languages for engineers now. https://www.beningo.com/the-best-embedded-programming-languages-for-engineers-now

Chowdhury, J., Fu, C., & Jabbarvand, R. (2026, February). SPARC: Scenario planning and reasoning for automated C unit test generation. arXiv. https://arxiv.org/abs/2602.16671

International Electrotechnical Commission. (2015). Medical device software—Software life cycle processes (IEC 62304:2006/Amd 1:2015). https://www.iso.org/standard/64686.html

Liu, J., Li, C., Chen, R., Li, S., Gu, B., & Yang, M. (2025, June). STRUT: Structured seed case guided unit test generation for C programs using LLMs. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3728970

Zapkus, D., & Slotkiene, A. (2024). Unit test generation using large language models: A systematic review. https://doi.org/10.15388/LMITT.2024.20

Appendix

Table 6 provides the raw coverage data for each of the 83 files referenced throughout the results.

Table 6. Raw Coverage Data (All 83 Files)

File	Testcases	Statement Coverage	Statement %	Branch Coverage	Branch %	Result
File1	24 / 24	54 / 56	96%	22 / 23	96%	PASS
File2	37 / 37	60 / 61	98%	24 / 28	86%	PASS
File3	70 / 75	224 / 258	87%	93 / 119	78%	FAIL
File4	326 / 380	1047 / 1444	73%	338 / 534	63%	FAIL
File5	68 / 71	176 / 208	85%	100 / 125	80%	FAIL
File6	251 / 280	418 / 571	73%	204 / 296	69%	FAIL
File7	98 / 109	238 / 347	69%	120 / 198	61%	FAIL
File8	15 / 15	7 / 9	78%	5 / 7	71%	PASS
File9	—	—	—	—	—	FAIL
File10	63 / 91	275 / 366	75%	98 / 173	57%	FAIL
File11	60 / 60	266 / 280	95%	80 / 101	79%	PASS
File12	71 / 77	100 / 141	71%	67 / 91	74%	FAIL
File13	38 / 46	93 / 157	59%	39 / 63	62%	FAIL
File14	46 / 46	171 / 180	95%	35 / 47	74%	PASS
File15	138 / 152	287 / 533	54%	98 / 243	40%	FAIL
File16	23 / 23	47 / 49	96%	32 / 33	97%	PASS
File17	142 / 142	406 / 424	96%	142 / 150	95%	PASS
File18	—	—	—	—	—	FAIL
File19	63 / 63	173 / 178	97%	74 / 81	91%	PASS
File20	105 / 105	250 / 292	86%	127 / 169	75%	PASS
File21	34 / 37	91 / 95	96%	60 / 87	69%	FAIL
File22	45 / 45	120 / 130	92%	46 / 54	85%	PASS
File23	34 / 35	102 / 119	86%	56 / 73	77%	FAIL
File24	233 / 248	527 / 749	70%	192 / 342	56%	FAIL
File25	13 / 13	46 / 48	96%	11 / 15	73%	PASS
File26	15 / 15	47 / 49	96%	19 / 21	90%	PASS
File27	31 / 31	27 / 53	51%	16 / 31	52%	PASS
File28	4 / 4	32 / 32	100%	1 / 1	100%	PASS

File29	4 / 4	36 / 36	100%	1 / 1	100%	PASS
File30	31 / 39	78 / 97	80%	27 / 36	75%	FAIL
File31	7 / 7	26 / 26	100%	3 / 3	100%	PASS
File32	22 / 23	43 / 52	83%	18 / 27	67%	FAIL
File33	10 / 10	83 / 84	99%	15 / 17	88%	PASS
File34	5 / 5	18 / 19	95%	5 / 7	71%	PASS
File35	6 / 6	14 / 209	7%	3 / 52	6%	PASS
File36	208 / 237	295 / 510	58%	167 / 308	54%	FAIL
File37	61 / 79	107 / 163	66%	67 / 115	58%	FAIL
File38	43 / 44	410 / 442	93%	37 / 61	61%	FAIL
File39	12 / 12	21 / 21	100%	7 / 7	100%	PASS
File40	40 / 44	108 / 113	96%	36 / 46	78%	FAIL
File41	31 / 33	88 / 105	84%	33 / 49	67%	FAIL
File42	9 / 9	36 / 47	77%	10 / 25	40%	PASS
File43	27 / 36	48 / 85	56%	20 / 46	43%	FAIL
File44	0 / 224	—	—	—	—	FAIL
File45	1 / 1	53 / 248	21%	2 / 83	2%	PASS
File46	—	—	—	—	—	FAIL
File47	15 / 15	12 / 13	92%	8 / 9	89%	PASS
File48	30 / 68	32 / 120	27%	16 / 71	23%	FAIL
File49	487 / 491	713 / 788	90%	336 / 409	82%	FAIL
File50	19 / 20	26 / 26	100%	16 / 17	94%	FAIL
File51	127 / 131	274 / 298	92%	123 / 148	83%	FAIL
File52	63 / 63	118 / 118	100%	41 / 41	100%	PASS
File53	97 / 126	180 / 274	66%	114 / 175	65%	FAIL
File54	343 / 356	546 / 1124	49%	261 / 401	65%	FAIL
File55	173 / 203	602 / 1217	49%	213 / 545	39%	FAIL
File56	63 / 66	62 / 64	97%	46 / 50	92%	FAIL
File57	106 / 111	163 / 231	71%	80 / 95	84%	FAIL
File58	36 / 36	73 / 84	87%	23 / 29	79%	PASS
File59	2 / 2	7 / 7	100%	2 / 3	67%	PASS
File60	—	—	—	—	—	FAIL
File61	7 / 11	7 / 9	78%	5 / 6	83%	FAIL
File62	38 / 39	87 / 113	77%	24 / 49	49%	FAIL
File63	26 / 41	30 / 78	38%	13 / 40	32%	FAIL
File64	51 / 65	99 / 151	66%	47 / 77	61%	FAIL
File65	63 / 66	167 / 172	97%	84 / 98	86%	FAIL
File66	60 / 60	48 / 57	84%	34 / 41	83%	PASS
File67	44 / 61	101 / 225	45%	19 / 46	41%	FAIL
File68	52 / 64	131 / 136	96%	48 / 54	89%	FAIL
File69	—	—	—	—	—	FAIL
File70	26 / 26	61 / 80	76%	18 / 34	53%	PASS
File71	43 / 43	115 / 118	97%	30 / 37	81%	PASS
File72	8 / 8	5 / 5	100%	2 / 2	100%	PASS
File73	36 / 46	90 / 125	72%	33 / 51	65%	FAIL
File74	37 / 41	72 / 74	97%	45 / 52	87%	FAIL
File75	15 / 15	13 / 13	100%	4 / 4	100%	PASS
File76	14 / 28	40 / 95	42%	12 / 41	29%	FAIL
File77	68 / 72	91 / 126	72%	61 / 82	74%	FAIL
File78	147 / 167	222 / 316	70%	121 / 209	58%	FAIL
File79	38 / 38	80 / 118	68%	25 / 44	57%	PASS
File80	138 / 138	305 / 344	89%	108 / 133	81%	PASS
File81	—	—	—	—	—	FAIL

File82	77 / 78	175 / 194	90%	65 / 93	70%	FAIL
File83	1 / 1	90 / 130	69%	14 / 41	34%	PASS