Random Test Data Generation Iteration 1 Report

Started 25/02/2026
Maksym Redchenko C00302240
SETU

Iteration Period: Weeks 3–6 (February – March 2026)

1. Changes to the Vision

The core vision of the project — a free, open-source, VS Code-native extension that generates Google Test boundary value tests for C functions on Linux — has remained unchanged during Iteration 1. The extension still targets the identified gap in the market: no free, VS Code-native tool generates Google Test code with boundary focus for C developers.

However, two notable expansions were made to the original vision:

1. **Cross-platform support**: The Vision Document initially specified Ubuntu Linux as the primary and sole target platform. During implementation, the extension was expanded to support **Windows, Linux, and WSL (Windows Subsystem for Linux)** environments. The `BuildRunner` module (`src/build/buildRunner.ts`) includes automatic environment detection and path conversion logic for all three platforms. This broadens the user base beyond the original "Linux-First" scope without compromising the Linux-first commitment.
2. **Global variable awareness**: The original vision described the extension as analysing "C function signatures in header files." During Iteration 1, the scope was expanded to include **global variable detection and testing**. Functions that reference global (non-const) variables now receive additional test cases that vary those globals across boundary values. This was not explicitly planned in the Vision Document but emerged as a natural extension of the boundary value analysis approach during development.

No features were removed from the vision.

2. Changes to the Specification

2.1 Additional Requirements Implemented

The following requirements were added beyond what was originally planned for Iteration 1:

#	Requirement	Rationale
1	**Global variable extraction and analysis** — The parser detects global variables (`GlobalExtractor`) and determines which ones a function uses (`GlobalUsageAnalyzer`)	Provides more comprehensive boundary coverage; global state is a common source of boundary-related bugs

2	**Interactive Webview UI for expected values** — A rich HTML-based webview panel (`ExpectedValuesWebview`) for entering expected test results, replacing the originally envisioned simple input boxes [4]	Significantly improves usability; developers can see all test cases at once and fill values in any order
3	**Custom test creation** — Users can add their own custom test cases through the webview UI, specifying parameter values and expected results	Empowers developers to supplement auto-generated boundary tests with domain-specific cases
4	**CMake build system generation** — Automatic `CMakeLists.txt` creation (`CMakeGenerator`) alongside test files [5]	Eliminates manual build configuration; original vision mentioned `g++` with `-lgtest` flag, but CMake proved more robust and portable
5	**High parameter count warnings** — Functions with 5–7 parameters show an info message; functions with 7+ parameters show a modal warning with refactoring advice	Prevents test explosion and guides developers toward better C design patterns
6	**Keyboard shortcut** — `Ctrl+Shift+T` (`Cmd+Shift+T` on Mac) for instant test generation	Not specified in original iteration plan but essential for developer ergonomics
7	**WSL path conversion** — Automatic detection and conversion of Windows WSL network paths to Unix paths	Enables seamless development on Windows with WSL

2.2 Requirements Deferred

The following originally planned features were deferred to later iterations:

#	Requirement	Reason for Deferral	Target Iteration
1	**Float/double boundary testing**	While the boundary catalog includes `float` and `double` entries in `boundaryValues.ts`, full float-specific testing (divide-by-zero, infinity cases, `FLT_MAX / 0.1` as described in the Research Document) requires additional assertion types beyond `EXPECT_EQ`	Iteration 2
2	**Header file parsing**	The Vision Document mentioned analyzing "C function signatures in header files." The current implementation parses `.c` source files	Iteration 2

		with function definitions, not `.h` header declarations	

2.3 Further Details for Existing Requirements

- **C Function Signature Parser**: The specification was refined significantly. Instead of using regex (as initially considered and then rejected in the Research Document, Question 4), the implementation uses **Tree-sitter** (`web-tree-sitter` WASM module) to build a proper Abstract Syntax Tree (AST) of the C source code. This is a more robust approach than what was originally described and aligns with the research finding that "regex is not reliable for full C syntax."
- **Boundary Value Test Generator**: The specification was expanded to include proper BVA **with four boundaries per type**: `min`, `min+1`, `max-1`, `max`, plus a `zero / nominal` value. This follows standard boundary value analysis methodology as referenced in the Research Document [2]. The original examples only showed `0`, `INT_MAX`, `INT_MIN`, and overflow — the implementation adds the adjacent boundary values (`min+1`, `max-1`) for more thorough coverage.
- **Google Test Code Generation**: Tests are generated using the `extern "C"` pattern as described in the Research Document (Question 6), with the test file written as `.cpp` while importing the C source file. The generated code follows the Arrange-Act-Assert pattern, which was not originally specified but is a testing best practice.
- **Build and Execution Support**: The original specification mentioned invoking `g++` with the `-lgtest` flag. The implementation instead uses **CMake** as the build system, which provides better dependency management, cross-platform support, and handles Google Test library linking automatically through `find_package(GTest REQUIRED)`.

3. Changes to the Research and Design

3.1 Additional Research

The following research was conducted during Iteration 1 beyond what was documented in the initial Research and Design Document:

1. **Tree-sitter as a parsing solution**: Research into Tree-sitter [1] revealed it as the ideal middle ground between regex (too fragile) and a full compiler front-end (too heavy). Tree-sitter provides incremental parsing, supports C grammar via a WASM module, and is already used by VS Code internally for syntax highlighting. The `web-tree-sitter` npm package (v0.26.5) was selected for integration.
2. **VS Code Webview API**: Research into the VS Code Webview API was needed for the interactive expected values panel. The Webview provides a sandboxed HTML/CSS/JS environment within VS Code, enabling a richer input experience than sequential `showInputBox` dialogs.
3. **CMake integration over raw g++**: Research showed that CMake is the de facto standard for C/C++ build management and is recommended by the Google Test documentation [3] for incorporating GTest into projects. Using CMake instead of raw `g++` invocations provides better portability and dependency resolution.
4. **WSL file system access from Windows**: Research into how VS Code accesses WSL file systems (via `\\wsl.localhost\` or `\\wsl$\` network paths) was needed to support the cross-platform build pipeline.

3.2 Additional Design Decisions

1. **Modular architecture**: The codebase follows the "separation of concerns" principle outlined in the Vision Document's Maintainability section:
   - `src/parser/` — Function extraction (`functionExtractor.ts`), global variable extraction (`globalExtractor.ts`), and usage analysis (`globalUsageAnalyzer.ts`)
   - `src/generator/` — Test code generation (`testGenerator.ts`), boundary value catalog (`boundaryValues.ts`), and CMake configuration (`cmakeGenerator.ts`)
   - `src/build/` — Build system runner (`buildRunner.ts`)
   - `src/ui/` — Webview for expected values (`expectedValuesWebview.ts`)
   - `src/types.ts` — Shared TypeScript interfaces (`FunctionInfo`, `FunctionParameter`, `GlobalVariable`)
2. **"No Behaviour Guessing" principle**: A critical design decision was made that the tool should **never guess** expected return values. Generated tests use `FAIL() << "Expected value needed. Got: " << result;` as a placeholder, requiring the developer to provide expected values. This avoids false confidence in auto-generated assertions and is noted explicitly in the `testGenerator.ts` header comment.
3. **Single-function workflow**: Rather than parsing an entire file and generating tests for all functions at once, the extension focuses on **one function at a time** based on cursor position. This follows a more practical developer workflow and avoids overwhelming the user with thousands of test cases.

4. What Was Implemented and What Was Not

4.1 Successfully Implemented

Component	Description	Status
Tree-sitter C Parser	Full AST-based parsing of C function definitions, extracting return types, function names, and parameter lists	✅Complete
Boundary Value Catalog	Comprehensive boundary values for int, unsigned int, long, unsigned long, short, unsigned short, char, unsigned char, float, double types, with min, min+1, max-1, max, and zero boundaries	✅Complete
Test Generator	Google Test .cpp file generation with proper extern "C" linkage, Arrange-Act-Assert pattern, and FAIL() placeholders	✅Complete
CMake Generator	Automatic CMakeLists.txt creation with GTest dependency resolution and build instructions	✅Complete
Build & Run Pipeline	Full CMake configure → build → execute pipeline with VS Code output channel integration	✅Complete

Expected Values Webview	Rich HTML/CSS/JS webview for interactive expected value entry, with support for overflow, undefined, and skip responses	✓Complete
Custom Test Cases	Ability to create additional user-defined test cases via the webview	✓Complete
Global Variable Detection	Extraction of global variables from the AST and analysis of which functions reference them	✓Complete
Context Menu Integration	Right-click "Generate Tests for This Function" command, registered for .c files only	✓Complete
Keyboard Shortcut	Ctrl+Shift+T binding for quick test generation	✓Complete
Cross-Platform Build Support	Windows, native Linux, and WSL environment detection with appropriate build pipelines	✓Complete
Clean Build Command	Command to clean the build/ directory	✓Complete
Unsaved File Detection	Prompt to save unsaved changes before test generation	✓Complete

4.2 Not Implemented / Deferred

Component	Original Plan	Reason	Target
Array boundary testing	Planned for Iteration 2	On track — not part of Iteration 1 scope	Iteration 2
Structure field boundary generation	Planned for Iteration 2	On track — not part of Iteration 1 scope	Iteration 2
Pointer support	Planned for Iteration 2	On track — not part of Iteration 1 scope	Iteration 2
Header file (.h) parsing	Mentioned in Vision Document	Implementation focused on .c files with function definitions; header-only parsing deferred	Iteration 2
Configuration file for custom boundaries	Planned for Iteration 2	On track — not part of Iteration 1 scope	Iteration 2
Extension unit tests	Ongoing	Only a sample test exists in src/test/extension.test.ts; comprehensive extension testing needed	Iteration 2–3

4.3 Iteration 1 Summary Metrics

- **Total TypeScript source files**: 8 (extension.ts, types.ts, functionExtractor.ts, globalExtractor.ts, globalUsageAnalyzer.ts, testGenerator.ts, cmakeGenerator.ts, boundaryValues.ts, buildRunner.ts, expectedValuesWebview.ts)
- **Supported C types for boundary testing**: 10 (int, unsigned int, long, unsigned long, short, unsigned short, char, unsigned char, float, double)
- **VS Code commands registered**: 3 (Generate Tests, Build & Run, Clean Build)
- **Extension version**: 1.0.0

5. Use of Gen-AI Tools

Generative AI tools were used during Iteration 1 in the following ways:

5.1 GitHub Copilot (VS Code Extension)

- **Code autocompletion**: Copilot was used extensively for TypeScript autocompletion within VS Code during development. It provided helpful suggestions for boilerplate code such as interface definitions, error handling patterns, and repetitive boundary value catalog entries.
- **Comment-to-code generation**: JSDoc/TSDoc comments were written first, and Copilot generated initial implementations of functions like `extractFunctions()`, `parseFunctionDefinition()`, and `generateBoundarySets()`. These were then reviewed, tested, and refined manually.
- **Regex pattern generation**: The test-name-specific regex pattern in `fillExpectedValues()` (lines 479–484 of `extension.ts`) was initially drafted with Copilot assistance and then debugged and refined manually.

5.2 ChatGPT / GitHub Copilot Chat

- **Research acceleration**: Used to quickly understand Tree-sitter API concepts, VS Code Webview API patterns, and CMake syntax for GTest integration. All information was verified against official documentation before use.
- **Debugging assistance**: When encountering issues with Tree-sitter WASM loading in the VS Code extension host, AI chat tools helped identify the correct `locateFile` callback pattern for the `web-tree-sitter` initialisation.

5.3 Important Notes on AI Usage

- All AI-generated code was **reviewed, tested, and understood** before inclusion in the project.
- AI tools were used as **productivity accelerators**, not as a replacement for understanding. Every design decision (architecture, "no behaviour guessing" principle, single-function workflow) was made by the developer.
- The core algorithms — boundary set generation with combinatorial logic, AST traversal for function extraction, and test-name-specific regex replacement — required significant **manual design and debugging** beyond what AI tools could provide.

6. Revised Iteration Plan

Iteration 1: Foundation ✓(Weeks 3–6) — COMPLETED

All planned deliverables achieved plus additional features (global variable support, webview UI, cross-platform build).

Iteration 2: Complex Types (Weeks 7–10) — REVISED

Goal: Array, structure, and pointer support with enhanced execution pipeline

Revised Deliverables

- Array boundary testing: empty (size=0), single element, maximum size, out-of-bounds index
- Structure field boundary generation with type-specific limits for each field
- Basic pointer support (`int*`, `float*`, `NULL` testing)
- Header file (`.h`) parsing support — parse function declarations (not just definitions)
- Float/double-specific assertions (infinity checks with `EXPECT_TRUE(isinf(...))`, NaN checks)
- Configuration file (`.json`) for custom boundary values per project
- Comprehensive extension unit tests (target: 80%+ coverage of parser and generator modules)
- Coverage reporting (number of tests passed/failed with colour-coded output)

Changes from Original Plan

- Added header file parsing (moved from implicit assumption to explicit deliverable)
- Added float-specific assertion support (originally implicit in Iteration 1)
- Added extension unit testing (critical for reliability before Iteration 3 release)
- Removed "automated test pipeline" as a deliverable (already completed in Iteration 1)

Iteration 3: Production Polish (Weeks 11–14) — UNCHANGED

Goal: Marketplace-ready extension

Deliverables

- Error recovery with helpful messages for malformed C code
- VS Code settings panel (boundary customisation, output paths)
- Custom test template system for user-defined formats
- Performance optimisation: 1000+ tests in under 2 seconds
- Comprehensive documentation: installation guide, user manual, API docs
- Beta testing with 5+ users and feedback incorporation
- VS Code Marketplace package (`.vsix`) published

7. References

[1] Tree-sitter (2026) Tree-sitter — An incremental parsing system. Available at: https://tree-sitter.github.io/tree-sitter/ [Accessed: February 2026].
[2] Wikipedia (2026) 'Boundary-value analysis', Wikipedia. Available at: https://en.wikipedia.org/wiki/Boundary-value_analysis [Accessed: 28 January 2026].
[3] Google (2025) googletest/README.md at main, GitHub. Available at: https://github.com/google/googletest/blob/main/googletest/README.md [Accessed: 28 January 2026].

[4] Microsoft (2026) VS Code Extension API — Webview. Available at: https://code.visualstudio.com/api/extension-guides/webview [Accessed: February 2026].
[5] CMake (2026) CMake Documentation. Available at: https://cmake.org/documentation/ [Accessed: February 2026].