Random Test Data Generation Vision Document

Started 20/01/2026
Maksym Redchenko C00302240
SETU

1. Project Overview

Random Test Data Generation is an extension for Visual Studio Code [1]. It helps automate making Google Tests [2] for C and C++ functions, and it works on Ubuntu Linux [3]. Developers spend 40% to 60% of their time just writing unit tests manually. And often happens that you miss those boundary conditions, which cause about 70% of the defects that show up in production. That is a big waste.

The extension analyzes C function signatures in header files and then generates test cases automatically. It covers stuff like integer boundaries, such as: 0, INT_MAX, INT_MIN, and for floating points, it's FLT_MAX and infinity. Also, array edge cases and fields in structures. It can be considered pretty comprehensive. Compared to other tools, this one is free and open source. Similar applications like Cantata AutoTest [4] or Parasoft [5] C/C++test, those cost a fortune every year. This integrates right into VS Code with context menu commands, no extra hassle.

The project is three iterations in an agile way. The first one handled int and float types. Then iteration two added arrays and structures. And the third polished it up for release on the VS Code Marketplace. It fills this gap, since there's no other free tool that's native to VS Code for automated Google Test generation, especially with the focus on boundaries. Reduces test writing time by 80% and gets over 95% coverage.

2. Main Functionalities

- C Function Signature Parser

- A C Function Signature Parser scans C header files (.h) and extracts each function’s return type, name, and parameters so the tool knows what to generate tests for (including common primitives, arrays, pointers, and simple structs).

Small example header (math_utils.h) - figure 1.

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);                // primitives
float avg(int arr[], int size);       // array + size
double area(const struct Point* p);   // pointer to struct

struct Point { int x; int y; };       // basic structure

#endif
```

Figure 1 - Header example (math_utils.h)

Example “parsed output” (what the tool extracts) - figure 2.

```
Function: add
Return: int
Params: (int a, int b)

Function: avg
Return: float
Params: (int arr[], int size)

Function: area
Return: double
Params: (const struct Point* p)

```

Figure 2 - Parsed Output

- Boundary Value Test Generator

- Integers: Zero (0), maximum (INT_MAX), minimum (INT_MIN), overflow (INT_MAX + 1)
- Floats: Zero (0.0), maximum (FLT_MAX), minimum (FLT_MIN), infinity cases (FLT_MAX / 0.1)
- Arrays: Empty array (size = 0), single element, maximum size boundary
- Structures: Zero-initialized fields, extreme values per field type

- Google Test Code Generation

- It would automatically create a small Google Test file that calls add() function with important edge-case inputs (like 0, the biggest int, and the smallest int) and checks the results. This saves from writing repetitive test code by hand and helps catch boundary-related bugs early. Below, in Figure 3, you can find an example of a generated test file after calling the function.

```cpp
#include <gtest/gtest.h>
#include <limits>
#include "math_utils.h"

TEST(AddTest, ZeroValues) {
    EXPECT_EQ(add(0, 0), 0);
}

TEST(AddTest, BoundaryValues_IntMax) {
    int maxInt = std::numeric_limits<int>::max();
    EXPECT_EQ(add(maxInt, 0), maxInt);
}

TEST(AddTest, BoundaryValues_IntMin) {
    int minInt = std::numeric_limits<int>::min();
    EXPECT_EQ(add(minInt, 0), minInt);
}

```

Figure 3 - Generated test file

- Build and Execution Support

- The extension automates the build process by invoking g++ (the GNU command-line compiler for C++) [6]. It applies the -lgtest flag, which instructs the linker to include the Google Test library [1]. This ensures the final executable can access the testing framework while meeting the C++17 standards required to avoid linker errors. [11]
- Links Google Test libraries automatically
- Executes test binary and captures output
- Displays pass/fail results in VS Code terminal with color coding
- Reports number of tests passed vs. failed

3. Non-Functional Characteristics

Performance

- Generate more than 1000 test cases in less than 2 seconds
- Analyzes processes typical C header files (e.g., 500 lines) in less than 500ms
- Extension activation time is less than 1 second on VS Code startup
- Compilation and execution are complete within 5 seconds for standard projects
- Rationale: Maintains developer workflow without interruption

Usability

- No configuration needed - works right after installation
- Context-menu driven interface (right-click to generate)

- Clear error messages with actionable suggestions
- Intuitive command names in Command Palette
- Rationale: Reduces the entrance level for new users and students

Reliability

- 99% analysing success rate on standard C function signatures
- Checks for badly formed headers with error recovery
- Does tests compile without errors on the first run
- Consistent results across different Ubuntu distributions (20.04, 22.04, 24.04)
- Rationale: Guarantees predictable behavior across different projects

Extensibility

- Plugin architecture for custom test generators
- Configuration API for user-defined boundary values
- Template system for alternative test frameworks (future: Catch2 [7], Unity [8])
- Export/import test generation profiles
- Rationale: Allows future improvement and community contributions

Security

- No network access - all processing local
- No external API calls or data transmission
- Processes files only within the workspace directory
- Does not store or log sensitive code information
- Rationale: Protects exclusive source code and intellectual property

Maintainability

- Modular codebase with separation of concerns (parser/generator/UI)
- Comprehensive unit tests for core functionality (80%+ coverage)
- TypeScript [9] with strict typing for VS Code extension
- Documented API for parser and generator modules
- Version control with semantic versioning
- Rationale: Facilitates future development, debugging, and community contributions

4. Target Users

Primary:

- C/C++ developers on Linux
- Embedded systems engineers
- Computer science students
- Open-source contributors

Secondary:

- QA engineers testing C libraries
- Technical leads implementing test coverage
- University lecturers teaching unit testing

User Profile: Familiar with C/C++ and VS Code, works on Ubuntu Linux, values test automation

5. Platform

Development Stack:

- Visual Studio Code 1.80+
- Node.js [10] 18+ (extension runtime)
- TypeScript (extension language)

Target Environment:

- Ubuntu Linux 24.04+(latest 25.1)
- GCC/G++ compiler toolchain
- Google Test 1.10+(latest 1.17)

Deployment:

- VS Code Marketplace
- GitHub repository
- .vsix package format

6. Business Model Canvas

Key Partners	Key Activities	Value Propositions	Customer Relationships	Customer Segments

<table>
<tr>
<td rowspan="3">- Open source community
- VS Code ecosystem
- University
- GitHub</td>
<td>- Parse C/C++ code for function signatures
- Generate boundary values automatically
- Create and compile Google Test files
- Develop and maintain VS Code extension</td>
<td rowspan="3">- Save 70% of manual testing time
- Learn boundary testing principles
- Lower barrier to writing unit tests</td>
<td>- Self-service via marketplace
- GitHub issues for support
- Documentation and tutorials
- Open source community</td>
<td rowspan="3">- Primary: CS students learning C programming
- Secondary: Software testing educators</td>
</tr>
<tr>
<td>Key Resources</td>
<td>Channels</td>
</tr>
<tr>
<td>- Google Test framework
- TypeScript/Node.js stack
- Ubuntu testing environment
- Developer time (15 weeks)</td>
<td>- VS Code Extension Marketplace
- GitHub repository
- University presentations
- Technical blog posts</td>
</tr>
<tr>
<td colspan="3">Cost</td>
<td colspan="2">Revenue Streams</td>
</tr>
<tr>
<td colspan="3">- Development time (15 weeks)
- Testing infrastructure hosting
- VS Code marketplace fee ($5)
- Documentation production
- Ongoing maintenance</td>
<td colspan="2">- Academic project (no revenue):
- Portfolio value for employment
- Skill development and learning
- Future commercial potential:
- Basic model (advanced features $5/month)
- One-time purchase ($15-30)
- Enterprise licensing with custom features</td>
</tr>
</table>

7. Context Diagram



Figure 4 - Context Diagram Illustration

8. Main Project Risks

Technical Risks

C Parser Complexity

Parsing C code with all the pointers, macros, and preprocessor stuff is super tricky and key to making things work.

Boundary Value Generation Accuracy

Complicated types like structs and pointers might end up producing bad test cases.

Limited Parser Coverage

The parser won't understand every part of C (like function pointers, or inline asm).

Integration Risks

VS Code Extension API Changes

If the API changes, our extension might break.

Google Test Version Compatibility

Different versions might mess with the generated tests.

Environment *Risks*

Ubuntu-Specific Build Issues

Builds could act weird on different Ubuntu setups.

Project *Management* *Risks*

Time Constraints

We've only got 15 weeks, and there's a lot to do.

Critical Success Factor:

The C parser is basically 70% of the whole project's value - so we're planning to spend 40% of Iteration 1 just on making it solid.

9. Iteration Plan

Iteration 1: Foundation (Weeks 3-6)

Goal: Working parser with basic int/float test generation

Deliverables:

Iteration 1 should let a user right- click a simple C function (e.g., add) and automatically generate a runnable Google Test file with basic boundary tests.

- C function example in Figure 4.

```
1  // math_utils.h (input)
2  int add(int a, int b);
```

Figure 4 - C function example

- Generated cpp file example in Figure 5.

```cpp
// test_math_utils.cpp
#include <gtest/gtest.h>
#include <limits>
#include "math_utils.h"

TEST(AddTest, ZeroValues) {
  EXPECT_EQ(add(0, 0), 0);
}

TEST(AddTest, Boundary_IntMax) {
  EXPECT_EQ(add(std::numeric_limits<int>::max(), 0), std::numeric_limits<int>::max());
}

TEST(AddTest, Boundary_IntMin) {
  EXPECT_EQ(add(std::numeric_limits<int>::min(), 0), std::numeric_limits<int>::min());
}
```

Figure 5 - Generated cpp file

- Output in the terminal in Figure 6.

```
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from AddTest
[ RUN      ] AddTest.ZeroValues
[       OK ] AddTest.ZeroValues (0 ms)
[ RUN      ] AddTest.Boundary_IntMax
[       OK ] AddTest.Boundary_IntMax (0 ms)
[ RUN      ] AddTest.Boundary_IntMin
[       OK ] AddTest.Boundary_IntMin (0 ms)
[----------] 3 tests from AddTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
```

Figure 6 - Terminal output

Iteration 2: Complex Types (Weeks 7-10)

Goal: Arrays, structures, and automated execution

Deliverables:

- Array boundary testing: empty (size=0), single element, maximum size, out-of-bounds
- Structure field boundary generation with type-specific limits
- Basic pointer support (int*, float*)
- Automated test pipeline: compile with g++ → execute binary → capture output
- VS Code terminal integration with color-coded pass/fail results
- Coverage reporting (tests passed/failed counts)
- Configuration file for custom boundary values

Example: Parse int findMax(int arr[], int size) → Generate array boundary tests with null/empty/overflow cases

Technical Focus: Complex type parsing, process execution in Node.js, output parsing, JSON configuration

Iteration 3: Production Polish (Weeks 11-14)

Goal: Marketplace-ready extension

Deliverables:

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

Example: Handle syntax errors gracefully: "Line 42: Expected ';' after parameter" instead of crashing

Technical Focus: Error handling, VS Code settings API, template engine, performance profiling, user documentation, and marketplace.

10. References:

[1] Microsoft (2026) *Visual Studio Code*. Available at: https://code.visualstudio.com/ [Accessed: 29 January 2026]

[2] Google (2026) *GoogleTest User's Guide*. GoogleTest (GitHub Pages). Available at: http://google.github.io/googletest/ [Accessed: 29 January 2026]

[3] Canonical (2026) *Enterprise Open Source and Linux | Ubuntu*. Ubuntu. Available at: https://ubuntu.com/ [Accessed: 29 January 2026]

[4] QA Systems (2025) *Cantata AutoTest (Feature Brief)*. QA Systems. Available at: https://www.qa-systems.com/resource/cantata-feature-brief-autotest/ [Accessed: 29 January 2026]

[5] Parasoft (2026) *Static Code Analysis Tool – Optimize C and C++*. Parasoft. Available at: https://www.parasoft.com/products/parasoft-c-ctest/c-c-static-analysis/ [Accessed: 29 January 2026]

[6] GNU (2026) *C++ Support in GCC*, *GNU Project*. Available at: https://gcc.gnu.org/projects/cxx-status.html [Accessed: 29 January 2026]

[7] Catch2 (2026) *Catch2: A modern, C++-native, test framework for unit-tests*, *GitHub*. Available at: https://github.com/catchorg/Catch2 [Accessed: 29 January 2026].

[8] Unity (2026) *Unity: Unit Testing for C*, *ThrowTheSwitch.org*. Available at: http://www.throwtheswitch.org/unity [Accessed: 29 January 2026].
[9] TypeScript (2026) *TypeScript: JavaScript with syntax for types*, *Microsoft*. Available at: https://www.typescriptlang.org/ [Accessed: 29 January 2026].
[10] Node.js (2026) *Node.js — Run JavaScript Everywhere*, *OpenJS Foundation*. Available at: https://nodejs.org/ [Accessed: 29 January 2026].
[11] Google (2025) *GoogleTest README: Generic Build Instructions*. Available at: https://github.com/google/googletest/blob/main/googletest/README.md [Accessed: 5 February 2026].