University of Alberta

ECE 322 – Software Requirements and Testing

Lab 5 Report

Using Large Language Models for Test Design

Name:	Diepreye Charles-Daniel
CCID:	DIEPREYECD
Lab Section:	D22
Instructor:	Witold Pedrycz
Date:	December 2025

Task 1 – Drone Program (LLM-Generated Black-Box Tests)

Prompt Used

```
Attached is the drone autopilot program from my software testing lab.
It reads three integer inputs x1, x2, x3 representing distances to waypoints
and outputs either "success", "failure", or an error message.

Please design a set of black-box test cases for this program.
```

LLM-Generated Test Cases

ID	x_1	x_2	x_3	Expected	Description / Rationale
TC-01	10	20	30	success	Nominal case: valid positive integers with sum $60 < 100$.
TC-02	0	0	0	success	Lower boundary: minimum non-negative values; sum is 0.
TC-03	33	33	34	success	Upper boundary: sum is exactly $100 = k$.
TC-04	100	0	0	success	Upper boundary (single leg): one leg uses entire range; sum is 100.
TC-05	33	34	34	failure	Just above boundary: sum is $101 > 100$.
TC-06	101	0	0	failure	Just above boundary (single leg): sum is $101 > 100$.
TC-07	50	50	50	failure	Max range exceeded: sum $150 \gg 100$.
TC-08	-1	10	10	error message	Negative value in x_1: invalid distance.
TC-09	10	-1	10	error message	Negative value in x_2: invalid distance.
TC-10	10	10	-1	error message	Negative value in x_3: invalid distance.
TC-11	10.5	10	10	error message	Invalid data type: non-integer input provided.
TC-12	`ten`	10	10	error message	Invalid data type: non-numeric string input provided.

Table 1: Drone program – black-box test cases generated by Gemini 3 Pro.

Observations and Analysis

- **Correctness of test cases:** Overall, the LLM's expected outputs are consistent with the specification $x_1, x_2, x_3 \geq 0$ and $x_1 + x_2 + x_3 \leq 100$. The test cases TC-01–TC-04 correctly predict `success` when the inputs are non-negative and the sum is at most 100. TC-05–TC-07 correctly predict `failure` when the total distance exceeds 100. TC-08–TC-12 exercise invalid inputs (negative or non-integer values) and expect an error message, which matches the requirement that invalid distances should be rejected.

- **Implicit testing strategy:** The LLM appears to implicitly apply a form of boundary value analysis. It selects values well inside the valid region (TC-01), then explicitly includes lower and upper boundaries (TC-02, TC-03, TC-04) and just-above boundary cases (TC-05, TC-06). It also explores some clearly invalid sums far beyond the limit (TC-07) and includes several tests with negative and non-numeric inputs (TC-08–TC-12). However, it does not explicitly partition the input domain into equivalence classes in the systematic way we did in Lab 2.

- **Number and variety of test cases:** Gemini produced 12 test cases, which is a reasonable size for this simple input domain. The set covers typical valid inputs, boundary values, and a mix of invalid cases. That said, the LLM did not try to minimize the suite or justify why these particular cases are sufficient (for example, by explicitly arguing coverage of each equivalence class).

- **Comparison to our manual EPC and weak $n \times 1$ suites:** Many of the LLM's test cases overlap with the types of tests we designed manually in Lab 2 (e.g., sums of 0, exactly 100, and slightly above 100). However, our EPC and weak $n \times 1$ strategies were more explicit about the underlying subdomains and guaranteed at least one representative from each equivalence class and boundary. In contrast, the LLM's suite is more ad hoc: it hits several important cases but does not come with a proof that all relevant partitions and edges have been covered.

Task 2 – Remote-Controlled Car Program (LLM EPC and Weak $n \times 1$)

Prompt Used

```
Assume you are an expert in software testing.

Attached is the remote-controlled car program from my software testing lab
(Lab 2 Task 2). The program takes several input parameters that control
the car and outputs whether the configuration is valid or not (and/or
produces specific behaviours according to the specification).

Your task is to design black-box test cases using two strategies:

1. Equivalence Partitioning and Boundary Value Analysis (EPC)
   - Identify the input variables and their domains based on the given specification.
   - Identify equivalence classes (valid and invalid) and their boundary values.
   - Construct a set of test cases that cover all equivalence classes and important
     boundaries.
   - Present the EPC test cases in a table with the following columns:
     - Test ID
     - Inputs
     - Expected Output / Behaviour
     - Rationale

2. Weak n×1 strategy
   - Based on the same input variables, apply the weak n×1 strategy: for each
     variable, vary that variable across its boundary values while keeping
     the other variables at typical (nominal) values.
   - Construct a set of weak n×1 test cases.
```

```
  - Present the weak n×1 test cases in another table with the following columns:
    - Test ID
    - Inputs
    - Expected Output / Behaviour
    - Rationale

Please clearly separate the two sets of test cases under headings:
- "EPC Test Cases"
- "Weak n×1 Test Cases".
```

LLM-Generated EPC Test Cases

Test ID	x	y	Expected	Rationale (Class / Boundary)
TC-EPC-01	0	0	Valid / Success	Valid class: nominal point near centre of domain.
TC-EPC-02	0.707	0.707	Valid / Success	Boundary: top-right corner of valid square (max x, max y).
TC-EPC-03	-0.707	-0.707	Valid / Success	Boundary: bottom-left corner of valid square (min x, min y).
TC-EPC-04	0.708	0	Invalid / Failure	Invalid class (high x): just above valid x upper limit.
TC-EPC-05	-0.708	0	Invalid / Failure	Invalid class (low x): just below valid x lower limit.
TC-EPC-06	0	0.708	Invalid / Failure	Invalid class (high y): just above valid y upper limit.
TC-EPC-07	0	-0.708	Invalid / Failure	Invalid class (low y): just below valid y lower limit.
TC-EPC-08	10.0	10.0	Invalid / Failure	Grossly out-of-bounds input: far outside the domain.

Table 2: RC car program – EPC test cases generated by Gemini 3 Pro.

LLM-Generated Weak $n \times 1$ Test Cases

Test ID	x	y	Expected	Rationale (Variable / Boundary Exercised)
TC-WNX-01	0	0	Valid / Success	Base case: both variables at nominal values.
TC-WNX-02	-0.707	0	Valid / Success	Var x min: x at lower boundary, y nominal.
TC-WNX-03	-0.708	0	Invalid / Failure	Var x min-1: x just below lower boundary, y nominal.
TC-WNX-04	0.707	0	Valid / Success	Var x max: x at upper boundary, y nominal.
TC-WNX-05	0.708	0	Invalid / Failure	Var x max+1: x just above upper boundary, y nominal.
TC-WNX-06	0	-0.707	Valid / Success	Var y min: y at lower boundary, x nominal.
TC-WNX-07	0	-0.708	Invalid / Failure	Var y min-1: y just below lower boundary, x nominal.
TC-WNX-08	0	0.707	Valid / Success	Var y max: y at upper boundary, x nominal.
TC-WNX-09	0	0.708	Invalid / Failure	Var y max+1: y just above upper boundary, x nominal.

Table 3: RC car program – weak $n \times 1$ test cases generated by Gemini 3 Pro.

Observations and Analysis

- **Correctness of test cases:** The expected outputs are consistent with the usual specification for this program, where x and y must both lie within a symmetric range (approximately $[-0.707, 0.707]$) for the configuration to be valid. Points on the boundary (e.g., $(0.707, 0.707)$ and $(-0.707, -0.707)$) are treated as valid, while any value just outside that range (e.g., 0.708 or -0.708) is treated as invalid. TC-EPC-08 additionally checks a clearly out-of-bounds case $(10.0, 10.0)$, which is sensible.

- **EPC strategy:** For EPC, Gemini covers: (i) a nominal valid point at the centre (TC-EPC-01), (ii) two corner boundary points (TC-EPC-02, TC-EPC-03), (iii) four "just outside" invalid cases along the axes (TC-EPC-04–07), and (iv) a grossly invalid outlier (TC-EPC-08). This does exercise the main valid and invalid equivalence classes and hits important boundaries, although the suite is fairly small and does not explore mixed cases such as one coordinate near the upper bound while the other is near the lower bound.

- **Weak $n \times 1$ strategy:** The weak $n \times 1$ tests follow the intended pattern quite closely: starting from a nominal base case (TC-WNX-01), the LLM varies x across its min, min-1, max, and max+1 values while keeping y nominal (TC-WNX-02–05), then varies y similarly while keeping x nominal (TC-WNX-06–09). This matches the definition of weak $n \times 1$: one variable moves across its boundary values at a time, while the other variables stay fixed at typical values.

- **Number of test cases:** Gemini generated 8 EPC tests and 9 weak $n \times 1$ tests. This is comparable in size to the hand-designed suites from Lab 2 and is sufficient to illustrate both strategies, although a human tester might add or remove a few cases depending on how aggressively they want to minimize or expand the suite.

- **Comparison to my manual Lab 2 suites:** The LLM's tests are very similar in spirit to the tests we designed manually for this program. Our Lab 2 EPC suite explicitly derived equivalence classes (inside the square, on the boundary, outside the square) and then chose representatives; Gemini implicitly reproduces that structure but without formally naming the classes. For weak $n \times 1$, the LLM correctly applies the idea of varying one coordinate at a time across its boundary and off-by-one values, just as we were instructed to do. The main limitation is that the LLM does not justify why these particular boundaries and deltas are chosen, nor does it give an explicit argument that all equivalence classes are covered.

Task 3 – MyBisect Program (LLM White-Box Tests)

Prompt Used

```
Assume you are an expert in software testing and white-box analysis.

Attached is the MyBisect program from my software testing lab. Your task is
to perform white-box test design for this program. Please do the following:

1. Code understanding and structure
   - Briefly summarize what the MyBisect function does and what its inputs
     and outputs are.
   - Identify the main conditional branches and any important error/exception
     cases in the code.

2. White-box test design (branch / path coverage)
   - Based on the code, design a set of test cases aimed at achieving good
     white-box coverage (at least branch coverage, and path coverage where
     appropriate).
   - Cover \normal" successful executions as well as all possible error or
     early-return behaviours present in the code.
   - For each test case, be explicit about:
     - The input values to MyBisect
     - The expected output (or exception / error condition)
     - Which branches / paths in the code are exercised by that test
     - A short rationale

3. Present the test suite in a table
   - Return the tests in a table with at least the following columns:
     - Test ID
     - Inputs
     - Expected Output / Behaviour
     - Branches / Paths Covered
```

```
  - Rationale

Please make sure the test suite reflects white-box reasoning about the
code structure, not just black-box input/output guessing.
```

LLM-Generated White-Box Test Cases for `MyBisect`

Table 4: `MyBisect` program – white-box test cases generated by Gemini 3 Pro.

Test ID	Inputs / Setup	Expected Output / Behaviour	Branches / Paths Covered	Rationale
WB-01	`MyBisect(0.1, 10, P1)`	Object created successfully.	`__init__`: 3-argument constructor path.	Exercises explicit initialization path.
WB-02	`MyBisect(0.05, P1)`	Object created (maxIterations = 50).	2-arg constructor path: first arg is float.	Covers tolerance-based constructor.
WB-03	`MyBisect(100, P1)`	Object created (tolerance = default).	2-arg constructor path: first arg is int.	Covers maxIterations-based constructor.
WB-04	`MyBisect(P1)`	Object created using defaults.	1-argument constructor path.	Covers default initialization.
WB-05	`b = MyBisect(P1); b.tolerance = -1`	Tolerance unchanged.	Tolerance setter: branch where `tol > 0` is false.	Tests invalid tolerance handling.
WB-06	`b = MyBisect(P1); b.run(2, 4)`	Raises `ValueError`.	`run`: early exit because `f1 * f2 > 0`.	Interval does not bracket a root.
WB-07	`b = MyBisect(P2); b.run(1, 2)`	Returns approx. $\sqrt{2}$.	`run`: left update branch taken.	Normal successful convergence.
WB-08	`b = MyBisect(P2); b.run(1.4, 1.5)`	Returns approx. $\sqrt{2}$.	`run`: right update branch taken.	Covers complementary branch.
WB-09	`b = MyBisect(P1); b.run(-2, 2)`	Returns 0.0.	Loop exit via `abs(fmid) <= tolerance`.	Exact root at midpoint.
WB-10	`b = MyBisect(1, P2); b.run(0, 3)`	Raises `ValueError`.	Loop ends due to maxIterations exceeded.	Covers non-convergence error path.

Observations and Analysis

- **Correctness of expected behaviours:** The expected results for each test are consistent with the intended behaviour of the bisection method and the `MyBisect` API. For `P1` (where $f(x) = x$), the root at $x = 0$ is correctly reported in WB-09. For `P2` (where $f(x) = x^2 - 2$), the root near $\sqrt{2} \approx 1.414$ is returned in WB-07 and WB-08. The tests that expect `ValueError` (WB-06 and WB-10) line up with the code paths where either the interval does not bracket a root or the algorithm fails to converge within `maxIterations`.

- **Coverage of branches and paths:** The test suite explicitly targets different branches in both the constructor and the `run` method. WB-01–WB-04 cover all of the constructor paths corresponding to different numbers and types of arguments. WB-05 exercises the branch in the tolerance setter where a non-positive tolerance is rejected. In `run`, WB-06 covers the early-exit error when $f(x_1)$ and $f(x_2)$ have the same sign; WB-07 and WB-08 exercise both sides of the interval update condition (*left-half* vs. *right-half* updates); WB-09 covers termination via the function-value tolerance `abs(fmid) <= tolerance`; and WB-10 covers the path where the loop terminates because the iteration limit is exceeded and triggers the post-loop error.

- **Use of internal code structure:** Unlike the black-box prompts in Tasks 1 and 2, the LLM explicitly reasons about internal branches (constructor overload emulation, precondition check, loop update condition, and termination cases). The rationale for each test refers to specific branches (for example, the precondition check and the two different update branches in the loop), indicating genuine white-box reasoning rather than just black-box input-output guessing.

- **Comparison to our manual white-box suite:** The LLM-generated suite is similar to the tests we designed manually in the earlier lab: both aim to cover the error path when the initial interval does not bracket a root, both update directions in the bisection loop, and multiple termination conditions (interval size, function value, and maximum iterations). One difference is that our manual suite explicitly tied each test to a path in the control-flow graph, while the LLM provides more informal descriptions of the branches. The LLM also focuses on a small number of representative tests; a human tester might add or remove cases depending on how much path coverage beyond simple branch coverage is desired (for example, testing zero-iteration cases or different ways to exceed the iteration limit).

Discussion and Conclusion

This lab explored the use of a modern large language model (Gemini 3 Pro) as an automated test designer across three progressively more structured tasks. The goal was to compare LLM-generated test suites with the systematic black-box and white-box techniques taught in class, and to evaluate the strengths and weaknesses of using an LLM as part of a software testing workflow.

Black-Box Test Generation (Tasks 1 and 2)

In Task 1 (drone distance checker), we intentionally used a simple, barebone-style prompt. Gemini responded with a small but coherent suite of valid, boundary, and invalid test cases. The tests were generally correct, but the LLM did not explicitly articulate equivalence classes or boundary reasoning. Instead, it inferred the shape of the domain and selected representative points heuristically. This demonstrates that an LLM can generate reasonable input-output tests even with minimal guidance, but the coverage is not guaranteed and the underlying rationale is implicit rather than systematic.

In Task 2 (RC car controller), we shifted to a structured prompt requiring both EPC and weak $n \times 1$ strategies. The LLM followed instructions more faithfully: it identified boundaries such as ± 0.707, produced representatives from valid and invalid partitions, and created a clean weak $n \times 1$ suite that varied one variable at a time while holding the other constant. Although the LLM did not derive equivalence classes as rigorously as a human tester would, the resulting test sets closely resembled the manually constructed suites from Lab 2. This illustrates the value of prompt engineering: when the testing strategy is explicitly requested and structured, the LLM is capable of reproducing it.

White-Box Test Generation (Task 3)

Task 3 involved white-box reasoning over the `MyBisect` algorithm. Here the LLM performed notably better than in the black-box tasks. It correctly summarized the control flow, identified key branches in the constructor, setter, precondition checks, and the main bisection loop, and constructed a test suite that achieved full branch coverage. It recognized early failure conditions, both update branches of the bisection step, the different loop termination conditions, and the path where the iteration limit is exceeded. These tests closely matched the white-box suite we constructed manually in the earlier lab.

The LLM's success in this task suggests that it can effectively reason about source code structure when explicitly instructed to do so. However, its branch classifications remained informal, and it did not attempt true path coverage nor did it refer to precise CFG paths as a human tester might.

Strengths, Limitations, and Observations

- **Strengths:** The LLM quickly produced test suites of reasonable quality, especially when the prompt included explicit structure. It demonstrated awareness of boundaries, robustness cases, and control-flow reasoning. Its tests were generally correct and aligned with the expected program behaviour.

- **Limitations:** The LLM does not guarantee completeness. It may miss edge cases or produce redundant tests. Its white-box reasoning, although competent, is not formal and does not produce explicit path sets or coverage proofs. Additionally, its behaviour is highly sensitive to how the prompt is phrased; unstructured prompts lead to inconsistent or underdeveloped test suites.

- **Prompt Dependence:** A key observation from this lab is that prompt structure directly affects the quality of the generated tests. Minimal prompts yield heuristic, black-box-style tests, while structured prompts lead to much more systematic and strategy-aligned suite generation. This mirrors the role of a human tester: clearer requirements produce better tests.

- **Human vs. LLM Comparison:** LLM-generated tests can approximate human-designed EPC, weak $n \times 1$, and branch coverage suites, but they do not replace the need for human oversight. Humans provide the formal reasoning about equivalence classes, domain modelling, and path coverage that an LLM cannot fully guarantee.

Conclusion

Overall, Gemini 3 Pro performed surprisingly well as a test generator, particularly when directed with a structured prompt. While it cannot fully replace manual systematic testing, it serves as a useful assistant capable of generating initial test suites, identifying boundary cases, and even highlighting relevant branches in non-trivial code. The effectiveness of the LLM depends strongly on prompt quality, and the results must still be validated by a human tester. Nevertheless, LLMs represent a promising tool for augmenting software testing workflows by accelerating the early stages of test design and reducing the manual effort required to enumerate representative test cases.