LLM-Assisted Automated Test Generation

Project Report

Arthur RICHELET Cédric DERACHE

1 Problem Description

Modern software systems increasingly rely on extensive test suites to guarantee correctness, reliability, and maintainability. However, the creation of these test suites remains largely manual, requiring significant human effort and domain understanding. The problem addressed in this project is the automation of test procedure generation using Large Language Models (LLMs), given a project's formal requirements and its source code.

Manually writing tests requires developers or verification engineers to parse and interpret potentially large and complex technical documentation. Requirements may span dozens or hundreds of pages, be written at various abstraction levels, and often contain ambiguities or cross-dependencies. Developers must integrate this information with the program's code structure to identify expected behaviors, corner cases, input constraints, and error-handling paths. This process is inherently tedious and error-prone, leading to common issues such as:

- missed or incomplete coverage of the functional specification,
- inconsistencies across test procedures written by different developers,
- significant time investment, especially when requirements evolve or codebases grow.

Modern development teams must also satisfy both requirement-based testing and coverage-driven testing. Ensuring that tests simultaneously align with the functional specification and exercise a sufficiently large portion of the source code can greatly increase the number of test cases that need to be written, amplifying the overall workload.

LLMs present a promising solution, as they are capable of processing and synthesizing large volumes of textual and structural information, including natural language requirements and source code. The central problem this project attempts to solve can be stated as follows:

> *How can we design a toolchain that automatically retrieves requirements and source code, and uses an LLM to generate coherent, complete, and specification-aligned test procedures while reducing human workload and minimizing missed coverage?*

To address this problem, the project aims to integrate:

1. a module capable of extracting requirements from a requirements database,
2. a module capable of fetching source files through a Git interface,
3. an LLM-driven component that interprets both sources of information and produces meaningful test cases,
4. a controlled interface allowing the LLM limited access to create or modify files on the tester's machine.

This work ultimately targets improved efficiency, consistency, and coverage in automated test generation workflows.

2 Experimental Comparison

To evaluate how effectively our pipeline generates test specifications, we compared four levels of code context provided to the LLM: `none`, `file_list`, `file_snippets`, and `full`. These levels represent how much knowledge about the codebase is exposed to the model. With `none`, the LLM only sees the textual requirement specification. With `file_list`, it receives only the names of the source files, giving it a minimal structural view of the project. With `file_snippets`, the LLM is given short excerpts of code (function signatures, class headers, or small method bodies), enabling it to align tests with concrete code elements. Finally, `full` attempts to provide the entire source code.

As a baseline, we used our manually written test specification ("*task_service_manual_tests.md*"), which defines the expected behavior of the `TaskService` component using traditional human-authored test descriptions. This baseline allows us to compare the completeness and precision of automatically generated tests against what a human engineer would typically produce.

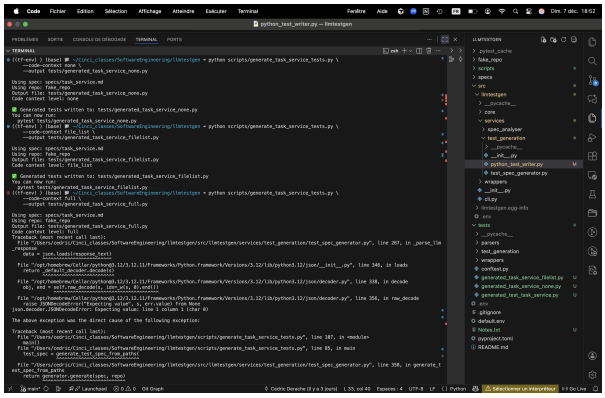
Our goal was to measure (1) the completeness of generated test cases, (2) their alignment with the functional requirements, and (3) the degree of structural coupling with the actual source code.

2.1 Baseline: Manual Test Specification

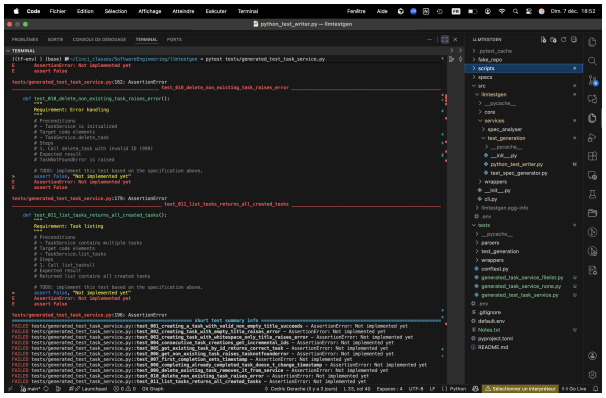
The manual reference tests describe eleven functional behaviors (task creation, validation, retrieval, deletion, completion, and timestamp semantics). These tests serve as the target quality level: coherent requirement coverage, explicit preconditions, and logically ordered steps.

2.2 Experimental Setup

To evaluate our tool, we generated test suites for the same Python service (`TaskService`) under four different code–context settings: *None*, *File List*, *File Snippets*, and *Full Code*. For each configuration, the LLM received the same requirement specification but a different level of visibility into the project structure. The generated tests were then compared to a manually written baseline test suite.



(a) LLm call



(b) PyTest after the call

2.3 Results Overview

Across all configurations, the LLM produced coherent and structured test cases aligned with the functional requirements. The *None* mode generated high–level tests that remained close to the natural language specification, while *File List* introduced stronger alignment with the codebase (e.g., referring to specific modules or methods). *File Snippets* consistently produced the most accurate and complete tests, with correct identification of methods, expected behaviours, and corner cases. The *Full* mode did not yield reliable output in our experiment due to excessive prompt size, confirming that full–code prompting is impractical for even moderately sized repositories.

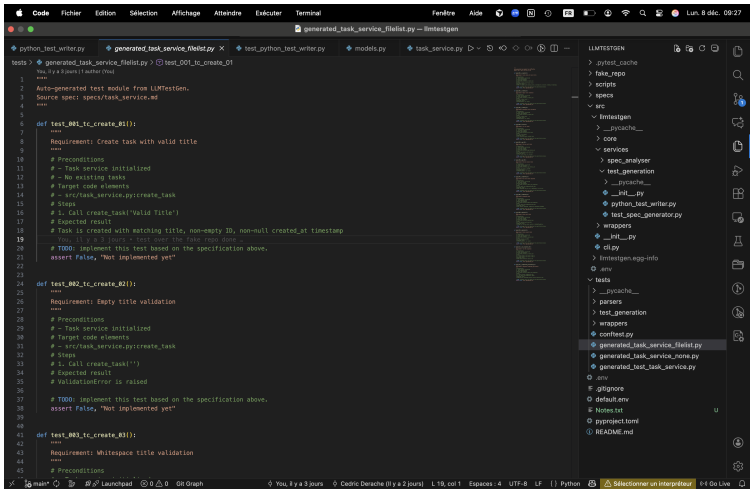


Figure 2: Example of a generated test file

2.4 Comparison With Manual Baseline

The manually written test suite served as a reference for completeness, correctness, and structure. Compared to this baseline, LLM–generated tests in *Snippets* mode matched nearly all core behaviours: validation rules, error conditions, state transitions, ID assignment, and timestamp handling. However, the generated tests remained non-executable placeholders, lacking the concrete Python assertions and setup logic present in the manual baseline. Despite this, they significantly accelerate the design phase by providing a correct and consistent test skeleton.

2.5 Summary of Results

Table 1 summarizes the observed behavior for each configuration.

Context Level	Requirement Coverage	Code Alignment	Validity
None	High	Low	Valid
File List	High	Medium	Valid
File Snippets	Very High	High	Valid
Full	N/A	N/A	Invalid (JSON error)

Table 1: Comparison of LLM-generated test suites under different code context modes.

2.6 Discussion

The results indicate that increasing the contextual richness improves the quality of generated tests, but only up to a certain point. The `file_snippets` mode clearly provides the best alignment with both the manual baseline and the project code structure. The `full` mode, though conceptually ideal, suffers from practical limits related to prompt size and structured-output stability.

Overall, the experiment demonstrates that:

- LLMs can autonomously generate coherent and requirement-aligned test suites;
- providing moderate amounts of structured code context yields significant improvements;
- our pipeline successfully automates a task that is normally time-consuming and error-prone.

These findings support the viability of LLM-assisted test generation and justify further investment in prompt engineering, code summarization techniques, and evaluation metrics.