

# Understanding Specification-Driven Code Generation with LLMs: An Empirical Study Design\*

Giovanni Rosa<sup>†</sup>, David Moreno-Lumbreras<sup>†</sup>, Gregorio Robles<sup>†</sup>, Jesús M. González-Barahona<sup>†</sup>

<sup>†</sup>Escuela de Ingeniería de Fuenlabrada, Universidad Rey Juan Carlos, Fuenlabrada, Spain

**Abstract**—Large Language Models (LLMs) are increasingly integrated into software development workflows, yet their behavior in structured, specification-driven processes remains poorly understood. This paper presents an empirical study design using CURRANTE, a Visual Studio Code extension that enables a human-in-the-loop workflow for LLM-assisted code generation. The tool guides developers through three sequential stages—Specification, Tests, and Function—allowing them to define requirements, generate and refine test suites, and produce functions that satisfy those tests. Participants will solve medium-difficulty problems from the *LiveCodeBench* dataset, while the tool records fine-grained interaction logs, effectiveness metrics (e.g., pass rate, all-pass completion), efficiency indicators (e.g., time-to-pass), and iteration behaviors. The study aims to analyze how human intervention in specification and test refinement influences the quality and dynamics of LLM-generated code. The results will provide empirical insights into the design of next-generation development environments that align human reasoning with model-driven code generation.

**Index Terms**—Specification-Driven Development, Large Language Models, Code Generation, Test-Driven Development, Empirical Software Engineering

## I. INTRODUCTION

Test-Driven Development (TDD) has long been a fundamental paradigm in software development, but the introduction of Large Language Models (LLMs) is paving the way for new approaches to coding assistance, providing a potential transformation in how developers write and verify code.

The current most-used assistance for code generation is provided via code completion plugins, such as GitHub Copilot [1], or fully-featured AI-powered IDEs, such as Cursor.<sup>1</sup>. These tools offer powerful automation of the coding process leveraging natural language instructions, shifting from the traditional code-centric development approach to a more abstract specification-driven process, namely *Spec-Driven Development* (SDD) [2]. Still, they lack a structured approach to requirements specification and testing that are crucial factors for ensuring code quality as highlighted in several studies on the integration of LLMs with TDD workflows [3]–[6].

This prior studies validated these TDD workflows leaving the **human factor** underexplored. In fact, this presents a **gap** in understanding the user’s role in guiding the LLM through structured requirements specification and test case definition,

\*This paper is a Stage 1 Registered Report. The study protocol and analysis plan were peer reviewed and accepted at SANER 2026 with a Continuity Acceptance (CA) score for Stage 2.

<sup>1</sup><https://cursor.com/>

and, more importantly, how effectively they can express their intent through the overall TDD-based workflow.

In this paper, we propose a an empirical experiment to investigate how developers can effectively interact with such a workflow, focusing more on how they can express their intent through the specification and test case definition phases. To this end, we rely on CURRANTE, a plugin that implements a typical TDD-code generation workflow based on the existing literature [3], [4], [6], within a popular IDE, Visual Studio Code (VSCode). CURRANTE mainly consists of a three-phase TDD workflow: the *problem description* phase, (1) where the user inputs the specification, (2) the *test cases* phase, where the user guides the LLM in generating and refining a test suite that formally describes the requirements. (3) the final phase, *i.e.*, *code generation*, is delegated entirely to the LLM, which generates the code and verifies its execution results against the test suite. We plan a controlled experiment with human participants, asked to solve a set of programming problems from the LiveCodeBench dataset. Working exclusively within CURRANTE and VS Code, participants will be asked to produce a valid solution for the proposed problem. We will collect detailed data on effectiveness (e.g., *PassAll*, *PassRate*, *TestCoverage*, *TestDiversity*), efficiency (e.g., *TimeToPass*), and interactions (e.g., *TestEdits*, *SuiteRegenerations*) in order to analyze the real impact of user involvement in the test curation phase on the overall code generation success.

The expected outcome of our study is twofold: first, to provide empirical evidence on the effectiveness of a TDD-based workflow for LLM-assisted code generation focusing on the user perspective and second, to inform the design of future IDEs, clarifying the trade-off between investing in requirements engineering and code refinement, and how the user can best contribute to the process.

The rest of the paper is structured as follows. Section II provides background information and related works. Sections IV and VI presents CURRANTE and the proposed experiment, while Section VII discusses threats to validity and Section VIII summarizes the conclusions.

## II. BACKGROUND AND RELATED WORK

A significant contribution in this area was made by Chen *et al.* [7], who introduced the *Codex* model and an evaluation protocol defining the *pass@k* metric, the current standard, and the HumanEval benchmark, consisting of coding problems with associated unit tests.

Other research has examined how LLMs can be combined with established software engineering methodologies to improve code quality and reliability, such as TDD. Mathews *et al.* [3] proposed the TGen approach, which implements a TDD-guided code generation flow using LLMs. Starting from a set of initial test cases, TGen generates code and iteratively updates it until all tests pass. Their experiments showed that integrating TDD workflows enhances both the accuracy and dependability of code produced by LLMs. Along the same lines, Fakhouri *et al.* [4] introduced TICODER, incorporating human feedback into the TDD-based code generation loop. Additionally, Piya *et al.* [5], [8] explored how combining TDD practices with assistants like ChatGPT improves problem-solving on *LeetCode*-style tasks.

In parallel, recent work has investigated the integration of LLMs within development environments and the dynamics of human–AI interaction. Amershi *et al.* [9] established fundamental principles for effective human–AI collaboration, emphasizing timely feedback, transparency, and user control—principles embodied in our structured *Specification–Tests–Function* workflow. Sergeyuk *et al.* [10] and Nghiem *et al.* [11] highlight the growing importance of IDE-embedded assistants and the need for transparent, user-centric design in coding tools. Complementarily, Crupi *et al.* [12] and Evtikhiev *et al.* [13] discuss the limitations of existing automatic evaluation metrics, such as BLEU or *pass@k*, arguing for interactive and context-aware evaluation frameworks. Finally, Gao *et al.* [14] point to broader challenges in the use of LLMs for software engineering, including trust, reproducibility, and transparency—issues that our work directly addresses through controlled, in-IDE experimentation.

We build upon these prior works by exploring how a structured TDD workflow, starting from user-defined specifications, can be effectively leveraged for generating correct code with LLMs. Our study focuses on the interaction between the user and the TDD approach itself, an aspect that has been limitedly investigated, with the aim of examining the role of user involvement in test case definition and refinement within TDD.

### III. THE CURRENTE PLUGIN

We designed and developed a Visual Studio Code (VS Code) plugin named CURRENTE, which serves as the experimental platform for this study. CURRENTE implements a TDD workflow that leverages LLMs to generate code based on user-provided specifications formally defined through test cases. The workflow is structured into three main phases, represented in the graphical user interface (GUI) as follows:

- 1) **Problem description (*area 1*):** A text area where the user can input a structured natural language specification of the problem to be solved. The specification follows a TOML format, capturing the user intent and guiding the LLM to generate the test suite. It also contains the function signature and any necessary constraints or requirements.

- 2) **Test cases (*area 2*):** The central section, where the user can review, refine, and improve the test cases describing the problem requirements. An initial test suite is automatically generated from the TOML specification, which serves as input specification for the function generation phase.
- 3) **Code generation (*area 3*):** The final section, which displays the generated code and the corresponding test suite execution results. The user can regenerate the code after refining the test cases.

The current prototype of CURRENTE operates as follows. The user starts by defining the problem requirements in a structured way, via a TOML file. The format is designed to be human-readable and easy to edit<sup>2</sup>. Upon launching CURRENTE, the TOML specification is automatically loaded into *area 1*, where it can be further edited. Next, the user initiates the test suite generation process, which employs an LLM to produce an initial set of test cases, displayed in *area 2*. While the TOML specification provides the initial context for test generation, the test suite itself serves as input specification for the overall Spec-Driven Development workflow. An example of the user interface is shown in Figure 1. The user can iteratively refine the generated test cases by (i) requesting natural language explanations for individual tests, (ii) deleting irrelevant ones, or (iii) providing additional guidance to the LLM to regenerate a single test or the entire suite. Once satisfied with the resulting test suite, the user proceeds to the code generation phase (*area 3*), where CURRENTE uses the LLM to produce the corresponding code implementation based on the refined tests.

## IV. EXPERIMENT GOAL AND RESEARCH QUESTIONS

The *goal* of this experiment is to evaluate whether an LLM-based TDD-inspired workflow is effective in generating correct code by providing and refining a formal input specification (i.e., test cases). We implemented this workflow in a VS Code plugin, CURRENTE, which we use to conduct the experiment. The *perspective* is that of software developers and researchers interested in LLM-assisted code generation workflows.

### A. Research Questions

The experiment is guided by the following research questions:

**RQ<sub>1</sub>:** *To what extent is CURRENTE able to generate code based on the specification provided by the user?* We aim to measure the effectiveness of a TDD-based workflow, implemented through CURRENTE, in generating correct code solutions for a given problem without requiring the user to write code directly, but instead by providing and refining a formal specification through test cases. This RQ focuses more on the validation of the tool’s code generation capabilities.

**RQ<sub>2</sub>:** *To what extent does CURRENTE allow the user to express their intent through a test suite specification?*

<sup>2</sup><https://toml.io/en/>

```

Specification Tests Function
Tests
When the specification is ready, use this tab to produce a suitable testsuite for the function. Ask for a new explanation, or for a new version of individual tests if needed. When the testsuite is good enough, ask for the function.

New version of the whole testsuite Ask for the function

test_multiply_samples_basic_case
Explain New version Delete

1 | import unittest
2 | from _generated_function import multiply_samples
3 |
4 |
5 | class TestMultiplySamples(unittest.TestCase):
6 |
7 |     def test_multiply_samples_basic_case(self):
8 |         """Test basic multiplication with positive factor"""
9 |         samples = [1, 2, 3, 4, 5]
10 |        factor = 2.5
11 |        expected = [3, 5, 8, 10, 13]
12 |        self.assertEqual(multiply_samples(samples, factor), expected)

Explanation
## What this test verifies
**Test name**: test_multiply_samples_basic_case
- Tests the basic functionality of multiplying a list of numbers by a factor

**Input/Setup**:
- Samples = [1, 2, 3, 4, 5] (a list of integers)
- Factor = 2.5 (a positive float)

**Expected output**:
- [3, 5, 8, 10, 13] (each original number multiplied by 2.5 and rounded to nearest integer)

**Why this matters**:
This test checks that the function correctly:
1. Multiplies each sample by the factor (1x2.5=2.5 → rounds to 3. 2x2.5=5 → stays 5, etc.)

```

Fig. 1: Screenshot of the prototype of CURRANTE plugin integrated into Visual Studio Code (VS Code).

This RQ focuses more on the human-in-the-loop aspect, measuring how well users are able to express their intent producing an effective test suite that captures the functional problem requirements.

#### B. Context Selection

The context of the study is composed of subjects and objects. The subjects are software developers, using CURRANTE to solve a set of coding problems. The objects are coding problems selected from the *LiveCodeBench* dataset [15].

**1) Participants and Settings:** We aim to recruit participants with a diverse range of experience levels (*i.e.*, both junior and senior developers), backgrounds (*i.e.*, software developers, computer science students, and professionals from different industries), and demographics. All sessions are individual and conducted on a *single-monitor* workstation (desktop or laptop) with keyboard and mouse, using VS Code (fixed version) and the CURRANTE extension; the LLM endpoint, prompts, and parameters are fixed across participants. Before the tasks, participants receive a brief tutorial and one warm-up example; during the study, they work only within the IDE (no external tools or web search). The extension logs time-stamped actions (produce/explain/regenerate tests; ask/regenerate function; re-run tests) and unit-test outcomes (per-test pass/fail). Participation is voluntary under informed consent; data are pseudonymized and contain only the artifacts and interaction logs necessary for the study.

**2) Dataset Selection:** We plan to select a sample of coding problems from the *LiveCodeBench* dataset [15], which contains a large set of high-quality coding problems, defined for different coding tasks, extracted from competitive programming platforms. The benchmark is continuously updated with new problems, providing detailed information for each

problem, such as test cases, problem description, difficulty level, and solution. More details are reported in the *HuggingFace* dataset card<sup>3</sup>. The version of the dataset will be *release\_v5*, containing a total of 880 coding problems until January 2025. We will select a sample of three short warmup problems (easy difficulty), and three evaluation-candidate problems (medium difficulty). The experiment will be conducted only on the evaluation-candidate problems.

The problem candidates will be selected based on the following criteria:

- **Difficulty level:** We will select problems labeled as *easy* and *medium* difficulty, to ensure a reasonable challenge for participants, while being solvable within the time constraints of the experiment.
- **Diversity of problem types:** We will select a mix of problem types (*e.g.*, algorithms, data structures) to evaluate different coding scenarios, as diverse as possible, with a trade-off ensuring the feasibility of the experiment. A possible approach is to (i) randomly select a problem, (ii) evaluate its type and feasibility, and (iii) keep or discard it until the desired diversity is achieved. This will apply to the evaluation-candidate problems. The warm-up problems will be selected to be simple and quick to solve, helping to familiarize participants with the tool and workflow.
- **Problem length:** We will select problems that can be reasonably solved within the time constraints of the experiment, ensuring that the problem description is easily understandable and the required constraints and edge cases are not overly complex.

<sup>3</sup>[https://huggingface.co/datasets/livecodebench/code\\_generation\\_lite](https://huggingface.co/datasets/livecodebench/code_generation_lite)

- **Avoiding data contamination:** We will avoid problems that are likely to have been included in the training data of the LLM used in CURRANTE, based on the reported dates and known datasets.

### C. Variables

We consider three groups of variables aligned with our CURRANTE workflow on *LiveCodeBench* tasks. *Independent variables* capture task context (task identifier). *Dependent variables* measure effectiveness and process—final correctness (all tests passed and pass rate), efficiency (time to pass), test suite strength (tests coverage and diversity) and iteration dynamics (function regenerations, per-test edits, full suite regenerations, and advice triggers). *Confounding variables* control for prior skill and habits (years of programming experience, Python familiarity, prior TDD experience, and prior use of LLMs for coding). All metrics are computed from the plugin’s time-stamped logs; times are measured from the first produced test suite, and counts aggregate user actions recorded by CURRANTE. A complete list with scales is summarized in Table I.

## V. EXPERIMENTAL PROCEDURE

We follow the *ACM SIGSOFT Empirical Standards* [16] for quantitative studies involving human participants. Before the main task, each participant receives a short interactive tutorial introducing the CURRANTE interface and workflow, including the three tabs (*Specification*, *Tests*, and *Function*), and a step-by-step demonstration of how to produce, refine, and validate test suites. After this tutorial, participants complete one randomly selected *warm-up task* to familiarize themselves with the process and the experimental environment. This training phase ensures that participants understand the workflow and controls for learning effects during the main evaluation.

The experiment follows a *between-subjects* design: each participant solves one *LiveCodeBench* problem using the same CURRANTE workflow (*Specification* → *Tests* → *Function*). Each participant is randomly assigned one *medium*-difficulty problem from a set of three candidates, ensuring balanced assignment across participants. We use the specific *TaskId* as a blocking factor to account for variation among problems.

Measurements are collected only during the main evaluation task (excluding the warm-up), capturing user actions, outcomes, and performance metrics. This procedure supports a fair assessment of effectiveness and efficiency under a consistent spec-driven process, while satisfying core empirical standards regarding design clarity, control of nuisance variables, and reproducibility.

The underlying LLM model will be selected based on the most recent and capable open-weight model available at the time of experiment execution. A possible option is *Qwen3-Coder* [17], which is specialized for coding tasks. The prompts will be kept simple to avoid additional influencing factors (*i.e.*, no advanced prompting techniques), following the TDD workflow outlined above.

### A. Data Collection

We instrument CURRANTE to capture time-stamped interaction logs and outcome data aligned with the variables in Table I. *Demographics* (pre-study) data collection include years of programming experience, Python familiarity, prior TDD experience, and prior LLM usage for coding; *post-task feedback* captures perceived usability and workload (short Likert items and free-text). *Process logs* record every action with timestamps and payload hashes: produce test suite, per-test actions (explain/regenerate/delete), full-suite regenerations, ask/regenerate function, re-run tests, and function viewing events (e.g., when the participant inspects or edits the generated code). *Execution results* store per-test pass/fail, failure messages, aggregate pass rate, and whether all tests passed; time-to-pass is computed from the first produced test suite to the first all-pass submission (or budget exhaustion). *LLM usage metrics* include token counts for each API call and total tokens consumed per participant, enabling efficiency and cost analyses. *Artifacts* (specification in TOML format, test versions, function versions) are stored as text with content hashes to ensure integrity and reproducibility without retaining identifiable project code. All logs are pseudonymized, collected on the study workstation, and exported to a CSV/JSON bundle alongside artifact snapshots for analysis. Even if participants do not complete the task or do not produce usable tests, we still analyze their usage logs to have insights on potential issues in the CURRANTE workflow.

### B. Coding Tasks

Participants will complete the coding task (*i.e.*, problem) entirely within VS Code using CURRANTE. The in-experiment workflow is:

- Open specification.** Ensure the TOML specification of the assigned problem is open in the editor (Specification tab).
- Produce initial tests.** Invoke *Produce test suite* to generate a runnable unit-test file.
- Refine tests (human-in-the-loop).** Inspect and curate the suite using per-test actions: *Explain*, *Regenerate*, *Delete*; optionally regenerate the whole suite. Aim for a clear, sufficient specification-by-tests.
- Generate function & execute.** Use *Ask for the function*; CURRANTE first generates the code function, then runs the tests and reports per-test results and aggregate pass rate, plus auto-generated advice from failing outputs.
- Iterate to completion.** If failures remain, either (i) refine the tests or (ii) *Re-generate function* guided by the advice; re-run until completion criteria are met.

*Completion criteria.* The task finishes when the function passes all tests or when the fixed time budget elapses.

*Measurement.* Time-to-pass is measured from the first test-suite production; all actions (per-test edits, suite regenerations, function regenerations) and outcomes (per-test pass/fail) are logged.

*Note.* The concrete wording of the assigned medium-difficulty problem and any minor UI micro-steps will be finalized at

Variable Type	Name	Description	Scale
Independent	TaskId	Identifier of the specific problem instance used in the session (blocking factor).	Categorical (string)
Dependent	PassAll	Whether the final generated function passes <i>all</i> tests in the suite.	Binary: 0 (no), 1 (yes)
Dependent	PassRate	Fraction of passed tests over total tests for the final submission.	Real number [0, 1]
Dependent	TestCoverage	How much of the source code is executed when the test suite runs.	Real number [0, 1]
Dependent	TestDiversity	How different the test cases are from one another, using a similarity metric (e.g., Jaccard similarity).	Real number [0, 1]
Dependent	TimeToPass	Time from producing the first test suite to the first submission that passes all tests (or budget expiration).	Real number (seconds)
Dependent	IterationsToPass	Number of function regenerations until first all-pass (or budget expiration).	Integer (count)
Dependent	TestEdits	Number of per-test actions (explain / regenerate / delete) performed by the participant.	Integer (count)
Dependent	SuiteRegenerations	Number of full test-suite regenerations triggered from the Specification/Tests tab.	Integer (count)
Dependent	AdviceTriggers	Number of times advice was requested/generated from failing outputs.	Integer (count)
Confounding	ProgrammingExperienceYears	Participant’s general programming experience.	Integer (years)
Confounding	PythonFamiliarity	Self-reported familiarity with Python.	Categorical: “none”, “low”, “medium”, “high”
Confounding	PriorTDDExperience	Self-reported experience with TDD-like workflows.	Categorical: “none”, “low”, “medium”, “high”
Confounding	PriorLLMCodeGenUse	Prior use of LLMs for coding tasks.	Categorical: “never”, “occasionally”, “frequently”

TABLE I: Variables used in the experiment with CURRANTE on LiveCodeBench tasks.

experiment execution time, preserving the workflow described above.

### C. Metrics

We derive all metrics from CURRANTE logs and test executions, aligned with Table I:

- **Correctness.** *PassAll* (binary: all tests passed at task end) and *PassRate* (final #passed / #total tests).
- **Efficiency.** *TimeToPass* (seconds from first test-suite production to first all-pass submission, or to time-budget expiration), and *IterationsToPass* (number of function regenerations until first all-pass, capped at budget).
- **Process activity.** *TestEdits* (count of per-test actions: explain/regenerate/delete), *SuiteRegenerations* (count of full-suite regenerations), and *AdviceTriggers* (count of advice requests generated from failing outputs).
- **Per-test outcomes.** For each test, pass/fail label and last failure message (used only for advice generation and qualitative inspection).

*Computation and handling.* All timestamps are recorded using the system clock; rapid duplicate clicks are filtered out during logging. If the participant does not achieve an all-pass result, *PassAll* is set to 0 and *TimeToPass* is assigned the full time budget. Any outliers due to interruptions are flagged and excluded in sensitivity checks. We plan to report common descriptive statistics (median and interquartile range for times and counts; proportions for *PassAll*), and use Spearman’s  $\rho$  for exploratory correlation analysis.

*Note.* The exact operationalization of all metrics (e.g., timer start/stop points, aggregation/capping rules) will be finalized

at experiment execution time and documented in the study protocol.

## VI. EXECUTION PLAN

The study will proceed in three main phases to ensure methodological consistency and transparency of results: *Preparation*, *Execution*, and *Analysis*. Each phase is described below.

**Preparation Phase.** We will freeze the experimental environment by fixing the VS Code version, CURRANTE build, prompt templates, and LLM endpoint parameters (temperature, model, and seed) to maximize consistency across runs. The *medium*-difficulty subset of LiveCodeBench will be filtered to remove duplicates and problems potentially included in LLM pretraining corpora. We will select three *training* tasks (easy difficulty) and three *evaluation* tasks (medium difficulty), ensuring diversity in algorithmic type and data structure usage. A pilot session (2–3 participants) will validate the selected problems, timing, LLM configurations, and data collection pipeline. All experimental materials (consent form, pre/post questionnaires, task sheets) will be finalized after the pilot, documenting any changes and their rationale.

**Execution Phase.** Each participant will complete the study individually on a single-monitor workstation with VS Code and CURRANTE pre-installed. After giving informed consent, participants will complete a short pre-study questionnaire (demographics, programming experience, Python familiarity, prior TDD and LLM-assisted coding). They will then perform: (i) a brief *training task* randomly assigned from the three easy problems to become familiar with the interface; and (ii) one *evaluation task* randomly assigned from the three

medium-difficulty problems, using balanced assignment across participants.

During the evaluation, participants follow a fixed workflow within CURRANTE (Specification → Tests → Function) under a fixed time budget (approximately 30–45 minutes in total). All actions are logged automatically by the extension, including time-stamped interactions, per-test outcomes, token counts, and code inspection events (e.g., when the participant views or edits the generated function). External tools or web search are not permitted. After completing the task, participants will fill out a brief post-study questionnaire to report perceived usability and workload (Likert items and free-text comments). All logs and artifacts (TOML specification, test versions, function versions) are stored as text with content hashes to preserve integrity and anonymity.

**Analysis Phase.** We will start with descriptive summaries of all variables listed in Table I (e.g., proportions for categorical outcomes, medians/IQR for continuous variables, distributions of process actions). Further analyses will be chosen *as warranted by the collected data*. Depending on the data characteristics, we may employ non-parametric comparisons for times and counts, basic regression analyses for associations with participant factors, or survival-style summaries if time-to-pass shows right-censoring. Any effect-size estimation, multiple-comparison handling, or robustness checks will be applied pragmatically and reported transparently in the final analysis plan. All de-identified logs, scripts, and artifact snapshots will be archived for replication and secondary analysis.

This structured plan ensures a coherent progression from preparation to execution and analysis, minimizing bias while maintaining flexibility to adapt the statistical approach to the actual data collected.

## VII. THREATS TO VALIDITY

Threats to validity are potential biases or uncontrolled influences that can distort findings. We distinguish *internal validity* (factors inside the study that affect causal interpretation, e.g., participant heterogeneity, instrumentation, learning effects) from *external validity* (limits to generalizing results beyond our sample, tasks, tools, and setting).

### A. Internal Validity

**Subjects.** Participant heterogeneity (programming experience, Python familiarity, prior TDD/LLM use) may influence outcomes. We mitigate this by collecting demographics pre-study and analyzing with these factors as covariates; instructions and time budget are kept constant across sessions.

**Task sampling.** Each participant solves one *medium*-difficulty LiveCodeBench task, randomly selected from a set of three possible candidates, treating *TaskId* as a blocking factor.

**Learning and fatigue.** A short warm-up precedes the evaluation task to reduce initial confusion; only one evaluation task per participant limits fatigue and carryover effects.

**Instrumentation.** Logging or timing errors could bias metrics (e.g., TimeToPass). We validate the logger in a pilot, store artifact hashes, and compute times from first test-suite production to passing (or budget).

**LLM non-determinism.** LLM outputs can vary due to inherent non-determinism. We will fix the prompt templates and key parameters (e.g., temperature, model endpoint) to maximize consistency across runs. Although exact reproducibility cannot be guaranteed in human–LLM interaction studies (due to model stochasticity and user behavior) we expect the outcomes to remain aligned across replications. Running several independent sessions allows to average out the variation, similar to individual differences observed in human-subject experiments.

**Researcher/experimenter effects.** Standardized instructions and scripted briefings are used; no assistance is given during tasks beyond clarifying the UI workflow.

### B. External Validity

**Population.** The sample may not represent all developers. We report demographics and caution when generalizing beyond similar profiles.

**Setting and tools.** Results reflect a single-monitor VS Code workflow, Python/unittest, and a fixed LLM. Generalization to other IDEs, languages, or models may be limited. However, the structured TDD-based approach should generalize conceptually, and the model choice will be documented for context.

**Task domain.** Using only *medium*-difficulty LiveCodeBench problems constrains scope. The findings may not be generalizable for easy/hard problems, real use-case scenario, such as in industry. However, LiveCodeBench provides a diverse set of algorithmic and data-structure tasks that are representative of common coding challenges.

**Ecological validity.** Disallowing external web search or auxiliary tools increases internal control but may under-approximate real workflows; we note this trade-off when interpreting efficiency results.

**Sample size.** A modest *N* limits precision; we prioritize descriptive summaries and only apply simple inferential analyses as warranted by data characteristics.

*Note.* Additional threats may emerge during the full execution; any newly identified risks and mitigations will be documented in the final study protocol.

## VIII. CONTRIBUTIONS AND IMPLICATIONS

This registered report introduces CURRANTE, a VS Code extension operationalizing an LLM-assisted, test-first workflow (Specification → Tests → Function) with per-test controls and automatic advice. We contribute: (i) a reproducible task protocol on LiveCodeBench; (ii) a fine-grained telemetry schema capturing time-stamped actions and process metrics; and (iii) a transparent study plan for human-in-the-loop TDD. All materials will be open-sourced for reuse and replication.

For research, this enables principled comparisons of LLM workflows and test curation strategies. For practice, results inform IDE design for safe *spec-by-tests* development. For education, CURRANTE provides scaffolded TDD to improve feedback cycles. Broadly, our methodology offers a reusable blueprint for evaluating emerging LLM coding tools with a focus on human–AI interaction and realistic constraints.

## ACKNOWLEDGMENT

The study presented in this paper was funded in part by the ADVISE project, funded by the Spanish AEI with reference 2024/00416/002.

## REFERENCES

- [1] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, “Productivity assessment of neural code completion,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 21–29. [Online]. Available: <https://doi.org/10.1145/3520312.3534864>
- [2] J. Anton Arhipov, “How to use a spec-driven approach for coding with ai,” <https://blog.jetbrains.com/junie/2025/10/how-to-use-a-spec-driven-approach-for-coding-with-ai/>, 2025, accessed: 3 Nov 2025.
- [3] N. S. Mathews and M. Nagappan, “Test-driven development and LLM-based code generation,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1583–1594.
- [4] S. Fakhouri, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, “LLM-based test-driven interactive code generation: User study and empirical evaluation,” *IEEE Transactions on Software Engineering*, 2024.
- [5] S. Piya, A. Samadi, and A. Sullivan, “Is more or less automation better? an investigation into the LLM4TDD process,” in *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 2025, pp. 161–168.
- [6] T. Ridnik, D. Kredo, and I. Friedman, “Code generation with alphacodium: From prompt engineering to flow engineering,” 2024.
- [7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [8] S. Piya and A. Sullivan, “LLM4TDD: best practices for test driven development using large language models,” in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 14–21.
- [9] S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. N. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, and E. Horvitz, “Guidelines for human-ai interaction,” in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–13. [Online]. Available: <https://doi.org/10.1145/3290605.3300233>
- [10] A. Sergeyuk, S. Titov, and M. Izadi, “In-ide human-ai experience in the era of large language models: a literature review,” in *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, ser. IDE ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 95–100. [Online]. Available: <https://doi.org/10.1145/3643796.3648463>
- [11] K. Nghiêm, A. M. Nguyễn, and N. Bùi, “Envisioning the next-generation ai coding assistants: Insights & proposals,” in *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, ser. IDE ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 115–117. [Online]. Available: <https://doi.org/10.1145/3643796.3648467>
- [12] G. Crupi, R. Tufano, A. Velasco, A. Mastropaoletti, D. Poshyvanyk, and G. Bavota, “On the effectiveness of llm-as-a-judge for code generation and summarization,” *IEEE Transactions on Software Engineering*, vol. 51, no. 8, pp. 2329–2345, 2025.
- [13] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, “Out of the bleu: How should we assess quality of the code generation models?” *Journal of Systems and Software*, vol. 203, p. 111741, Sep. 2023. [Online]. Available: <http://dx.doi.org/10.1016/j.jss.2023.111741>
- [14] C. Gao, X. Hu, S. Gao, X. Xia, and Z. Jin, “The current challenges of software engineering in the era of large language models,” *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, May 2025. [Online]. Available: <https://doi.org/10.1145/3712005>
- [15] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, “Livecodebench: Holistic and contamination free evaluation of large language models for code,” *arXiv preprint arXiv:2403.07974*, 2024.
- [16] P. Ralph, “ACM SIGSOFT empirical standards released,” *SIGSOFT Softw. Eng. Notes*, vol. 46, no. 1, p. 19, Feb. 2021.
- [17] A. Yang *et al.*, “Qwen3 technical report,” 2025. [Online]. Available: <https://arxiv.org/abs/2505.09388>