Journal of Database Management
Volume 37 • Issue 1 • January-December 2026

Benchmarking Software Test Cases Produced by Generative AI Against Traditional Methods:
A Design Science Approach

Mark L. Gillenson
http://orcid.org/0009-0002-1868-9611
University of Memphis, USA

Pavankumar Mulgund
http://orcid.org/0000-0001-8434-5070
University of Memphis, USA

Ankur Arora
http://orcid.org/0000-0002-9502-5547
University of Memphis, USA
Received: August 25th, 2025 | **Accepted:** April 28th, 2026

ABSTRACT

As artificial intelligence (AI) continues to shape commercial and governmental innovation, understanding the potential of large language models (LLMs) to improve software testing has become increasingly critical. This paper explores the application of LLMs for generating test cases—an essential yet resource-intensive activity in software quality assurance. Using theoretical lenses of the Design Science Research Evaluation (DSRE) framework, Measurement Theory, and Cognitive Load Theory, the authors systematically design and evaluate procedures for test case generation and execution to benchmark AI-generated test cases against those produced by human testers and traditional pairwise analysis. They develop two custom-built applications seeded with intentional defects and execute test cases generated through each method. This research contributes to the evolving discourse on AI-driven software engineering, offering insights into when LLMs can autonomously generate test cases, when human expertise is indispensable, and when hybrid approaches yield the greatest value.

KEYWORDS

Generative AI, LLMs, Software Testing, Test Cases, Design Science, Pairwise Analysis

INTRODUCTION

> *…is a far cry from claiming generative AI can write complete test automation so fast and so perfectly and done so instantly and autonomously that we can safely ignore all costs of that automation.*
>
> *— Blake Norrish, Sr. Director of Quality Engineering at Slalom, Medium Article*

Artificial intelligence (AI), which has been a subject of inquiry since at least the 1950s (McCarthy et al., 2006), has more recently evolved into a practical and impactful solution for various business contexts. From robot arms to facial recognition to fraud detection, AI has become pervasive and now

DOI: 10.4018/JDM.409970

offers a valuable set of artifacts for information systems and organizational applications (Russell & Norvig, 2021). In parallel, software testing remains a fundamental process within information systems development, ensuring that software artifacts meet both functional and non-functional requirements. Software testing is a critically important aspect of the software development process (Burnstein, 2003). Over the years, numerous software failures have occurred owing to inadequate, incorrect, or even totally absent software testing (Ewusi-Mensah, 2003). Functionality testing, which determines whether software conforms to the requirements and logic of an application, is the primary focus of testing, but other non-functional aspects include performance testing, security testing, and interoperability testing (Garousi et al., 2019). Furthermore, software testing must adapt to the development methodology in use. Thus, testing practices vary according to whether an application follows a traditional systems development life cycle or an agile development approach (Beck, 2003).

Software testing involves designing test cases alongside expected outputs to verify system behavior under specific input conditions (Panichella, 2018). This study focuses on functional testing, which assesses whether software behaves as specified in its requirements (Myers et al., 2004). Functional testing is appropriate in the business-critical systems context with applications such as package shipping and payroll because these applications rely on deterministic logic and rule-based processing, where correctness of outputs (e.g., shipment cost, payroll computation) is paramount. Unlike non-functional testing, which evaluates attributes such as performance or security, functional testing enables the systematic validation of input–output mappings against predefined specifications, thereby ensuring application reliability (Bertolino, 2007).

While large language models (LLMs) are often framed as technologies designed for natural language understanding and generation, software testing represents a fundamentally different—and particularly well-aligned—application domain. Unlike open-ended human language, software requirements and functional test cases are formal, rule-based, and deterministic, relying on explicit syntax, logical constraints, and verifiable input–output mappings. These properties closely mirror the structured token sequences and deterministic patterns that LLMs learn during training, making software testing a domain in which LLM capabilities can be evaluated with greater precision and objectivity than many natural-language tasks.

Within functional testing, the two traditional approaches for generating test cases are test-first development (TFD) and pairwise combinatorial analysis. Both methods aim to ensure that the software behaves as required. In TFD, test cases are created manually by those who define the application requirements, to anticipate and identify potential defects early in the development cycle (Beck, 2003). In contrast, pairwise analysis is a heuristic technique that ensures all possible pairs of input parameter values are covered. This approach provides broad functional coverage with fewer test cases (Kuhn et al., 2004). Although both methods contribute to the goals of functional testing, each has its limitations. TFD can be labor-intensive and prone to errors, whereas pairwise analysis does not ensure comprehensive coverage of complex input interactions. Furthermore, many years of the history of software development have demonstrated that the test cases produced by these traditional methods do not stop software defects from turning up in the finished code. Consequently, there is increasing interest in exploring alternative methods that are both more efficient and more effective in identifying functional defects.

Recent advances in generative artificial intelligence (GenAI)—a class of AI systems capable of producing text, code, images, and other outputs on the basis of learned patterns—have opened new possibilities for automating complex software engineering tasks. A prominent subset of GenAI is LLMs, which are deep learning models trained on vast corpora of text to understand and generate human-like language (Brown et al., 2020). LLMs, such as ChatGPT-4.0, have demonstrated promising capabilities in domains including natural language processing, code synthesis, and test case generation, making them powerful tools within the broader GenAI landscape.

Importantly, LLMs are trained not only with natural-language corpora but also using large-scale repositories of source code, test cases, and software specifications, exposing them to recurring testing