Journal of Computer Science

Original Research Paper

Intelligent Software Testing for Test Case Analysis Framework Using ChatGPT with Natural Language Processing and Deep Learning Integration

[1]Alaa Najmi and [1,2]Mohamed El-Dosuky

[1]Department of Computer Science, Arab East Colleges, Riyadh, Saudi Arabia
[2]Faculty of Computers and Information, Mansoura University, Egypt

Article history
Received: 11-10-2024
Revised: 30-11-2024
Accepted: 24-12-2024

Corresponding Author:
Mohamed El-Dosuky
Department of Computer Science, Arab East Colleges, Riyadh, Saudi Arabia
Email: maldosuky@arabeast.edu.sa

Abstract: Effective testing scenarios are necessary to guarantee the dependability and Caliber of software. Conventional techniques for creating these scenarios frequently involve a great deal of manual labor and might not fully cover all software requirements. In order to improve the automation and Caliber of software testing scenario development, this study investigates the combination of Natural Language Processing (NLP) and Deep Learning (DL) approaches with ChatGPT, an advanced language model by OpenAI. The suggested method automatically creates a variety of thorough test cases by utilizing ChatGPT's sophisticated natural language processing capabilities. To evaluate the model's capacity to comprehend intricate software requirements and generate pertinent situations, a comparison between conventional scenario-generation techniques and those improved by ChatGPT is carried out. The process is divided into four stages: Requirement parsing, in which natural language software requirements are analyzed and interpreted using NLP models; scenario generation, in which a transformer-based model is used to generate testing scenarios that are logical and appropriate for the environment. an automation pipeline that uses Hugging Face Transformers and Python to speed up the scenario generating process and evaluation metrics that evaluate the created scenarios according to requirement coverage and relevance coherence. The effectiveness of this method is illustrated through a case study on evaluating an Optical Character Recognition (OCR) system for private documents. The results show that integrating ChatGPT with NLP and DL greatly enhances the depth of testing scenarios, speeds up the generation process, and lowers manual labor. The potential of ChatGPT to automate and optimize software testing is demonstrated in this study, providing a more effective and flexible solution for a variety of testing scenarios.

Keywords: Software Testing, Intelligent Test Case ChatGPT, NLP, Deep Learning

Introduction

Thorough software testing is vital to ensuring the reliability and quality of software products. Recognizing the potential consequences of delivering faulty software to users, companies are cautious about releasing their products without a rigorous testing process. By employing meticulous testing strategies, organizations can minimize the risk of critical failures, usability challenges, or security vulnerabilities, which might otherwise lead to financial losses or erode customer confidence. Furthermore, identifying and addressing issues early in the development process significantly reduces long-term maintenance costs, reinforcing the importance of testing throughout the software lifecycle (Wang *et al*., 2024).

The importance of software testing has attracted a lot of interest from the academic and business worlds, and it is now a very busy and well-liked field of study in software engineering. The popularity of testing-related subjects at significant software engineering conferences and symposiums is evidence of this. These topics often account for the majority of submissions and are regularly chosen for

publication, highlighting their vital significance in furthering the profession (Wang *et al*., 2024).

High-quality software has grown much more in demand as software systems become more complicated and technology gets more deeply integrated across sectors. Robust software quality models are necessary for assessing and guaranteeing software efficiency, dependability, and overall excellence. These models offer structures for evaluating and enhancing the quality of software. One metric-based technique that is highly respected for its clear and organized evaluation methodology is the Quality Assessment Model (QAM), which is especially useful in a variety of development scenarios. In order to better understand the applicability and efficacy of existing software quality models in various scenarios, this study will examine and compile them (Yan *et al*., 2017).

The promise of modeling code was initially demonstrated by Hindle *et al*. (2016), and language modeling of code has since become a significant area of study in software engineering research. Researchers started using Deep Learning (DL) architectures to learn rich, hierarchical representations of code that could be used for a variety of downstream tasks as language modeling techniques advanced (White *et al*., 2015). Concurrently, the fields of machine learning and natural language processing initiated the construction of extensive models that were based on a particular category of neural architecture known as the transformer (Vaswani *et al*., 2017; Brown *et al*., 2020; Devlin *et al*., 2018), trained on extensive text datasets. The representational capability of these Large Language Models (LLMs), as well as language models designed expressly for code, was demonstrated through experiments (Lertbanjongngam *et al*., 2022; Feng *et al*., 2020; Allamanis *et al*., 2017; Chen *et al*., 2021).

An advanced Artificial Intelligence (AI) system known as an LLM has undergone significant training on a variety of data sources, such as books, code, articles, and webpages. To create cohesive material, these models-which include Generative Pre-trained Transformer-3 (GPT-3), Bidirectional Encoder Representations from Transformers (BERT), (Text-to-Text transfer transformer) T5 and extreme Multilingual language model (XLNet)-use the language's natural patterns and linkages. This comprises syntactically correct code snippets, human-like paragraphs, and grammatically perfect phrases. Due to their capacity to comprehend and produce contextually appropriate language, LLMs are becoming increasingly popular across a range of areas. They represent a significant improvement in machine learning and natural language processing (Ozkaya, 2023; Chang *et al*., 2024).

Nevertheless, until recently, these models were mainly limited to particular job environments and did not offer organic means of communication with end users.

Randoop and other random-based approaches are frequently used to automate the creation of test cases. The feedback-directed random testing methodology used by Randoop starts by producing arbitrary method call sequences. It assesses the results of these runs in order to direct the creation of further tests, highlighting untried approaches or routes. Randoop increases the variety and comprehensiveness of testing by keeping an eye on execution outcomes to make sure the program operates as intended and adjusting test cases in response to the behavior it observes (Pacheco and Ernst, 2007).

In a similar vein, new developments such as A3Test use assertion-augmented techniques to generate test cases automatically. By including assertions into the process, A3Test improves the quality of test cases that are generated, improving the identification of unexpected program behaviours and boosting the dependability of the testing results (Alagarsamy *et al*., 2024).

ChatGPT (OpenAI, 2024), an AI tool developed on top of pre-existing LLMs that allowed for communication through an interface, was released by OpenAI in late 2022. Using techniques from earlier work on InstructGPT (Ouyang *et al*., 2022), which trained LLMs with both unsupervised data and with supervision in the form of task instruction, OpenAI used reinforcement learning from human input to enable this kind of engagement. Essentially, the model was trained on actual human text-based conversations at first, and it subsequently learned to improve its responses based on feedback from human assessors who assessed the Caliber of responses in a reinforcement learning environment. The effort of developing an interface that allowed users to quickly access the latent "knowledge" of LLMs proved to be quite effective.

Code inspections, requirements inspections, compliance checklists, module and system testing, document inspections and testing, service testing, and distribution media testing are important methods for ensuring quality and identifying errors. By ensuring comprehensive validation across the whole development and deployment process, these approaches improve the dependability and consistency of software and associated deliverables (Jeanrenaud and Romanazzi, 2024).

While authors and reviewers are seated in the same room to review code updates, the purpose of code inspections is to identify errors. The most recent compendium of research on code inspection was created by Kollanus and Koskinen (2009). They discovered that empirical research accounts for the great bulk of code inspection investigations. It is widely agreed upon that code inspection is a valuable method for identifying defects and that using reading comprehension techniques to keep inspectors interested is also beneficial. As the internet has grown and asynchronous code review techniques have proliferated, research on code inspection has generally decreased after 2005.

DOI: 10.3844/jcssp.2025.1140.1155

Email-based asynchronous review process: The majority of sizable open-source software projects used remote, asynchronous reviews up until the late 2000s, depending on patches submitted to mailing lists and issue-tracking systems. Members of the project assess patches that have been contributed and request changes using these channels. The fix is committed to the codebase by core developers once it is judged to be of sufficient quality. Rather than performing pre-commit reviews, trusted committers may use a commit-then-review procedure (Rigby and Bird, 2013). Primarily, this form of review "has little in common (with code inspections) beyond a belief that peers will effectively find software defects," according to Rigby *et al*., who were among the first to conduct considerable work in this scenario (Rigby *et al*., 2014). In a similar situation, Kononenko *et al.* found that review response time and acceptance are correlated with social aspects that were absent from code inspections, such as reviewer load and change author experience (Kononenko *et al*., 2016).

Model of development that is pull-based: A developer who wants to make changes forks an existing Git repository and makes the changes in their fork through the GitHub pull request process (GitHub, 2016). A pull request that has been submitted gets added to the project's pull request list and becomes available to all project viewers. Analogous to earlier tool-based code evaluations, Gousios *et al*. qualitatively examined the work patterns and difficulties faced by pull-request integrators (Gousios *et al*., 2015) and contributors (Gousios *et al*., 2016).

Why is AI software testing necessary? New needs and incentives are brought about by the rapidly expanding AI software and the growing popularity of big data-based applications. AI-based features and functions will be incorporated into many programs both now and in the future. The methods and resources now in use are insufficient for testing AI-based features and capabilities. There aren't enough precise, experienced quality validation methods available, models, and standards for evaluation. Furthermore, there aren't many AI-based testing techniques or AI software solutions available. Therefore, the definition that follows explains what it means to test AI software. The term "testing AI software" describes a variety of testing procedures for AI-based systems and software. In order to facilitate test activities and meet pre-selected sufficient testing criteria and quality assurance standards, well-defined quality validation models, methodologies, techniques, and tools must be developed and deployed for AI-based software. As a result, evaluating AI features in software involves a variety of testing tasks to identify bugs in the product, confirm its functionality, and ensure that quality validation techniques need to be created. The purpose of testing is to ensure that the under-tested AI software satisfies quality assurance standards, pre-established testing criteria, and well-defined test requirements (Tao *et al*., 2019).

AI software testing should encompass commonly used intelligent features like recommendation, recognition, and prediction, as it is constructed using a variety of machine learning models and data-driven technologies. The main area of AI software testing is depicted in Fig. (1). AI software testing includes a significant amount of testing, including objects (human and animal), such as object identification, recognition, and behavior detection. Current significant AI testing subjects include a variety of intelligent applications, including question-and-answer capabilities, analytics and prediction capabilities, intelligent commands and actions, business decisions, recommendations, and selection (Yin *et al*., 2018; Sun *et al*., 2017; Qi *et al*., 2018) and analytics and prediction capability (Sun *et al*., 2019; Yin *et al*., 2017; Qi *et al*., 2017; Nakao and Eschbach, 2008). Furthermore, a significant challenge for AI testing and quality validation will be figuring out how to do control validation and healthcare checks, given the development of unmanned vehicles and their potential large markets. Furthermore, context-related problems with AI software typically include scenario, location (Yin *et al*., 2018), time, and stakeholder difficulties. This leads to new testing challenges with context identification and classification. The following is a summary of AI software testing's main objectives.

Software testing has a big impact on quality assurance and cost control. The "Cost of Poor Software Quality in the US" paper, for instance, emphasizes how inadequate testing may lead to significant costs, especially in vital systems where human safety is at stake. These mistakes cause long-term financial and operational inefficiencies, which postpone the software's delivery. Bugs are far less expensive to find and solve early in development than they are to fix later in production (Tuteja and Dubey, 2012; Krasner, 2021). These expenses can also be decreased by using quality assurance procedures meant to stop errors, such as testing at every stage of the Software Development Life Cycle (SDLC). Businesses may reduce maintenance costs and improve software quality over time by utilizing automated testing and proactive quality management (Tuteja and Dubey, 2012).

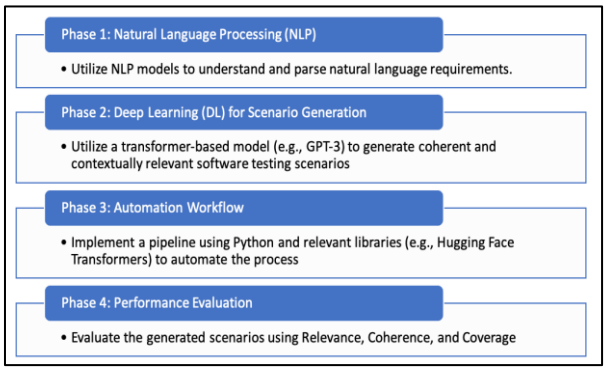

Fig. 1: Proposed methodology

Test case selection has been optimized using a variety of techniques, including meta-heuristic algorithms like Genetic Algorithms (GA) and Particle Swarm Optimization (PSO). By increasing each test case's efficiency, these methods seek to lower testing expenses while also enhancing the overall quality of the program. To ensure that the most important tests are prioritized, PSO may modify the "weight" or importance of each test case. On the other hand, GA can optimize test cases by experimenting with different combinations to find the most effective ones. By identifying flaws early in the development process, automating the selection and execution of these optimized test cases can improve software performance after release and lower the expenses related to subpar software quality (Furtado *et al*., 2016; Prakash *et al*., 2020).

The most common problem category, according to research on bug repair in machine-learning libraries, was bugs, with a sizable portion of reported issues being incorrectly categorized. It highlighted the necessity of improved issue management to shorten resolution timeframes and proposed that streamlining procedures and enlisting additional contributors should speed up bug resolution (Ajibode *et al*., 2023). In a similar vein, studies on software maintenance have shown how crucial adaptive and corrective maintenance is to bug management. Reiterating that efficient bug identification is essential for software quality, particularly in production contexts where undiscovered flaws may seriously harm operations and reputation, it covered how new issues might occur when resolving old ones (Bello and Tobi, 2024).

The potential of ML techniques to automate test case selection and prioritizing, improving testing efficiency, was investigated in a recent literature review. In particular, machine-learning approaches that rank test cases according to factors like coverage, test performance, and historical data are advantageous for regression testing and other methodologies (Pan *et al*., 2022). Numerous studies have addressed important issues that developers and testers confront, including test case selection, priority, and optimization. A systematic review on test case prioritization, for example, stresses the significance of ranking test cases, especially in regression testing, and stresses the importance of prioritization in managing resource constraints during software maintenance and guaranteeing software reliability (Mehmood *et al*., 2024; Singhal *et al*., 2021).

Additionally, studies indicate that machine learning may optimize test suites, resulting in better coverage, lower costs, and better test case selection. This is particularly crucial for decision-making processes like figuring out which test cases have the biggest effects and when to quit testing. Furthermore, unfixed flaws can result in serious operational concerns. Therefore, testing methods need to change to reflect the ever-changing landscape of software development (Mehmood *et al*., 2024; Singhal *et al*., 2021).

The core issue of software testing, "What is a test data adequacy criterion?" is highlighted by the guidelines for determining whether a software system has undergone sufficient testing or software test adequacy criteria. Numerous test data adequacy criteria, including data flow-based, fault-based, error-based, and control flow-based test adequacy criteria, have been put out and examined in the literature. Statement coverage, branch coverage, path coverage, length-i path coverage, loop coverage, relational operator coverage, table coverage (if every entry in a given array has been referenced), and the cyclomatic number criterion are examples of control flow-based adequacy criteria. All definitions and use criteria are part of the data-flow-based adequacy criteria. The criteria for fault-based adequacy encompass error seeding, mutant coverage, and mutant killing score. Every criterion has advantages and disadvantages of its own. How test adequacy criteria relate to fault detection capacity is a key subject in the research on test adequacy criteria (Nakao and Eschbach, 2008; Qu *et al*., 2008).

The quantity of defects discovered and fixed before the system's release determines how successful this verification and validation process was. This, in turn, is dependent on how well the test cases are created. An input for the software being tested is called a test case. It is a collection of parameters or circumstances that a tester uses to assess whether or not a software system or application is operating as intended. It is the process of figuring out whether a system or software application has succeeded or failed. Finding program input that satisfies testing criteria is the process of data production for software testing. Test data generators employ two distinct methodologies, which are path-oriented and goal-oriented (Nakao and Eschbach, 2008; Xin *et al*., 2007). Creating error-free software typically requires a large number of test cases. Since thorough testing is not feasible, the test cases that are produced should be as good as possible, covering the whole program and identifying as many flaws as they can. One of the challenges with this technique is the automatic production of optimum test scenarios (Tallam and Gupta, 2006).

A test suite sometimes called a validation suite, is made up of a number of test cases intended to confirm how a software program works and behaves under particular circumstances. Instructions for setting up the system, test case goals, and precondition information required for their execution are usually included in these suites. Current research emphasizes the difficulties in keeping test suites effective, especially when it comes to reducing superfluous or duplicate test cases, which can drive up testing expenses. (Mehmood *et al*., 2024) Address the utilization of machine learning methods to optimize test suites, emphasizing the removal of redundancies and the improvement of test coverage through the utilization of prediction models and historical data.

In addition, Singhal *et al.* (2021) offer a methodical analysis of test case prioritization and selection, focusing on resource-efficient and dynamic testing techniques to guarantee successful validation while lowering expenses. These revelations highlight the significance of test case optimization for improving software testing procedures' overall effectiveness and guaranteeing the best possible use of available resources. Furthermore, (Kiran *et al*., 2019) offer a thorough analysis of contemporary trends, tools, and methodologies for test suite optimization, highlighting the necessity of better methods to lessen testing-related financial pressure. According to previous performance statistics, (Bagherzadeh *et al*., 2022) also emphasize the potential of reinforcement learning approaches for test case prioritization, which optimizes testing procedures by concentrating on the most important tests (Kiran *et al*., 2019; Bagherzadeh *et al*., 2022).

Finding a subset of test cases that reduces redundancy while meeting the same coverage criteria as the original suite is known as "minimizing" a test suite. Since this problem is computationally demanding (NP-complete), heuristic approaches are frequently used. For example, optimization has been successfully accomplished through the use of evolutionary algorithms in conjunction with mutation testing. Using mutation testing to increase test suite efficiency, (Zheng *et al*., 2017) presented a many-objective evolutionary optimization approach that shows promise in reducing test case redundancy while maintaining essential test coverage.

Problems of a peculiar character are ones that need an unusual combination of data-driven and knowledge-driven methods to be solved. Test case optimization is a search space problem for which a hybrid approach combining data-driven and knowledge-driven methods is needed to achieve a close-to-optimal solution. Thus, test case optimization is a unique kind of challenge (Berndt and Watkins, 2005).

A subset of AI known as ML has been transforming various fields in the previous several decades, having begun in the 1950s. A branch of machine learning called Neural Networks (NN), from which DL originated, has introduced and produced ever-larger disruptions while demonstrating remarkable success in nearly every application domain. ML techniques classified as DL (deep architecture of learning or hierarchical learning approaches) were primarily developed in 2006. Estimating model parameters is a step in the learning process that enables the learned model or algorithm to carry out a particular task. For instance, the weight matrices ($w\ i,\ 's$) are the parameters of Artificial Neural Networks (ANN). On the other hand, DL has multiple layers that sit between the input and output layers. These layers enable the use of multiple hierarchical designs for non-linear information processing units, which are then used for feature learning and pattern categorization (Schmidhuber, 2015; LeCun *et al*., 2015).

Representation learning is another term for a learning approach based on data representations (Bengio *et al*., 2013). According to recent research, representation learning based on DL entails a hierarchy of features or concepts, where low-level concepts can be defined from high-level ones and high-level concepts from low-level ones. As DL is not task-specific, it has also been referred to in some articles as a universal learning strategy that can address nearly any type of issue in a variety of application areas (Bengio, 2009; Alom *et al*., 2019).

In its sixth round, the Message Understanding Conference (MUC) introduced the idea of Named Entity Recognition (NER) (Grishman and Sundheim, 1996). Scholars have persisted in classifying alignment into increasingly discrete categories, including places, people, and proper nouns (Lee and Lee, 2005). People can be classified into a variety of groups, including politicians, entertainers, and associations (Liu *et al*., 2022). Artificial or manufactured languages, such as computer languages, are not considered to be part of the category of natural languages spoken by people. Computational methods for computer-assisted natural language processing are included in NLP. Since voice is frequently used without being excluded, NLP is introduced to include speech and other NLP components (Hannan *et al*., 2012). natural language understanding, machine translation, speech recognition, voice synthesis, and other subfields make up NLP (Saetre, 2006).

DL is a subfield of machine learning, which is a subfield of AI (Sze *et al*., 2017). The superior performance of DL comes from its ability to extract high-level features from raw data.

Related Work

ChatGPT's potential as an extensive language model for software testing instruction. While some instructors are concerned that students may abuse these technologies, others see them as an opportunity to design creative and engaging learning environments. Researchers assessed the capacity of ChatGPT to respond to practice questions from a well-known software testing textbook. The majority of the questions (77.5%) could be answered by ChatGPT, although its accuracy wasn't perfect-it answered questions correctly or somewhat correctly 55.6% of the time. Similar percentages of explanations (53% correct or somewhat correct) were provided. It's interesting to note that ChatGPT's Accuracy increased little when a series of related questions were asked. The model's degree of confidence in its responses, meanwhile, didn't appear to be correlated with Accuracy. This study offers a dataset of ChatGPT replies and an evaluation of its advantages and disadvantages, which will be helpful for future research. Along with exploring the pedagogical implications of employing LLMs in education, the authors provide resources to assist others in replicating their work. The study's overall conclusion is that ChatGPT has

potential as a teaching tool, but before it is widely used in software testing instruction, its drawbacks should be carefully considered (Jalil *et al*., 2023).

Its feasibility is in leveraging LLMs like ChatGPT to automatically generate test cases from informal bug reports. It addresses a significant challenge in software development: Creating test cases for problems that users report that are often too complex and challenging for traditional test creation methods to handle. The researchers created test cases from the Defects4J dataset using ChatGPT and codeGPT to illustrate the potential of LLMs in this field. Automating the creation of test cases, this technique could speed up the software development process and improve the efficacy of identifying and fixing issues (Plein *et al*., 2024).

The potential of ChatGPT as a helpful AI language model for software developers. It investigates how AI might improve upon normal software development processes, which typically entail a number of steps such as requirement analysis, design, coding, and testing. AI offers skills like pattern recognition and decision-making to expedite these processes. The study, which focuses on ChatGPT's support for developers, finds that it can be a helpful tool for saving time, optimizing workflow, and testing automation. The paper does mention the need for more research and responsible AI deployment to address potential ethical concerns. Taking everything into account, the research indicates that ChatGPT and other AI technologies have the potential to drastically change software development by boosting output, creativity, and quality (Özpolat *et al*., 2023).

How ChatGPT, a powerful language model, may enhance software testing using Metamorphic Testing (MT). Using relationships, or metamorphic relations, between inputs and outputs, MT is a technique that checks programs for correctness. The most difficult task in MT is locating new MRs. The study's goal is to ascertain whether ChatGPT can generate precise MRs for a variety of software systems, including ones that have never undergone MT testing. The results show that ChatGPT is capable of generating novel MRs; nevertheless, human intelligence is still needed to verify and fine-tune them. In summary, ChatGPT can be a helpful tool for enhancing software testing intelligence as it suggests MR candidates that could be used for test implementation (Luu *et al*., 2023).

Recently, LLM-based techniques for automated code creation have gained popularity. These techniques use the transformer architecture (Vaswani *et al*., 2017; Radford *et al*., 2018; Black *et al*., 2021; Achiam *et al*., 2023). According to Chen *et al*. (2021), codex is a decoder-only language model that has been refined using publicly accessible code and has remarkable programming abilities. In the meanwhile, an encoder-decoder architecture with identifier-aware pre-training tasks based on T5 is used in CodeT5, which was presented by Wang *et al*. (2021a); Raffel *et al*. (2020). AlphaCode, which employs reinforcement learning and test case execution to train models that can perform at a beginner level on the Codeforces platform, is another noteworthy addition (Li *et al*., 2022). Furthermore, code generation benchmarks, such as WizardCoder and InCoder (Fried *et al*., 2022), have been significantly improved by open-source LLMs trained on code (Li *et al*., 2023; Roziere *et al*., 2023; Luo *et al*., 2023).

To improve code ranking, one must have a strong understanding of code. The main components of code understanding are demonstrated by tasks including functionality classification, clone identification, and code search (Wang *et al*., 2020; Gu *et al*., 2021; Arakelyan *et al*., 2022; Li *et al*., 2024). By using pre-trained language models to capture meaningful representations and modeling code as a series of tokens, early methods concentrated on enhancing code representation (Kanade *et al*., 2020; Wang *et al*., 2021b).

In programming and natural languages, CodeBERT was the first pre-trained encoder-only architecture with a token replacement detection objective (Feng *et al*., 2020). GraphCodeBERT is an extension of this technique that uses structure-aware pre-training tasks to extract semantics from source code and data flow (Guo *et al*., 2020). A technique for transforming AST into sentences was presented in order to utilize AST data in a manner akin to that of natural and computer languages (Ahmad *et al*., 2021). To improve representation learning, UniXcoder used multimodal multimodal information, including AST, comments, and code fragments (Guo *et al*., 2022; Lyu *et al*., 2025). Table (1) provides a summary of the literature review.

Table 1: Summary of literature review

Study focus	Key findings	Implications	Reference
ChatGPT in software testing instruction	Answered 77.5% of textbook questions; 55.6% were correct or somewhat correct. Accuracy improved slightly with related questions. Confidence levels didn't correlate with Accuracy.	Supports interactive, self-directed learning Needs ethical guidelines to prevent misuse Promising for creating engaging learning environments	Jalil *et al*. (2023)
Test case generation from bug reports	ChatGPT and CodeGPT generated test cases from informal bug reports using the Defects4J dataset. Automates complex test case creation.	Speeds up the software development lifecycle. Enhance bug reporting and resolution efficiency.	Plein *et al*. (2024)
Improving software development workflow	ChatGPT supports tasks like requirement analysis, coding, and testing. Offers pattern recognition and decision-making capabilities	Saves time, optimizes workflows, and improves software quality. Requires responsible deployment to address ethical concerns	Özpolat *et al*. (2023)

Enhancing Metamorphic Testing (MT)	ChatGPT generates novel Metamorphic Relations (MRs) for testing but needs human validation. Facilitates MT application to untested systems	Reduces manual effort in creating MRs. Encourages AI-human collaboration in intelligent software testing	Luu *et al*. (2023)
LLMs for code understanding and generation	Codex, CodeT5, and AlphaCode exhibit coding abilities and competitive performance. CodeBERT, GraphCodeBERT, and UniXcoder enhance code representation with structural and semantic insights	Supports tasks like code completion, bug fixing, and clone detection. MultimodalMultimodal integration advances code understanding	Vaswani *et al*. (2017); Radford *et al*. (2018); Wang *et al*. (2021a); Li *et al*. (2022); Feng *et al*. (2020)

Materials and Methods

An intelligent software testing framework that incorporates ChatGPT, NLP, and DL is proposed in this paper using a theoretical and conceptual approach. There were no tangible tests or instruments utilized. The research was carried out using:

1. A thorough analysis of current research using sources like Google Scholar, IEEE Xplore, and the ACM Digital Library to examine the use of LLMs in software testing
2. Conceptual modeling of a framework that uses ChatGPT to develop adaptive test scenarios, comprehend software requirements, and minimize manual testing effort.
3. Design guidance is provided by referencing established AI models like Codex, CodeBERT, and AlphaCode

The research problem can be stated as follows. In the field of software development, building thorough and efficient test cases is essential to guarantee the software's functioning and quality. However, there are some obstacles that developers must overcome in this process, such as:

1. Complex requirements: It can be challenging to develop thorough and precise test cases due to the complexity and variety of software requirements
2. Inefficiency in manual generation: Producing test cases by hand can be laborious and inefficient, frequently resulting in insufficient coverage of possible use cases
3. Lack of adaptability: Manually generated test cases might not be able to alter quickly in response to updates or modifications to the product

This study proposes a new approach, which combines ChatGPT with DL and NLP methods and aims to tackle the aforementioned issue. By using sophisticated AI-driven test case production and optimization techniques in conjunction with intelligent requirement analysis, this methodology seeks to improve the precision and efficacy of test case creation:

1. Analyzing software requirements:
    - NLP: Software requirement documents can be analyzed and understood by applying NLP techniques. In order to retrieve pertinent data and context, textual data must be parsed. Important NLP jobs consist of:
        - Named Entity Recognition (NER): Recognize and categorize textual entities, such as system interactions, user roles, and functional components. This aids in comprehending the precise components and relationships that require testing
        - Dependency parsing: Determine the connections between the various needs' components by examining the grammatical construction of sentences
        - Semantic analysis: To make sure that test cases cover all required scenarios, comprehend the meaning and intent behind requirement descriptions
2. Generating test cases with ChatGPT:
    - Intelligent test case building: Apply ChatGPT to create test cases that are informed by the knowledge gleaned from NLP analysis. Based on a thorough understanding of the requirements, ChatGPT uses design prompts to help it develop test cases that cover a variety of topics, including functionality, security, performance, and integration
3. Integrating DL techniques:
    - DL models: DL methods are used to improve test case generation, optimization, and assessment. This comprises:
        - Convolutional Neural Networks (CNNs) or Deep Neural Networks (DNNs): Utilize these models to assess test case quality and forecast efficacy by drawing on past performance and trends
        - Predictive analytics: Prioritize and improve the generated instances by using predictive models to predict the possible impact and coverage of the test cases
4. Evaluating and analyzing test cases:
    - Quality metrics: Establish measures to assess the test cases' efficacy, such as:

DOI: 10.3844/jcssp.2025.1140.1155

- Coverage: Evaluate the extent to which the various software requirements are covered by the test cases
- Accuracy: Measure the precision of the test cases in identifying defects or issues
- Predictive power: Evaluate the ability of test cases to anticipate potential issues based on historical data
- Automated analysis: To evaluate test results and assess the test cases' efficacy, conduct automated analysis. Utilize insights derived from data to direct future enhancements

5. Continuous improvement:

- Interactive feedback: Establish a feedback mechanism to get developers' opinions on the efficacy and Caliber of the test cases. Examine this feedback so that you can make data-driven adjustments.
- Model enhancement: To guarantee constant improvement and Accuracy, ChatGPT and DL models should be updated frequently depending on user feedback and performance evaluations

The benefits are:

- Enhanced Accuracy: Apply DL and advanced natural language processing techniques to test cases to increase their thoroughness and precision
- Enhanced productivity: Optimize the productivity of the test case development and assessment procedures

Allow for flexible adaptation of the approach to meet the requirements of various testing scenarios and application kinds.

By presenting an intelligent software testing framework that combines ChatGPT with NLP and DL techniques to improve test case development and analysis, the research investigates a fresh approach to software testing. By utilizing cutting-edge AI capabilities, the framework seeks to overcome the drawbacks of conventional techniques by increasing the precision and effectiveness of test case production. In particular, it will make use of NLP techniques like dependency parsing and Named Entity Recognition (NER) to thoroughly examine and comprehend software requirements, guaranteeing that all important aspects are taken care of.

Based on these in-depth insights, ChatGPT will be used to dynamically create test cases that are continuously adjusted based on real-time feedback. Predictive analytics will be used to analyze the impact and coverage of the test cases, and DL models will be integrated to maximize their quality.

In addition to more conventional metrics like coverage and Accuracy, the research will concentrate on creating new evaluation criteria, including contextual relevance and flexibility. A combination of quantitative metrics and qualitative input will be used to evaluate the efficacy of this novel technique in order to pinpoint areas in need of development and offer practical suggestions for integrating AI into test case production procedures. To optimize the effectiveness of the framework and overall testing quality, important difficulties such as guaranteeing the Accuracy of input prompts and connecting the framework with current testing tools will be addressed.

Table (2) shows the comparison of traditional and intelligent methods for test case generation, highlighting the advantages of integrating AI technologies like ChatGPT, NLP and DL.

Table (3) shows the presented evaluation metrics for ChatGPT, NLP and DL-based intelligent test case frameworks, evaluating coverage, Accuracy, adaptability, relevance, efficiency and developer satisfaction.

Table (4) shows the highlights of the integration of NLP and DL technologies in test case generation, highlighting their effectiveness in improving quality and efficiency.

Table (5) shows the challenges in integrating the intelligent test case framework, proposing solutions like precision, consistency, and resource investment for smoother implementation and improved test case generation.

Table 2: Comparison of traditional vs intelligent test case generation

Criteria	Traditional methods	Intelligent framework (ChatGPT + NLP + DL)
Test case generation	Manual, frequently requiring a lot of time	Dynamic, automated, and based on insights from AI and NLP
Requirement analysis	Manual analysis and interpretation	Automated NLP analysis, such as dependency parsing and NER
Adaptability	Static and restricted to preconceived notions	Dynamic, responsive to immediate response and evolving needs
Coverage	Often constrained, may overlook special circumstances	Thorough and includes a wider variety of situations
Consistency	Varied and based on personal experience	Consistent with generation and analysis powered by AI
Efficiency	Laborious and time-consuming	High productivity via integration of AI and automation
Integration	Distinct from the testing instruments	Integrates well with test management technologies that are already in place
Evaluation metrics	Coverage, Accuracy	Contextual relevance, Accuracy, flexibility, and coverage
Cost	Lower starting cost but more labor over time	Greater setup costs upfront and possible continuing expenses for AI tools

Table 3: Evaluation metrics for intelligent test case framework

Metric	Description	Measurement method	Target values
Coverage	The degree to which test cases encompass various circumstances	The percentage of requirements met	90% or higher
Accuracy	The precision of created test cases in finding problems	Comparing the number of problems found to the real defects	Greater than 80% association
Adaptability	The adaptability of the framework to new or modified requirements	Time needed for test case adaptation and updating	<24 h
Contextual relevance	How closely do test scenarios match the real requirements	Developer feedback's relevance score	High degree of importance (>4/5)
Efficiency	The amount of time and materials needed to create and run test cases	The average amount of time for each test case	At least a 50% reduction
Developer satisfaction	The framework's perceived efficacy and ease of use by developers	The survey of feedback	Positive comments (more than 80%)

Table 4: Example of NLP and DL integration in test case generation

Component	Function	Technology used	Output
NLP analysis	Extract and evaluate the specifications	NER and dependency parsing	Thorough comprehension of the requirements
ChatGPT integration	Create test cases using the insights from NLP	GPT-4 or later versions	Test cases for various scenarios
DL models	Assess and enhance the created test cases	CNNs and DNNs	Evaluation and improvement of quality
Predictive analytics	Estimated efficiency and test case coverage	Models of prediction and historical data	Prioritization and impact analysis

Table 5: Challenges and solutions

Challenge	Description	Proposed Solution
Prompt Precision	Incorrect prompts can result in suboptimal test cases	Improve prompt generation techniques and offer precise instructions
Contextual Understanding	The individual is experiencing difficulty comprehending complex requirements.	Improve NLP models to improve semantic analysis
Consistency	The text explains the concept of variability in the generation of test cases.	Create standardized generating processes and prompt templates
Integration Complexity	The integration with existing tools is currently facing challenges	Make integration guidelines and API interfaces to ensure a smooth connection
Resource Investment	The initial setup and licensing costs are high	Evaluate cost-benefit analysis and look into options that are more affordable.

With ChatGPT's integration of NLP and DL, software test case development has improved significantly by combining cutting-edge AI capabilities with conventional approaches. While ChatGPT is excellent at quickly creating a variety of test cases and adjusting to changing needs, the quality of the underlying NLP models and the clarity of input prompts have a significant impact on how effective ChatGPT is. Notwithstanding its advantages, ChatGPT could have trouble capturing the subtle understanding needed in complex situations. On the other hand, conventional techniques offer the breadth of human judgment and all-encompassing coverage required for exhaustive testing. This implies that ChatGPT should be viewed as an additional tool that improves conventional testing methods rather than as a replacement when used in conjunction with NLP and DL. This hybrid approach offers a more robust, efficient, and accurate software testing process while preserving the crucial oversight and validation supplied by experienced engineers by fusing the sophisticated capabilities of AI with tried-and-true testing procedures.

Test Cases

System test cases are tabulated in Table (6), while acceptance test cases are tabulated in Table (7). Table (8) provides unit test cases, while Table (9) provides performance test cases. Table (10) lists the security test cases.

To make test cases for a task management system, let us bring this notion to ChatGPT and request that it do so. To ensure the robust functionality of a task management system, various types of testing are employed. System testing focuses on verifying that the entire system operates correctly, including user logins, task creation, and task details display. For instance, tests might check if users can log in and access their task dashboards or if newly created tasks are accurately displayed. Acceptance testing validates that the system meets user requirements, such as confirming that tasks can be created with all necessary fields and correctly assigned to users. Unit testing

examines individual components, such as functions responsible for task creation and status updates, ensuring they work as intended in isolation. Performance testing evaluates the system's ability to handle various loads, measuring response times for task creation and system behavior under high user activity to ensure efficiency and scalability. Security testing addresses potential vulnerabilities, ensuring unauthorized users cannot access tasks and that data protection mechanisms are in place to prevent injection attacks. Each type of testing plays a crucial role in delivering a reliable and user-friendly task management system.

Table 6: System Test Cases

Test case ID	Description	Preconditions	Test steps	Expected result
ST001	Verify user can log in and access the task dashboard	The user is registered and has valid credentials	1. Navigate to the login page 2. Enter a valid username and password 3. Click "Login"	The user successfully logs in and is redirected to the task dashboard
ST002	Verify task details are correctly displayed	The user is logged in and has tasks assigned	1. Navigate to the task dashboard 2. Select a task to view 3. Verify task details (title, description, due date)	Task details are correctly displayed as per the information in the system
ST003	Verify task can be created and appears in the list	The user is logged in	1. Navigate to the "Create Task" page 2. Enter valid task details 3. Click "Save". 4. Navigate to task list	A new task with the correct details appears in the task list.

Table 7: Acceptance test cases

Test case ID	Description	Preconditions	Test steps	Expected result
AT001	Validate task creation with all required fields	The user is logged in	1. Navigate to "Create Task" 2. Enter title, description, due date 3. Click "Save"	The task is created with all fields properly filled and visible in the task list
AT002	Validate task assignment to a specific user	The user is logged in and has permissions	1. Navigate to "Assign Task" 2. Select a task 3. Choose a user from the list 4. Click "Assign"	The task is assigned to the selected user and appears in their task list
AT003	Verify task status update functionality	The user is logged in and has tasks	1. Navigate to the task list 2. Select a task 3. Change the status to Completed 4. Click "Save"	The task status has been updated and reflected in the task list.

Table 8: Unit test cases

Test case ID	Description	Preconditions	Test steps	Expected result
UT001	Verify task creation function	The system is set up and running	1. Call the create task () function with valid parameters 2. Check the return value and task list	A task is created and appears in the task list
UT002	Verify task assignment function	The system is set up and running	1. Call the assigned task () function with a valid task ID and user ID 2. Check the task assignment details	The task is assigned to the correct user
UT003	Verify task status update function	The system is set up and running	1. Call the update task status () function with a valid task ID and status 2. Check the task status	The task status is updated correctly

Table 9: Performance test cases

Test Case ID	Description	Preconditions	Test steps	Expected result
PT001	Measure response time for task creation	The system is set up and has baseline performance data	1. Measure the time taken to create a task with valid details 2. Perform multiple task creations in succession	Response time is within acceptable limits (e.g., < 2 sec)
PT002	Assess system performance under load	The system is set up, and performance testing tools are available	1. Simulate multiple concurrent users creating and updating tasks 2. Monitor system performance and response times	The system maintains performance and stability under load

DOI: 10.3844/jcssp.2025.1140.1155

PT003	Verify system scalability	The system is set up, and baseline performance data is available	1. Gradually increase the number of tasks in the system 2. Monitor performance metrics (e.g., response time, CPU usage)	The system scales appropriately with increasing task volume without significant performance degradation

Table 10: Security test cases

Test case ID	Description	Preconditions	Test steps	Expected result
SCT001	Verify unauthorized users cannot access tasks	The user is logged out	Attempt to access task management features without logging in	Access is denied, and the user is redirected to the login page
SCT002	Verify task data is protected against injection attacks	The user is logged in	1. Attempt to input SQL injection or other malicious code in task fields 2. Submit and check for system behavior	Input is sanitized; no injection attack succeeds
SCT003	Check for proper session management	The user is logged in	Log in and then attempt to access tasks from a different browser or device	Session management prevents unauthorized access

Table 11: Scenario and metrics

Scenario	Objective	Relevance	Coherence	Coverage
Scenario 1: Text extraction accuracy in different fonts	Verify the OCR system's Accuracy in extracting text from documents using various fonts	High	High	Medium
Scenario 2: Handling redacted information	Test the OCR system's ability to correctly identify and ignore redacted portions while extracting text.	High	High	High
Scenario 3: Recognition of handwritten annotations	Evaluate the OCR system's performance in recognizing handwritten notes or annotations	Medium	High	Medium
Scenario 4: Security of OCR output storage	Ensure secure storage of extracted text from confidential documents with encryption and access control.	High	High	High
Scenario 5: OCR performance on multi-language documents	Test the OCR system's capability to accurately recognize and extract text from multi-language documents	High	High	High

Case Study

This section provides a case study on testing a system for Optical Character Recognition (OCR) for confidential documents. Table (11) shows that it provides a clear overview of the test scenarios, helping prioritize which areas to focus on based on relevance, coherence, and coverage.

Scenarios 1, 2, 4, and 5 are highly relevant, directly targeting critical aspects of OCR for confidential documents. Scenario 3 is slightly less relevant but still important. All scenarios are coherent, with logical steps that align with the testing goals. Scenarios 2, 4, and 5 provide high coverage by addressing key edge cases and system capabilities. Scenarios 1 and 3 offer medium coverage as they focus on specific aspects.

Results and Discussion

This paper offers a theoretical framework for intelligent software testing that supports and improves test case analysis by combining ChatGPT, DL, and NLP approaches. Despite the lack of empirical implementation or evaluation, the framework's potential efficacy can be deduced from existing literature and conceptual analysis.

1. Assessment of the Proposed Framework Conceptually

Utilizing the contextual awareness and reasoning powers of large language models (LLMs) to validate software test cases. The system's objectives are to detect test coverage gaps, automate the creation of test scenarios, and improve software testing decision-making by using natural language inputs.

Although actual outcomes have not yet been obtained, the concept theoretically provides:

- Improved completeness of test cases using clever recommendation systems
- Automating portions of the testing process reduces the amount of manual labor required.
- Context-aware language processing for enhanced flexibility across project areas

The results of earlier research that examined LLMs in software engineering settings are in excellent agreement with these theoretical advantages.

2. Comparing with Related Research

Recent research results bolster the justification and anticipated benefits of the suggested framework:

- ChatGPT's ability to respond to inquiries about software testing was emphasized by Jalil *et al*. (2023), who also suggested that it may be useful in learning settings and possibly in automated reasoning while designing tests
- Plein *et al*. (2024) showed how to utilize ChatGPT and CodeGPT to create test cases from bug reports, confirming that LLMs can handle unstructured or informal material to create structured test artifacts. Our framework also contemplates this method
- According to Özpolat *et al*. (2023), ChatGPT can assist with a number of development activities, including testing. This reinforces one of the tenets of our suggested approach, which is the wider involvement of AI in the software lifecycle
- A key concept in the current work is the potential for collaborative AI-human testing settings, which Luu *et al*. (2023) demonstrated can help with Metamorphic Testing by proposing novel relations
- The use of models like Codex, CodeT5, and CodeBERT to improve test case quality, fault detection, and software reliability has been validated by Vaswani *et al*. (2017), Radford *et al*. (2018), Wang *et al*. (2021a), Li *et al*. (2022), and Feng *et al*. (2020). These models have demonstrated strong code reasoning and generation abilities

3. Insights & Future Paths

Software testing procedures could be greatly enhanced by combining ChatGPT and deep learning into a single testing framework, according to the report. Further empirical research is necessary to test these assumptions and improve the framework, though, as the study is still in its theoretical phase, some potential avenues for future investigation are:

- Applying the framework to actual software testing settings
- Comparing its performance to both conventional and AI-based testing instruments
- Investigating domain-specific modifications and optimizing models for particular situations
- Looking into ethical issues and explainability in software testing with AI assistance

In summary, this work offers a systematic and progressive vision for intelligent software testing while being theoretical. It establishes a framework for upcoming experiments and implementations to enhance AI's function in software quality assurance.

Conclusion

This study explores the use of NLP and DL with ChatGPT, an OpenAI language model, to enhance software testing scenarios. The study demonstrates ChatGPT's ability to understand complex software requirements, generate relevant scenarios and adapt to different environments. It suggests that ChatGPT can reduce human labor, produce more reliable testing procedures and expedite scenario development.

One possible future direction is to fine-tune a GPT model on specific domain data for better scenario generation. Another future work is to integrate with testing frameworks like Selenium or Pytest to dynamically generate and execute tests.

Acknowledgment

Thank you to the publisher for their support in the publication of this research article. We are grateful for the resources and platform provided by the publisher, which have enabled us to share our findings with a wider audience. We appreciate the efforts of the editorial team in reviewing and editing our work, and we are thankful for the opportunity to contribute to the field of research through this publication.

Funding Information

No particular grant from a public or private funding agency was obtained for this research.

Author's Contributions

Both authors have contributed equally to this study.

Ethics

This piece of writing is unique and includes unreleased content. All co-authors have read and approved the article, and the corresponding author attests that there are no ethical concerns.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., McGrew, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., *et al*. (2023). Gpt-4 Technical Report. *ArXiv:2303.08774*. https://doi.org/10.48550/arXiv.2303.08774

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Unified Pre-Training for Program Understanding and Generation. *ArXiv:2103.06333*. https://doi.org/10.48550/arXiv.2103.06333

Ajibode, A., Yunwei, D., & Hongji, Y. (2023). Software Issues Report for Bug Fixing Process: An Empirical Study of Machine-Learning Libraries. *Software Engineering*. https://doi.org/10.48550/arXiv.2312.06005

Alagarsamy, S., Tantithamthavorn, C., & Aleti, A. (2024). A3Test: Assertion-Augmented Automated Test Case Generation. *Information and Software Technology*, *176*, 107565. https://doi.org/10.1016/j.infsof.2024.107565

Allamanis, M., Brockschmidt, M., & Khademi, M. (2017). Learning to Represent Programs with Graphs. *ArXiv:1711.00740*. https://doi.org/10.48550/arXiv.1711.00740

Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., Hasan, M., Van Essen, B. C., Awwal, A. A. S., & Asari, V. K. (2019). A State-of-the-Art Survey on Deep Learning Theory and Architectures. *Electronics*, *8*(3), 292–369. https://doi.org/10.3390/electronics8030292

Arakelyan, S., Hakhverdyan, A., Allamanis, M., Garcia, L., Hauser, C., & Ren, X. (2022). NS3: Neuro-Symbolic Semantic Code Search. *Advances in Neural Information Processing Systems*, *35*, 10476–10491.

Bagherzadeh, M., Kahani, N., & Briand, L. (2022). Reinforcement Learning for Test Case Prioritization. *IEEE Transactions on Software Engineering*, *48*(8), 2836–2856. https://doi.org/10.1109/tse.2021.3070549

Bello, R.-W., & Tobi, S. J. (2024). Software Bugs: Detection, Analysis and Fixing. *SSRN Electronic Journal*, *189*, 1–22. https://doi.org/10.2139/ssrn.4662187

Bengio, Y. (2009). *Learning Deep Architectures for AI*. *2*. https://doi.org/10.1561/2200000006

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1798–1828. https://doi.org/10.1109/tpami.2013.50

Berndt, D. J., & Watkins, A. (2005). High Volume Software Testing using Genetic Algorithms. *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, 318–318. https://doi.org/10.1109/hicss.2005.296

Black, S., Gao, L., Wang, P., Leahy, C., & Biderman, S. (2021). Gpt-neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. *If You Use This Software, Please Cite It Using These Metadata*, *58*(2). https://doi.org/10.5281/zenodo.5297715

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., … Amodei, D. (2020). Language Models are Few-shot Learners. *Advances in Neural Information Processing System*, *33*, 1877–1901.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P. S., Yang, Q., & Xie, X. (2024). A Survey on Evaluation of Large Language Models. *ACM Transactions on Intelligent Systems and Technology*, *15*(3), 1–45. https://doi.org/10.1145/3641289

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, Harri, Burda, Y., Joseph, N., Brockman, G., Ray, Alex, Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. *ArXiv:2107.03374*. https://doi.org/10.48550/arXiv.2107.03374

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *ArXiv:1810.04805*. https://doi.org/10.48550/arXiv.1810.04805

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBert: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, Eric, Shi, F., Zhong, R., Yih, W., Zettlemoyer, Luke, & Lewis, M. (2022). Incoder: A Generative Model for Code Infilling and Synthesis. *ArXiv:2204.05999*. https://doi.org/10.48550/arXiv.2204.05999

Furtado, A. P., Meira, S., Santos, C., Novais, T., & Ferreira, M. (2016). FAST: Framework for Automating Software Testing. *He Eleventh International Conference on Software Engineering Advances*, 91.

GitHub. (2016). *GitHub pull request process*. https://help.github.com/articles/using-pull-requests

Gousios, G., Storey, M.-A., & Bacchelli, A. (2016). Work Practices and Challenges in Pull-Based Development. *Proceedings of the 38th International Conference on Software Engineering*, 285–296. https://doi.org/10.1145/2884781.2884826

Gousios, G., Zaidman, A., Storey, M.-A., & Deursen, A. van. (2015). Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 358–368. https://doi.org/10.1109/icse.2015.55

Grishman, R., & Sundheim, B. (1996). Message Understanding Conference-6. *Proceedings of the 16th Conference on Computational Linguistics*, 466–471. https://doi.org/10.3115/992628.992709

Gu, J., Chen, Z., & Monperrus, M. (2021). MultimodalMultimodal Representation for Neural Code Search. *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 483–494. https://doi.org/10.1109/icsme52107.2021.00049

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, Long, Duan, Nan, Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2020). Graphcodebert: Pre-Training Code Representations with Data Flow. *ArXiv:2009.08366*. https://doi.org/10.48550/arXiv.2009.08366

Hannan, S. A., Ahmed, S. J., Ahmed, Q. N., & Thakur, R. A. (2012). Data Mining and Natural Language Processing Methods for Extracting Opinions from Customer Reviews. *International Journal of Computational Intelligence and Information Security*, *3*(6), 52–58.

Hindle, A., Barr, E. T., Gabel, M., Su, Z., & Devanbu, P. (2016). On the Naturalness of Software. *Communications of the ACM*, *59*(5), 122–131. https://doi.org/10.1145/2902362

Jalil, S., Rafi, S., LaToza, T. D., Moran, K., & Lam, W. (2023). ChatGPT and Software Testing Education: Promises & Perils. *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 4130–4137. https://doi.org/10.1109/icstw58534.2023.00078

Jeanrenaud, A., & Romanazzi, P. (2024). Software Product Evaluation Metrics: A Methodological Approach. *WIT Transactions on Information and Communication Technologies*, *9*. https://doi.org/10.2495/SQM940052

Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020). Learning and Evaluating Contextual Embedding of Source Code. *Proceedings of the 37th International Conference on Machine Learning*, 5110–5121.

Kiran, A., Butt, W. H., Anwar, M. W., Azam, F., & Maqbool, B. (2019). A Comprehensive Investigation of Modern Test Suite Optimization Trends, Tools and Techniques. *IEEE Access*, *7*, 89093–89117. https://doi.org/10.1109/access.2019.2926384

Kollanus, S., & Koskinen, J. (2009). Survey of Software Inspection Research. *The Open Software Engineering Journal*, *3*(1), 15–34.

Kononenko, O., Baysal, O., & Godfrey, M. W. (2016). Code Review Quality. *Proceedings of the 38th International Conference on Software Engineering*, 1028–1038. https://doi.org/10.1145/2884781.2884840

Krasner, H. (2021). *The cost of poor software quality in the US: A 2020 report*. https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

Lee, S., & Lee, G. G. (2005). Heuristic Methods for Reducing Errors of Geographic Named Entities Learned by Bootstrapping. *Natural Language Processing – IJCNLP 2005*, *3651*, 669–658. https://doi.org/10.1007/11562214_58

Lertbanjongngam, S., Chinthanet, B., Ishio, T., Kula, R. G., Leelaprute, P., Manaskasemsak, B., Rungsawang, A., & Matsumoto, K. (2022). An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode. *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, 10–15. https://doi.org/10.1109/iwsc55060.2022.00010

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Qian, Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Vries, H. de. (2023). Starcoder: May the Source be With You! *ArXiv:2305.06161*. https://doi.org/10.48550/arXiv.2305.06161

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., … Vinyals, O. (2022). Competition-Level Code Generation with AlphaCode. *Science*, *378*(6624), 1092–1097. https://doi.org/10.1126/science.abq1158

Li, Z., Pan, M., Pei, Y., Zhang, T., Wang, L., & Li, X. (2024). Empirically Revisiting and Enhancing Automatic Classification of Bug and Non-Bug Issues. *Frontiers of Computer Science*, *18*(5), 185207. https://doi.org/10.1007/s11704-023-2771-z

Liu, X., Chen, H., & Xia, W. (2022). Overview of Named Entity Recognition. *Journal of Contemporary Educational Research*, *6*(5), 65–68. https://doi.org/10.26689/jcer.v6i5.3958

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., & Jiang, Daxin. (2023). Wizardcoder: Empowering code large language models with evol-instruct. *ArXiv:2306.08568*. https://doi.org/10.48550/arXiv.2306.08568

Luu, Q.-H., Liu, H., & Chen, T. Y. (2023). Can ChatGPT advance software testing intelligence? An experience report on metamorphic testing. *Software Engineering*. https://doi.org/10.48550/arXiv.2310.19204

Lyu, Z., Li, X., Xie, Z., & Li, M. (2025). Top Pass: Improve Code Generation by Pass@k-Maximized Code Ranking. *Frontiers of Computer Science*, *19*(8), 198341. https://doi.org/10.1007/s11704-024-40415-9

Mehmood, A., Ilyas, Q. M., Ahmad, M., & Shi, Z. (2024). Test Suite Optimization Using Machine Learning Techniques: A Comprehensive Study. *IEEE Access*, *12*, 168645–168671. https://doi.org/10.1109/access.2024.3490453

Nakao, H., & Eschbach, R. (2008). Strategic Usage of Test Case Generation by Combining Two Test Case Generation Approaches. *2008 Second International Conference on Secure System Integration and Reliability Improvement*, 213–214. https://doi.org/10.1109/ssiri.2008.17

OpenAI. (2024). *ChatGPT*. https://openai.com/blog/chatgpt

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., & Lowe, R. (2022). Training Language Models to Follow Instructions with Human Feedback. *Advances in Neural Information Processing Systems*, *35*, 27730–27744.

Ozkaya, I. (2023). Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks and Implications. *IEEE Software*, *40*(3), 4–8. https://doi.org/10.1109/ms.2023.3248401

Özpolat, Z., Yildirim, Ö., & Karabatak, M. (2023). Artificial Intelligence-Based Tools in Software Development Processes: Application of ChatGPT. *European Journal of Technic*, *13*(2), 229–240. https://doi.org/10.36222/ejt.1330631

Pacheco, C., & Ernst, M. D. (2007). Randoop. *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, 815–816. https://doi.org/10.1145/1297846.1297902

Pan, R., Bagherzadeh, M., Ghaleb, T. A., & Briand, L. (2022). Test Case Selection and Prioritization Using Machine Learning: a Systematic Literature Review. *Empirical Software Engineering*, *27*(2), 29. https://doi.org/10.1007/s10664-021-10066-6

Plein, L., Ouédraogo, W. C., Klein, J., & Bissyandé, T. F. (2024). Automatic Generation of Test Cases based on Bug Reports: A Feasibility Study with Large Language Models. *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 360–361. https://doi.org/10.1145/3639478.3643119

Prakash, B., B, S., S, S., & R, V. (2020). Optimization of Test Cases: A Meta-Heuristic Approach. *International Journal of Advanced Trends in Computer Science and Engineering*, *9*(4), 6569–6576. https://doi.org/10.30534/ijatcse/2020/346942020

Qi, L., Dou, W., Wang, W., Li, G., Yu, H., & Wan, S. (2018). Dynamic Mobile Crowdsourcing Selection for Electricity Load Forecasting. *IEEE Access*, *6*, 46926–46937. https://doi.org/10.1109/access.2018.2866641

Qi, L., Zhang, X., Dou, W., & Ni, Q. (2017). A Distributed Locality-Sensitive Hashing-Based Approach for Cloud Service Recommendation From Multi-Source Data. *IEEE Journal on Selected Areas in Communications*, *35*(11), 2616–2624. https://doi.org/10.1109/jsac.2017.2760458

Qu, X., Cohen, M. B., & Rothermel, G. (2008). Configuration-aware regression testing: an empirical study of sampling and prioritization. *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 75–86. https://doi.org/10.1145/1390630.1390641

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). *Improving Language Understanding by Generative Pre-Training*.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, *21*(140), 1–67.

Rigby, P. C., & Bird, C. (2013). Convergent Contemporary Software Peer Review Practices. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 202–212. https://doi.org/10.1145/2491411.2491444

Rigby, P. C., German, D. M., Cowen, L., & Storey, M.-A. (2014). Peer Review on Open-Source Software Projects. *ACM Transactions on Software Engineering and Methodology*, *23*(4), 1–33. https://doi.org/10.1145/2594458

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., & Synnaeve, G. (2023). Code Ilama: Open Foundation Models for Code. *ArXiv:2308.12950*. https://doi.org/10.48550/arXiv.2308.12950

Saetre, R. (2006). *GeneTUC: Natural Language Understanding in Medical Text*.

Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. *Neural Networks*, *61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Singhal, S., Jatana, N., Suri, B., Misra, S., & Fernandez-Sanz, L. (2021). Systematic Literature Review on Test Case Selection and Prioritization: A Tertiary Study. *Applied Sciences*, *11*(24), 12121–12168. https://doi.org/10.3390/app112412121

Sun, X., Peng, X., Zhang, K., Liu, Y., & Cai, Y. (2019). How Security Bugs are Fixed and What can be Improved: An Empirical Study with Mozilla. *Science China Information Sciences*, *62*(1), 19102. https://doi.org/10.1007/s11432-017-9459-5

Sun, X., Yang, H., Xia, X., & Li, B. (2017). Enhancing Developer Recommendation with Supplementary Information Via Mining Historical Commits. *Journal of Systems and Software*, *134*, 355–368. https://doi.org/10.1016/j.jss.2017.09.021

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, *105*(12), 2295–2329. https://doi.org/10.1109/jproc.2017.2761740

Tallam, S., & Gupta, N. (2006). A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. *ACM SIGSOFT Software Engineering Notes*, *31*(1), 35–42. https://doi.org/10.1145/1108768.1108802

Tao, C., Gao, J., & Wang, T. (2019). Testing and Quality Validation for AI Software–Perspectives, Issues and Practices. *IEEE Access*, *7*, 120164–120175. https://doi.org/10.1109/access.2019.2937107

Tuteja, M., & Dubey, G. (2012). A Research Study on importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Models. *International Journal of Soft Computing and Engineering (IJSCE)*, *2*(3), 251–257.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all You Need. *Advances in Neural Information Processing Systems*, *30*, 6000–6010.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software Testing with Large Language Models: Survey, Landscape and Vision. *IEEE Transactions on Software Engineering*, *50*(4), 911–936. https://doi.org/10.1109/tse.2024.3368208

Wang, W., Li, G., Ma, B., Xia, X., & Jin, Z. (2020). Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 261–271. https://doi.org/10.1109/saner48275.2020.9054857

Wei, Z., Xiaoxue, W., Xibing, Y., Shichao, C., Wenxin, L., & Jun, L. (2017, November). Test suite minimization with mutation testing-based many-objective evolutionary optimization. In 2017 International Conference on Software Analysis, Testing and Evolution (SATE) (pp. 30-36). IEEE.

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021a). CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

Wang, X., Wang, Y., Mi, F., Zhou, P., Wan, Y., Liu, X., Li, Li, Wu, Hao, Liu, J., & Jiang, Xin. (2021b). SynCobert: Syntax-Guided MultimodalMultimodal Contrastive Pre-Training for Code Representation. *ArXiv:2108.04556*. https://doi.org/10.48550/arXiv.2108.04556

White, M., Vendome, C., Linares-Vasquez, M., & Poshyvanyk, D. (2015). Toward Deep Learning Software Repositories. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 334–345. https://doi.org/10.1109/msr.2015.38

Xin, W., Zheng, Q., & Fengyan, H. (2007). UML Based Hybrid Model for Generation of Software Reliability Test Cases. *Journal-Xian Jiaotong University*, *41*(4), 421.

Yan, M., Xia, X., Zhang, X., Xu, L., & Yang, D. (2017). A Systematic Mapping Study of Quality Assessment Models for Software Products. *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, 63–61. https://doi.org/10.1109/sate.2017.16

Yin, Y., Chen, L., xu, yueshen, & Wan, J. (2018). Location-Aware Service Recommendation With Enhanced Probabilistic Matrix Factorization. *IEEE Access*, *6*, 62815–62825. https://doi.org/10.1109/access.2018.2877137

Yin, Y., Xu, W., Xu, Y., Li, H., & Yu, L. (2017). Collaborative QoS Prediction for Mobile Service with Data Filtering and SlopeOne Model. *Mobile Information Systems*, *2017*(1), 1–14. https://doi.org/10.1155/2017/7356213