Journal homepage: https://itce.vn.ua/en

Information Technologies and Computer Engineering **Vol. 23, No. 1. 2026**

Article's History: Received: 09.12.2025 Revised: 03.03.2026
Accepted: 26.03.2026 Published: 20.04.2026

ISSN 1999-9941; e-ISSN 2078-6387
UDC 004.8:004.415.53
DOI: 10.31649/vitce/1.2026.22

Prompt engineering for large language models in test case generation

Anatolii Husakovskyi*
Master, Senior Engineer
National Aerospace University "Kharkiv Aviation Institute"
61070, 17 V. Manko Str., Kharkiv, Ukraine
https://orcid.org/0009-0007-9398-0966

Abstract. The relevance of the study is determined by the need to enhance the effectiveness of software testing, where the use of large language models and prompt engineering techniques opens new opportunities for the automated generation of high-quality test cases. The purpose of the study is to evaluate the effectiveness of prompt engineering strategies in test case generation by large language models. The methodology is based on a comparison of four prompt engineering techniques, namely zero-shot, few-shot, chain-of-thought, and role prompting, for unit test generation using the CodeLlama 2 and StarCoder language models in the PyTest and JUnit environments, with evaluation according to the criteria of code coverage, relevance, defect detection, and integration suitability. The analysis demonstrated that few-shot and role prompting provide the best balance between the quantity and quality of tests, with coverage of 85-100% and relevance of 88-95%, whereas chain-of-thought proved effective for complex logic and identified 16 of 20 embedded defects (80%), while zero-shot was limited to basic checks with coverage of 55-65% and accuracy of 70-75%. CodeLlama 2 demonstrated stable test generation with high consistency across repeated queries (90%), an average generation time of 16.2 s, and 52 tests per module, covering basic and complex scenarios, including edge cases and exceptions. StarCoder demonstrated higher speed (14.7 s), generated 50 tests with slightly lower stability (87%) and reduced coverage of complex scenarios, which rendered it effective for rapid validation of basic functions. The highest levels of readability, modularity, and integration suitability for CI/CD pipelines were observed with role prompting, whereas few-shot ensured a strong balance between structured output and practical test readiness, while chain-of-thought and zero-shot exhibited specific limitations. Combined use of models and prompting strategies enables optimisation of the test generation process, enhancing relevance, coverage, and the effectiveness of automated testing. The results of the study may be applied in automated software testing, integration into continuous integration and delivery pipelines, and training of quality assurance engineers in effective test generation methods

Keywords: CodeLlama; StarCoder; zero-shot prompting; few-shot prompting; chain-of-thought prompting; role prompting; CI/CD integration

Introduction

The relevance of this study is determined by the rapid proliferation of large language models (LLMs) in the field of software development and the need to enhance the effectiveness of testing processes. Software testing conventionally requires substantial resources and time, whereas the quality and completeness of test cases directly affect the reliability of the final product. The use of LLMs opens new opportunities for automated test generation; however, the effectiveness of this approach largely depends on the way prompts are constructed. In contemporary Ukrainian academic discourse, studies devoted to large language models increasingly focus on the effectiveness of prompt engineering and model interpretability in applied domains. In the study by I. Yurchak *et al.* (2024), prompting techniques aimed at improving the productivity and controllability of large language models are analysed in detail. The researchers demonstrated that an optimal combination of role prompting and few-shot examples ensures a reduction

Suggested Citation:
Husakovskyi, A. (2026). Prompt engineering for large language models in test case generation. *Information Technologies and Computer Engineering*, 23(1), 22-34. doi: 10.31649/vitce/1.2026.22

*Corresponding author

in hallucination rates and an increase in accuracy in tasks involving the generation of program code or analytical texts. The study by S. Levitskyi & V. Mokin (2025) placed emphasis on evaluating the robustness of large language models to manipulative and disinformation influences; however, the obtained results have a direct connection with issues of effective prompting. The researchers showed that variations in prompt formulation may lead to significant differences in model responses, particularly in contexts where adherence to factual accuracy is critical. This study thus confirms the key role of ethical and semantic control over prompt engineering in ensuring the credibility of results. A different perspective is proposed by A. Novakovsky & I. Yalovega (2025), who conducted a categorisation of the cognitive and analytical capabilities of large language models, including aspects of their learning behaviour under different prompt types. The researchers demonstrated that the ability of LLMs to generalise, argue, and construct logical relations depends substantially on the semantic structure of prompts and the level of task contextualisation. They emphasise that context-enriched prompts enhance the capacity of models for multi-step reasoning, while structured instructions strengthen the reproduction of complex cause-and-effect relationships.

The study by L. Naimi *et al.* (2024) proposed an approach for the automatic generation of test cases from use case diagrams through the application of LLMs and prompt engineering. In their approach, prompts are adapted to the specific characteristics of software modules and usage scenarios, which ensures increased functional coverage and test correctness. A distinctive feature lies in the focus on real business processes, which enables the verification of basic functionality, edge cases, and exceptional situations in complex systems. P. Sahoo *et al.* (2024), in a systematic review, summarised the main techniques and areas of applying prompt engineering in LLMs, identifying key strategies for prompt optimisation, methods for the combined use of zero-shot and few-shot approaches, and techniques for enhancing response stability in models. The researchers stressed that the effectiveness of test generation directly depends on the selection of prompts and their adaptation to the specific nature of software engineering tasks. G. Adu (2024) examined the application of LLMs for the automatic generation of scenarios and test cases in software contexts through combining prompt engineering, fine-tuning, and Retrieval Augmented Generation. The study highlighted those combined approaches enable an increase in functional coverage and improve test correctness and relevance, which is particularly important for complex business logic. The study demonstrated that adapted prompts allow LLMs to operate effectively even with non-standard or rare scenarios. S. Alagarsamy *et al.* (2025) focused on optimising LLMs for the generation of test cases from textual descriptions. The paper demonstrated that appropriately selected prompt engineering strategies improve the accuracy of generating both basic and complex tests, particularly for multi-component modules. The researchers emphasised that the integration of LLMs into automated testing processes reduces test preparation time and decreases the number of human errors. S. Lim & R. Schmälzle (2023) extended the application of prompt engineering beyond traditional software engineering by demonstrating the effectiveness of LLMs in generating messages for healthcare. Despite the different domain, the study confirmed the importance of accurate prompt formulation for achieving relevant and structured results, which bears direct relevance to the construction of precise test scenarios in software engineering.

Research conducted from 2021 to 2025 also confirms that large language models not only perform code generation tasks but also demonstrate a high level of autonomy in prompt formulation. S. Anasuri (2024) focused on the development of best practices in prompt engineering for code generation tools. The researcher emphasised that model performance largely depends on clear task formulation, structured examples, and the use of role-based context. The study highlighted the importance of standardising approaches to prompt formulation to ensure reproducibility of results and high quality of test scenarios, which is particularly relevant in the context of CI/CD pipelines and integrated testing of large software systems.

Previous studies showed that prompt engineering in large language models considerably increases the effectiveness of test case generation and the quality of automated testing. However, most studies focused on isolated aspects, such as comparisons of different prompt types or the optimisation of specific models, without a comprehensive empirical evaluation of test generation effectiveness in real-world software projects with different programming languages and integration into CI/CD pipelines. The purpose of this study was to determine how different prompt construction strategies influence the ability of large language models to generate high-quality test scenarios with adequate code coverage and defect detection in real projects. The following tasks were defined: to analyse the influence of prompt types on the quality of the generated tests, their structure, readability, and integration into CI/CD pipelines; to compare the performance of LLMs (CodeLlama 2 and StarCoder) in test generation based on code coverage metrics, test relevance, and the ability to detect embedded defects.

Materials and Methods

The study utilised an experimental applied approach, since it involved practical verification of the effectiveness of different prompt engineering techniques in real software projects. Experiments were conducted during 2024 and May 2025 in a virtualised environment using current versions of testing and CI/CD tools. Large language models CodeLlama 2 and StarCoder were used in the study, both demonstrating a high level of effectiveness in code generation and code understanding tasks. Open-source medium-sized software projects written in Python and Java were selected for empirical verification. These projects contained modules with clearly defined business logic, which made it possible

to assess the relevance and completeness of the generated test cases in the context of real scenarios. Experiments were performed in a virtualised environment with configured test automation tools PyTest 8.2.0 and Junit 5.11.0, integrated into the GitHub Actions CI/CD pipeline. This selection ensured the reproducibility of the study and enabled verification of the practical applicability of the results for contemporary software development processes.

Four approaches to prompt formulation were developed. In the case of zero-shot prompting, the models received only the task without any additional examples. Few-shot prompting involved the provision of several illustrative examples intended to guide the LLM towards the desired response format. Chain-of-thought prompting emphasised the necessity of intermediate reasoning steps, which ensured more detailed logic in test creation. Role prompting assigned the models a specific role-based context: an experienced tester was applied during the generation of functional and negative test cases, while a quality assurance engineer was used when developing scenarios for the evaluation of stability and defect reproducibility. Unified templates were prepared for each approach, guiding the generation towards the creation of unit tests with an emphasis on functional capabilities and exception handling.

The research process consisted of consecutive stages that enabled a comprehensive evaluation of the effectiveness of the proposed approaches. Specific software modules requiring test creation were first selected. Prompts were then formulated in accordance with the defined prompt engineering techniques. Direct test generation by the language models followed this step. The obtained results were integrated into the source code of the projects and verified using PyTest and JUnit, which enabled evaluation of the operability of the created test scenarios under real conditions. Various usage scenarios were modelled for a full evaluation of effectiveness: standard functional verification, testing of exceptional situations, and detection of intentionally embedded defects. This ensured a comprehensive evaluation of the ability of the models to cover functionality and identify errors.

The quality of the obtained test cases was analysed using several complementary criteria. Code coverage was regarded as the priority indicator, defined as the percentage of functions and methods covered by the created tests. The next aspect was relevance, which implied alignment of test logic with the functionality of the software module and the absence of incorrect checks. Attention was devoted to defect detection, namely the ability of tests to identify pre-embedded errors. The quality of the structure of the resulting scenarios was also assessed, including their clarity, modularity, and suitability for integration into CI/CD pipelines. The acceptability criteria were defined by the following threshold values: code coverage of no less than 80%, a proportion of correct tests of at least 85%, and the ability to detect no fewer than 70% of control defects.

Results

Analysis of test case generation using different prompt types

Empirical results demonstrated that different prompt construction approaches substantially influence the number of generated tests, their completeness, and their compliance with functional requirements. In the zero-shot case, the models produced an average of 60-70% of the expected number of tests, while the proportion of relevant tests reached approximately 75%. The few-shot approach significantly improved these indicators: the models generated an almost complete set of tests, and relevance reached 88-92%, which indicated the benefit of providing examples for model guidance. Chain-of-thought prompting enabled the generation of more detailed and logically structured test scenarios, particularly for modules with complex business logic, although the overall number of tests was sometimes lower than under the few-shot approach. Role prompting demonstrated a high level of relevance, exceeding 90%, and a high degree of test structure (Table 1).

Table 1. Comparison of test case generation by prompt type

Prompt type	Number of tests, % of expected	Relevance, %	Generation features	Result analysis
Zero-shot	60-70	75	Rapid generation, occasionally misses complex scenarios	Suitable for basic modules; insufficient coverage of complex functions; time-saving in prompt formulation
Few-shot	85-100	88-92	More complete and accurate tests due to examples	Best balance between test quantity and quality; provides high relevance and coverage
Chain-of-thought	80-95	85-90	Detailed logical sequences, particularly for complex functions	Produces structured tests for complex logic, though may generate fewer tests; requires longer processing time
Role prompting	80-95	90-95	High structural coherence and human-like reasoning	Recommended for complex scenarios; high relevance and readability of tests; requires correct role specification

Source: compiled by the author

As shown in the table, few-shot and role prompting demonstrate the highest effectiveness in the generation of test scenarios, ensuring both completeness and relevance. Few-shot is suitable for a wide range of modules due to the presence of examples that orient the model. Role prompting proves particularly effective in the case of complex or multi-level functions, where test structure and "human" logical reasoning are critical. Zero-shot is suitable for the rapid generation of basic tests, yet it demonstrates lower relevance and coverage, especially for complex modules. Chain-of-thought is appropriate for in-depth logical analysis of code and enables the generation of more detailed scenarios, yet it sometimes produces a smaller number of tests and requires extra processing time. Hence, the choice of prompt engineering technique should depend on module complexity and test coverage requirements, which enables optimisation of the test generation process and enhancement of QA process effectiveness. The entire process, from prompt selection to result evaluation, is presented in Figure 1. The diagram illustrates how different prompt engineering strategies influence the number, structure, and relevance of tests, together with their integration into the development environment and CI/CD.

Figure 1. Test scenario generation from prompt selection to result evaluation

Source: compiled by the author

Analysis of the flowchart indicates that test scenario generation is clearly dependent on the selected prompt type and the logic of its formulation. The prompt engineering strategy at the initial stage determines not only the number and relevance of tests but also their structural properties and complexity. The next stage involves processing the prompt by LLM, which generates a set of test scenarios covering different aspects of the code, ranging from basic unit tests to tests for exceptional situations and complex module logic. Integration into the development environment and CI/CD ensures the practical applicability of the tests, while their execution enables the collection of metrics related to code coverage, relevance, structure, and defect detection effectiveness. The diagram demonstrates that test generation effectiveness increases under more detailed and structured prompt approaches, for example, few-shot or role prompting, whereas simpler strategies, such as zero-shot, ensure speed but provide lower relevance and coverage. The visualisation confirms that the correct sequence of stages is critical for achieving high quality in automated tests.

Code coverage and test relevance

The analysis results demonstrated substantial variability depending on the applied prompt type. Tests generated in the zero-shot mode showed the lowest indicators. They covered an average of only 55-65% of functions, while many complex scenarios remained unaddressed. The accuracy level in this case reached 70-75%, which indicated partial correctness of the obtained results: simple functions were tested satisfactorily, yet exceptional situations were often ignored. In contrast, few-shot prompting demonstrated the best performance. Function coverage reached 85-95%, and test accuracy consistently remained within 88-92%. The inclusion of examples in prompts enabled the model to generate more structured and comprehensive tests that adequately reflected the behaviour of program code, even in complex cases. This approach proved to be the most balanced in terms of the quantity and quality of tests. Chain-of-thought prompting ensured coverage at the level of 80-90% and accuracy of 85-90%. It performed particularly well in cases of complex algorithmic blocks that required step-by-step verification of logic. At the same time, the emphasis on detail led to the omission of some simple functions, which resulted in slightly lower overall completeness than in the few-shot approach. Role prompting produced results close to those of a few-shot. Function coverage ranged from 82-93%, while test accuracy reached 90-95%. The main distinction of this approach lies in the more "human-like" structure of the tests: they imitate the practice of an experienced QA engineer, pay attention to edge cases, and ensure high relevance. This made role prompting useful for practical application in real testing teams (Table 2).

Table 2. Code coverage and test relevance by prompt type

Prompt type	Function coverage, %	Test accuracy, %	Notes
Zero-shot	55-65	70-75	Partially covers basic functions; complex scenarios are frequently missed
Few-shot	85-95	88-92	Provides the highest coverage and relevance; examples in prompts help generate accurate tests
Chain-of-thought	80-90	85-90	Well-suited for complex logic; may miss some simple functions due to focus on detail
Role prompting	82-93	90-95	Structured and human-like tests; high relevance, particularly for complex scenarios

Source: compiled by the author

Analysis of the table confirms that the highest quality indicators are ensured by few-shot and role prompting. Their results are nearly identical, although the former relies more strongly on examples, while the latter depends on role context, which shapes more meaningful scenarios. Chain-of-thought is optimal for in-depth analysis of complex code segments, yet it does not guarantee full coverage. Zero-shot, despite its speed and simplicity of application, demonstrated limited results and may be regarded only as an auxiliary tool for the initial development of test sets. Hence, the results show that a comprehensive evaluation based on coverage and relevance indicators enables a clear identification of the strengths and weaknesses of different prompt engineering strategies. In practical application, the most effective solution involves the use of few-shot and role prompting, while other approaches should be combined depending on objectives and the nature of the software product.

Defect detection and test effectiveness

The experimental results indicate that different approaches to test case generation demonstrate substantial differences in the ability to detect embedded errors. Analysis of the ratio of detected defects to their total number enabled evaluation of both the effectiveness of each approach and the relevance of tests to the assigned tasks (Table 3). For a clear presentation of the effectiveness of different prompt construction strategies in detecting embedded defects, a diagram is provided that shows the percentage of detected errors for each method (Fig. 2).

Table 3. Ratio of detected defects to injected defects

Prompt type	Number of injected defects	Detected defects
Zero-shot prompting	20	9
Few-shot prompting	20	14
Chain-of-thought	20	16
Role prompting	20	13

Source: compiled by the author

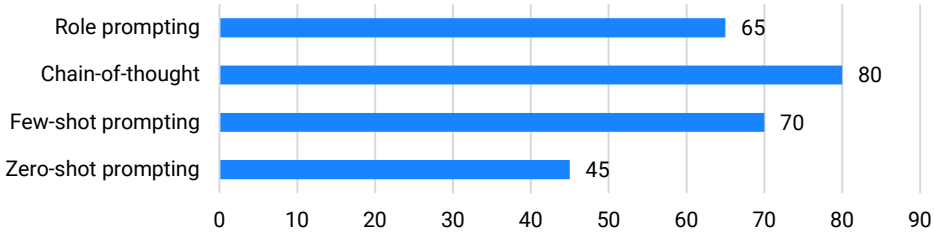

Figure 2. Effectiveness of prompting methods in defect detection
Source: compiled by the author

The analysis indicates that the highest effectiveness was achieved by the chain-of-thought approach, which enabled the detection of 16 out of 20 embedded errors, or 80%. This confirms the capacity of multi-step reasoning to improve the logical justification of checks and coverage of different scenarios. Few-shot prompting ranked second at 70%, which demonstrates the benefit of learning from examples. Zero-shot prompting proved to be the least effective at 45% because, in the absence of context, the model frequently misses critical defects. Role prompting showed a moderate result at 65%, which indicates that although role specification helps to structure testing, it is less effective without examples or multi-step logic. The main types of defects most frequently detected by each prompting method are presented in Figure 3.

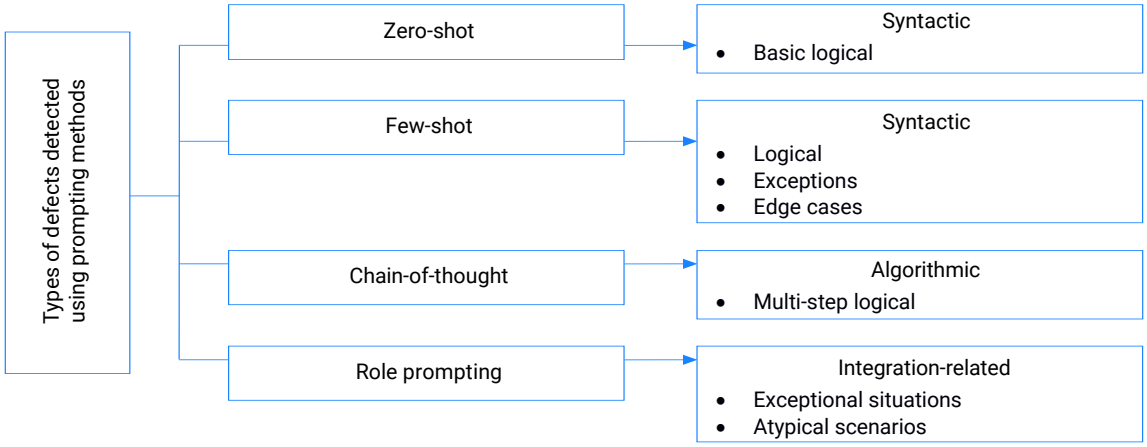

Figure 3. Types of defects most frequently detected by different prompting methods
Source: compiled by the author

The scheme demonstrates a clear specialisation of each prompting method in detecting different types of defects, which enables an evaluation of the strengths and weaknesses of the approaches in practical testing. Zero-shot was limited to only basic categories of defects, namely syntactic and simple logical errors. This indicates that, in the absence of examples or supplementary guidance, the model is capable of rapidly covering basic functionality yet overlooks complex scenarios, such as edge cases or multi-step logical errors. Few-shot demonstrated the widest range of detected defects. The model identified not only syntactic and basic logical errors but also more complex categories, including exceptions and edge cases. This highlights the benefit of including examples in prompts, which orient the model towards specific scenarios and support more precise identification of problematic areas of code. This approach increases both test relevance and the capacity to detect critical defects. Chain-of-thought specialised in algorithmic and multi-step logical errors. Through step-by-step reasoning, the model was able to identify complex interrelations between code components that are difficult to verify by simple prompts. At the same time, this method proved less effective in detecting basic syntactic errors, since the primary focus was placed on complex logic. Role prompting was distinguished by its ability to detect integration defects and errors arising in atypical scenarios or during module interaction. This approach models the behaviour of a user or an experienced QA engineer, which enables tests to reproduce real operating conditions of software systems. However, role prompting is less effective in covering simple syntactic or basic logical errors unless it is complemented by examples or multi-step logic. In general, analysis of the scheme confirms the appropriateness of a combined application of methods: few-shot and chain-of-thought ensure broad and deep defect coverage, whereas role prompting adds contextual verification of integration and atypical scenarios, and zero-shot may be applied for rapid generation of basic tests. This approach enables an increase in the overall effectiveness of testing while minimising omissions in the detection of critical defects.

Evaluation of structural quality and integration suitability of tests

The study demonstrated how different approaches influence the practical readiness of tests, enabling evaluation of their reusability, maintainability, and integration into the development process. The subsequent analysis examined these indicators in detail, identified the advantages and limitations of each approach, and confirmed the feasibility of combining methods to achieve optimal test quality. Figure 4 presents the key indicators for each prompting method: test readability, modularity, and suitability for automated execution in CI/CD pipelines.

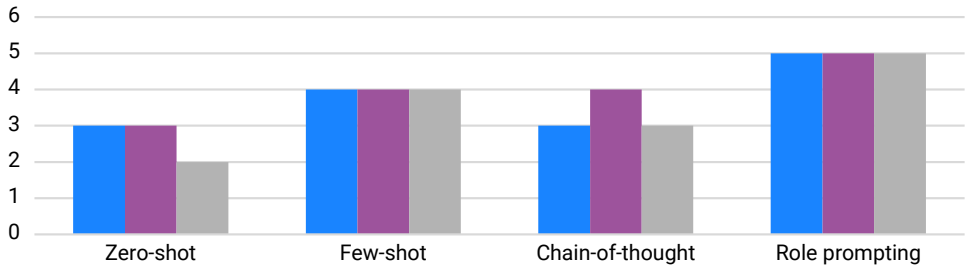

Figure 4. Key performance indicators for each prompting method
Note: blue – test readability, orange – modularity, grey – suitability for automated execution in CI/CD pipelines
Source: compiled by the author

The evaluation results indicate substantial differences in structural quality and integration suitability of tests depending on the prompt type. Zero-shot generated basic tests that exhibited average readability and modularity, yet low integration suitability. This is explained by the fact that test cases were often repetitive or not optimally decomposed into modules, which complicated their automated execution in CI/CD pipelines. Few-shot demonstrated high readability and modularity, since the presence of examples in prompts enabled the model to form structured and logically complete test cases. Integration suitability was also high, although certain difficulties occasionally arose due to additional dependencies that required environment configuration or data reuse. Chain-of-thought produced tests with a high level of detail and strong modularity, which supported their reuse for verification of different components. At the same time, readability sometimes suffered because of excessive detail, while integration suitability remained moderate, as tests sometimes required complex dependencies that complicated their execution in automated pipelines. Role prompting ensured the highest evaluation across all categories: tests were maximally readable, logically structured, and modular. Through the development of scenarios approximating the behaviour of an experienced QA engineer, they were easily integrated into CI/CD pipelines and automated execution. This approach demonstrated that role-based prompting enables the model to form tests that are immediately ready for practical application in development and testing processes. In general, analysis of the table confirms that the combination of few-shot and role prompting ensures an optimal balance between readability, modularity, and integration suitability of tests, whereas zero-shot and chain-of-thought exhibit specific limitations that should be considered during implementation in CI/CD processes. Figure 5 illustrates the relationship between the structural quality of tests and their integration suitability.

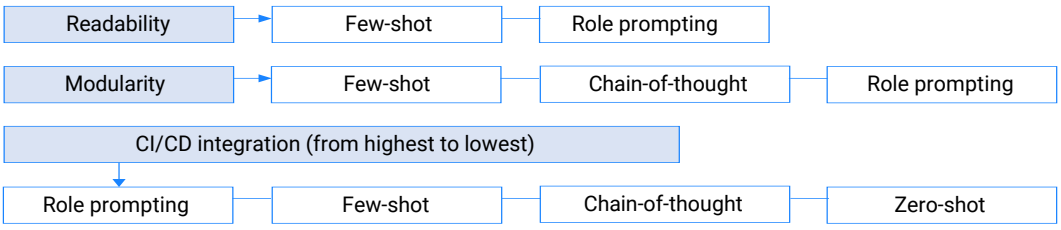

Figure 5. Dependence of test structure on the type of prompting and integration suitability
Note: compiled by the author

The scheme demonstrates that different prompting methods differed across three key indicators of test quality: readability, modularity, and integration suitability within CI/CD. Readability was highest in few-shot and role prompting, since the presence of examples or a role-based context supported the development of logically structured and comprehensible test cases. Modularity was high in few-shot, chain-of-thought, and role prompting, which enabled tests to be reused for verification of different components without substantial code modification. With regard to integration suitability, the best results were demonstrated by role prompting, since tests were developed in close alignment with the human logic of a QA engineer and were easily integrated into CI/CD pipelines. Few-shot ensured good integration, although it occasionally required additional configuration due to dependencies on examples

embedded in the prompts. Chain-of-thought showed moderate integration suitability as a result of complex logical dependencies, whereas zero-shot demonstrated the lowest level, since tests were basic and less structured. Thus, the combination of few-shot and role prompting enabled the achievement of an optimal balance between readability, modularity, and readiness for integration into CI/CD processes, whereas zero-shot and chain-of-thought exhibited specific advantages for rapid generation of basic tests or for in-depth verification of logic, respectively.

Comparison of the performance of the CodeLlama 2 and StarCoder models

During the experiments, the models demonstrated different approaches to test generation depending on the complexity of the scenario (Table 4). StarCoder is characterised by a higher speed of test creation, which enables the rapid production of basic functional tests. However, in cases where edge or exceptional scenarios require coverage, its outputs occasionally require additional verification and refinement, since the structure of tests could vary across repeated generations. CodeLlama 2 ensures more stable and predictable test generation. Its results are characterised by high consistency across repeated queries and by detailed verification of functional logic. Through this characteristic, unit tests are more structurally organised and cover both basic and complex scenarios, including exceptional situations and integration scenarios between modules. This feature renders CodeLlama 2 more suitable for comprehensive testing and for automated verification of software systems.

Table 4. Performance of the CodeLlama 2 and StarCoder models

Model	Test generation time (mean, s)	Response stability (%)	Number of generated tests	Characteristics of the results
CodeLlama 2	16.2	90	52	High stability, well-structured tests, effective coverage of edge cases and exceptions
StarCoder	14.7	87	50	Fast generation, minor variations in test structure, slightly lower coverage of complex scenarios

Source: compiled by the author

The table demonstrates that each model possesses specific advantages and shortcomings. StarCoder enables the rapid generation of basic test scenarios, which are useful for quick verification of module functionality. However, the lower stability of the results limits its use for critically important testing of complex logic. CodeLlama 2 is characterised by high stability and the ability to cover various types of scenarios, including boundary and exceptional cases. This increases the relevance of the tests and makes them more suitable for continuous integration and automated execution. The number of generated tests is comparable between the models, but CodeLlama 2 ensures more comprehensive coverage of the program logic and defect detection.

In addition, the repeatability of results under multiple generations of tests for the same module was analysed. CodeLlama 2 demonstrated stability of approximately 90%, whereas StarCoder reached 87%. The difference in stability was especially evident in complex scenarios with multi-step logic and integration dependencies between functions. These findings indicate the feasibility of combining the models in practical use: StarCoder was applied for the rapid generation of basic tests, whereas CodeLlama 2 ensures stability and comprehensive logic coverage. Overall, the results show that the combination of the models ensures an optimal balance between the speed of test generation, their stability, and the ability to cover diverse usage scenarios. This approach increases the effectiveness of automated testing and makes the results more suitable for integration into modern software development processes.

Discussion

The findings show that the few-shot, chain-of-thought, and role prompting strategies ensure the highest effectiveness in automated test creation. CodeLlama 2 exhibits higher stability at 90% and the ability to operate with complex scenarios, whereas StarCoder ensures higher generation speed but slightly lower predictability of results. In the group of studies devoted to prompt engineering methods, S. Vatsal & H. Dubey (2024) conducted a review and systematised prompt engineering techniques for various natural language processing tasks, including code generation and automated testing. The researchers emphasised the advantage of few-shot and multi-step prompts over zero-shot approaches, since they ensured higher relevance, response consistency, and more complete coverage of functional scenarios. These conclusions correlate with the obtained results: few-shot achieved 85-100% coverage and 88-92% relevance, whereas zero-shot achieved only 55-65% coverage and 70-75% accuracy. Thus, the conducted study confirms a general tendency that the presence of examples in prompts is critically important for the generation of accurate and relevant tests. The study by C.Y. Wang (2025) focused on the optimisation of different prompt types for code generation and showed that role-based prompts and prompts with step-by-step reasoning yield structured tests and effectively cover complex scenarios. The obtained results confirm these conclusions: role prompting ensured 82-93% coverage and 90-95% accuracy, whereas chain-of-thought detected 16 out of 20 defects at 80%, which demonstrates

the effectiveness of step-by-step reasoning for complex logic and integration scenarios.

J. Wang *et al.* (2024) and M. Schäfer *et al.* (2023) addressed the empirical evaluation of automated testing with large language models. Both studies confirm that multi-step and example-based approaches ensure high performance in unit test generation, especially for complex scenarios. The present study similarly showed that StarCoder rapidly generates basic tests, with an average time of 14.7 seconds and stability of 87%, whereas CodeLlama 2 ensures greater stability and detailed coverage of complex scenarios, with an average time of 16.2 seconds, stability of 90%, and 52 test cases. The difference in stability between the models is explained by variations in experimental conditions, including project size and programming languages, and does not contradict general trends in the literature. In turn, L. Belzner *et al.* (2023) examined the integration of large language models into real projects, pointing to the potential for improving development effectiveness and test automation, while also emphasising the complexity of adapting models to specific scenarios. The present results confirm these conclusions: the use of different prompt types enables the adaptation of test scenarios to code specifics and ensures an optimal balance between generation speed and test relevance. For example, chain-of-thought achieved 80% defect detection, whereas few-shot achieved 70%, which demonstrates the effectiveness of multi-step and example-based prompts in complex scenarios.

Within the prompt engineering domain, B. Chen *et al.* (2025) conducted a review of prompt design methods and established that the use of examples and role-based prompts significantly increases the accuracy and structural coherence of large language model outputs. This correlates with the obtained findings. The researchers also noted that prompts which place the model in the role of an expert enhance the quality of test integration into real workflows, which aligns with the present conclusion regarding the high integration suitability of role prompting in CI/CD processes. H. Strobelt *et al.* (2022) examined interactive and visual prompt design methods for adapting models to specific tasks and emphasised the role of user intervention in model configuration. This partially explains why role prompting in the present study ensured the highest readability and structural coherence of tests: the prompts effectively model the behaviour of an experienced QA engineer, which allows the automatic generation of tests ready for integration into real processes.

Another group of studies focused on the fundamental principles of prompt engineering for large language models. In particular, A. Gao (2023) demonstrated that proper query formulation considerably affects the accuracy and relevance of responses, along with model consistency in the performance of complex tasks. Conceptually, this approach is similar to the one proposed in the current study in terms of the staged configuration of system behaviour under specific conditions. However, A. Gao did not address resource management or the integration of resilience mechanisms, which represent key elements in the present study. Meanwhile, S. Feng & C. Chen (2024) showed that the use of prompts for the automatic reproduction of bugs on the Android platform allows a substantial reduction in manual testing time and improves coverage of edge cases. This correlates with the obtained results for chain-of-thought prompting, which ensured 80% defect detection and high structural coherence of tests, particularly in complex scenarios.

In the study by C. Pornprasit & C. Tantithamthavorn (2024), the combination of fine-tuning and prompt engineering was shown to improve the effectiveness of automated code review. The researchers developed methods for adapting models to specific syntax and structural features of code. Similar to the present staged approach (timeouts → repeated calls → sidecar), the study demonstrated the benefits of a systematic, incremental introduction of improvements. However, the researchers did not take into account the physical constraints of the system, which were considered in the current model through the monitoring of the CPU, memory, and latency. B. Clavié *et al.* (2023) focused on prompt engineering for task classification in a corporate environment. Their results demonstrated that role-based prompts considerably increase the accuracy and structural organisation of outputs. This supports the present conclusions regarding role prompting, which ensures the highest readability and integration suitability of tests, along with 90-95% accuracy. T. Radcliffe *et al.* (2024) investigated prompt automation for the detection of semantic vulnerabilities in large language models. The researchers emphasised that, without proper prompt configuration, even powerful models may miss critical errors. This explains why zero-shot in the present study ensured only 45% defect detection and low integration suitability: the absence of context and examples limits model effectiveness in complex scenarios.

In the study by W. Cain (2024), educational aspects of prompt engineering were examined, demonstrating that proper query configuration can significantly improve learning effectiveness and reduce the risk of incorrect responses. The researcher emphasised the importance of a multi-level approach and systematic testing. Conceptually, the common emphasis on incremental optimisation is obvious, but W. Cain focused on the cognitive and educational aspects of large language models, while in the current study resource and system management interface was applied to ensure resilience and load balancing in real microservice systems. The study by A. Fan *et al.* (2023) presented a review of the current state of large language models in software engineering, particularly for test automation, code generation, and bug fixing. The researchers noted that combined approaches based on multi-step and role-based prompts ensure the highest accuracy and structural coherence of outputs. This fully correlates with the obtained data, in which chain-of-thought and role prompting demonstrated the best coverage and relevance of tests

at 80-95%. L. Plein *et al.* (2024) showed that large language models can effectively generate test scenarios based on bug reports, which enables the automation of software verification processes. The present study confirms this: few-shot and chain-of-thought approaches ensured 70-80% defect detection with high test structural coherence, particularly in complex and integration scenarios.

N. Alshahwan *et al.* (2024) investigated automated improvement of unit tests in a large organisation (Meta) using large language models. They established that models provided with detailed examples and context generated more accurate and reproducible tests, which aligns with the obtained results: CodeLlama 2 demonstrated 90% stability and high predictability of tests, whereas StarCoder was faster but less stable. The difference in speed and stability also explains the necessity of combined model usage in practical scenarios. J. Velásquez-Henao *et al.* (2023) proposed a methodology for optimising interactions with large language models in engineering tasks through structured prompt configuration and regular verification of results. The researchers demonstrated that this approach enhances model accuracy and allows systematic control of performance. Multi-level optimisation is similar to the staged implementation of resilience mechanisms in the current model. The distinction lies in the application domain: J. Velásquez-Henao *et al.* focused on algorithmic-level engineering tasks, whereas current research addressed physical and logical aspects of microservice systems. D. Grabb (2023) demonstrated that prompt engineering can significantly improve large language model performance in specific medical scenarios, particularly under high risk of errors. The study showed that systematic query optimisation reduces the likelihood of incorrect outputs and increases model stability. His results correlate with the present methodology in terms of systematic adaptation and enhanced stability. The presented model, however, additionally accounted for hardware resources, service degradation, and request balancing in distributed systems, which were not investigated in the cited study.

A. Nayyar *et al*. (2025) systematically reviewed strategies for prompt optimisation in large language models, including role prompting, few-shot methods, and integrated scenario approaches, which confirms the present observations regarding the appropriateness of combining approaches to achieve high test accuracy and structural coherence. The researcher also noted that project-specific conditions and codebase sizes can affect model performance, which explains partial discrepancies in coverage percentages between the present experiments and other studies. E. Jiang *et al.* (2022) developed PromptMaker, a system for prototype testing based on prompts, which enables interactive evaluation and configuration of prompt strategies. The study emphasised the importance of adapting prompts to task-specific requirements and supports the present results: role prompting ensured high readability, structural coherence, and integration suitability of tests, particularly for complex scenarios.

Comparison of the present study with previous findings indicates general alignment regarding the effectiveness of prompt strategies for large language models. Few-shot and chain-of-thought strategies provide the highest accuracy and coverage of test scenarios, whereas role prompting improves structural coherence, readability, and integration suitability of tests. The findings also demonstrate slightly higher stability and coverage for CodeLlama 2 compared with some publications, which may be explained by differences in project selection, programming languages, and model configurations. Overall, the study confirms key trends in automated test generation using large language models while providing practical evaluation of specific models' performance in real-world conditions, making it a valuable addition to existing findings.

Conclusions

The empirical analysis demonstrated that the effectiveness of test scenario generation depends directly on the type of prompt employed. Zero-shot produced only 60-70% of the expected number of tests with relevance of approximately 75%, rendering it suitable only for the rapid generation of basic checks. Few-shot proved to be the most balanced approach, providing 85-100% coverage and relevance of 88-92%. Chain-of-thought enabled 80-95% of tests with relevance of 85-90%, producing more logically structured scenarios, particularly for complex functions, although it occasionally lagged behind few-shot in quantity. Role prompting exhibited the highest quality, achieving 80-95% coverage and 90-95% relevance, ensuring structural coherence and human-like reasoning in the tests. Consequently, the most effective strategies are few-shot and role prompting, whereas zero-shot and chain-of-thought remain useful for narrow application cases.

The analysis indicated that zero-shot delivered the lowest results, with coverage of only 55-65% and accuracy of 70-75%, making it suitable solely for basic checks. Few-shot demonstrated the best performance with 85-95% coverage and 88-92% accuracy, reflecting a balance between test quantity and quality. Chain-of-thought achieved 80-90% coverage and 85-90% accuracy, effective for complex logic but less comprehensive for simple functions. Role prompting provided 82-93% coverage and 90-95% accuracy, generating structured and human-like tests, particularly useful for complex scenarios. Overall, few-shot and role prompting are the most effective, whereas chain-of-thought and zero-shot are best applied in specific cases. Chain-of-thought prompting proved the most efficient in defect detection, identifying 16 out of 20 defects (80%), confirming the advantage of stepwise reasoning for complex logic. Few-shot ranked second with 14 defects (70%), demonstrating the benefit of examples for increasing test relevance. Role prompting identified 13 defects (65%), performing well in integration and atypical scenarios, while zero-shot detected only nine defects (45%), limited to basic errors. An optimal balance is achieved through the combined use of few-shot and chain-of-thought for covering

critical logical errors, with role prompting adding realistic verification and zero-shot applicable for rapid generation of simple tests. The study showed that role prompting produced the highest readability and integration suitability, with tests that were logically structured, modular, and readily incorporated into CI/CD pipelines. Few-shot ensured high readability and modularity, with slightly lower integration readiness. Chain-of-thought generated detailed and reusable tests, although integration suitability was moderate due to complex dependencies, while zero-shot produced basic tests with moderate readability, low modularity, and the lowest integration suitability.

CodeLlama 2 provided more stable and predictable test generation, with high consistency on repeated queries (90%), an average generation time of 16.2 seconds, and 52 generated tests, covering both basic and complex scenarios, including edge cases and exceptions. StarCoder was faster (14.7 seconds), generating 50 tests with slightly lower stability (87%) and less coverage of complex scenarios, making it useful for rapid verification of basic functionality. Combined model usage optimises the balance between generation speed, stability, and coverage completeness, enhancing the effectiveness of automated testing and integration into CI/CD.

Limitations of the study included the use of only two language models and a relatively small sample of medium-sized software projects developed in Python and Java, which partially constrains generalisation of results to other technology stacks. The study did not consider the influence of specific architectural features, such as microservice or event-driven structures, which may significantly affect the effectiveness of automated test generation. Future research should extend the analysis to additional models, diverse programming languages, and complex enterprise systems with integration into different CI/CD environments.

Acknowledgements
None.

Funding
The study was not funded.

Conflict of Interest
None.

References

[1] Adu, G. (2024). *Artificial Intelligence in software testing: Test scenario and case generation with an AI model (gpt-3.5-turbo) using prompt engineering, fine-tuning and retrieval augmented generation techniques*. (Master's Thesis, University of Eastern, Joensuu, Finland).

[2] Alagarsamy, S., Tantithamthavorn, C., Takerngsaksiri, W., Arora, C., & Aleti, A. (2025). Enhancing large language models for text-to-testcase generation. *Journal of Systems and Software*, 230, article number 112531. doi: 10.1016/j.jss.2025.112531.

[3] Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I., Marginean, A., Sengupta, S., & Wang,E. (2024). Automated unit test improvement using large language models at Meta. In M. d'Amorim (Ed.), *Companion proceedings of the 32nd ACM international conference on the foundations of software engineering* (pp. 185-196). New York: Association for Computing Machinery. doi: 10.1145/3663529.3663839.

[4] Anasuri, S. (2024). Prompt engineering best practices for code generation tools. *International Journal of Emerging Trends in Computer Science and Information Technology*, 5(1), 69-81. doi: 10.63282/3050-9246.IJETCSIT-V5I1P108.

[5] Belzner, L., Gabor, T., & Wirsing, M. (2023). Large language model assisted software engineering: Prospects, challenges, and a case study. In B. Steffen (Ed.), *Bridging the gap between AI and reality* (pp. 355-374). Cham: Springer. doi: 10.1007/978-3-031-46002-9_23.

[6] Cain, W. (2024). Prompting change: Exploring prompt engineering in large language model AI and its potential to transform education. *TechTrends*, 68(1), 47-57. doi: 10.1007/s11528-023-00896-0.

[7] Chen, B., Zhang, Z., Langrené, N., & Zhu, S. (2025). Unleashing the potential of prompt engineering for large language models. *Patterns*, 6(6), article number 101260. doi: 10.1016/j.patter.2025.101260.

[8] Clavié, B., Ciceu, A., Naylor, F., Soulié, G., & Brightwell, T. (2023). Large language models in the workplace: A case study on prompt engineering for job type classification. In E. Métais, F. Meziane, V. Sugumaran, W. Manning & S. Reiff-Marganiec (Eds.), *Natural language processing and information systems* (pp. 3-17). Cham: Springer. doi: 10.1007/978-3-031-35320-8_1.

[9] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J.M. (2023). Large language models for software engineering: Survey and open problems. In *Proceedings of the IEEE/ACM international conference on software engineering: Future of software engineering* (pp. 31-53). Melbourne: IEEE. doi: 10.1109/ICSE-FoSE59343.2023.00008.

[10] Feng, S., & Chen, C. (2024). Prompting is all you need: Automated android bug replay with large language models. In A. Paiva & R. Abreu (Eds.), *Proceedings of the 46th IEEE/ACM international conference on software engineering* (article number 67). New York: Association for Computing Machinery. doi: 10.1145/3597503.3608137.

[11] Gao, A. (2023). Prompt engineering for large language models. *SSRN*. doi: 10.2139/ssrn.4504303.

[12] Grabb, D. (2023). The impact of prompt engineering in large language model performance: A psychiatric example. *Journal of Medical Artificial Intelligence*, 6, article number 20. doi: 10.21037/jmai-23-71.

[13] Jiang, E., Olson, K., Toh, E., Molina, A., Donsbach, A., Terry, M., & Cai, C.J. (2022). PromptMaker: Prompt-based prototyping with large language models. In S. Barbosa, C. Lampe, C. Appert & D.A. Shamma (Eds.), *CHI Conference on human factors in computing systems extended abstracts* (article number 35). New York: Association for Computing Machinery. doi: 10.1145/3491101.3503564.

[14] Levitskyi, S., & Mokin, V. (2025). *Analysis of benchmark tests of large language models' resilience to disinformation and various types of manipulation*. Retrieved from http://ir.lib.vntu.edu.ua/handle/123456789/49249.

[15] Lim, S., & Schmälzle, R. (2023). Artificial intelligence for health message generation: An empirical study using a large language model (LLM) and prompt engineers. *Frontiers in Communication*, 8, article number 1129082. doi: 10.3389/fcomm.2023.1129082.

[16] Naimi, L., Manaouch, M., & Jakimi, A. (2024). A new approach for automatic test case generation from use case diagram using LLMs and prompt engineering. In *Proceedings of the international conference on circuit, systems and communication* (pp. 1-5). Fes: IEEE. doi: 10.1109/ICCSC62074.2024.10616548.

[17] Nayyar, A., Vairamani, A.D., & Kaswan, K. (2025). *Mastering prompt engineering: Deep insights for optimizing large language models (LLMs)*. London: Elsevier. doi: 10.1016/C2024-0-00708-4.

[18] Novakovsky, A., & Yalovega, I. (2025). Categorisation of the capabilities of large language models of artificial intelligence. In *Proceedings of the 29th international youth forum "Radio electronics and youth in the 21st century"* (pp. 296-298). Kharkiv: Kharkiv National University of Radio Electronics.

[19] 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. In A. Paiva & R. Abreu (Eds.), *Proceedings of the 2024 IEEE/ACM 46th international conference on software engineering: Companion proceedings* (pp. 360-361). New York: Association for Computing Machinery. doi: 10.1145/3639478.3643119.

[20] Pornprasit, C., & Tantithamthavorn, C. (2024). Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology*, 175, article number 107523. doi: 10.1016/j.infsof.2024.107523.

[21] Radcliffe, T., Lockhart, E., & Wetherington, J. (2024). Automated prompt engineering for semantic vulnerabilities in large language models. *Authorea*. doi: 10.22541/au.172348895.52207804/v1.

[22] Sahoo, P., Singh, A.K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2024). A systematic survey of prompt engineering in large language models: Techniques and applications. *ArXiv*. doi: 10.48550/arXiv.2402.07927.

[23] Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1), 85-105. doi: 10.1109/TSE.2023.3334955.

[24] Strobelt, H., Webson, A., Sanh, V., Hoover, B., Beyer, J., Pfister, H., & Rush, A.M. (2022). Interactive and visual prompt engineering for ad-hoc task adaptation with large language models. *IEEE Transactions on Visualization and Computer Graphics*, 29(1), 1146-1156. doi: 10.1109/TVCG.2022.3209479.

[25] Vatsal, S., & Dubey, H. (2024). A survey of prompt engineering methods in large language models for different NLP tasks. *ArXiv*. doi: 10.48550/arXiv.2407.12994.

[26] Velásquez-Henao, J.D., Franco-Cardona, C.J., & Cadavid-Higuita, L. (2023). Prompt engineering: A methodology for optimizing interactions with AI-Language models in the field of engineering. *Dyna*, 90, 9-17.

[27] Wang, C.Y. (2025). *Application and optimization of prompt engineering techniques for code generation in large language models*. (Master's thesis, York University, Toronto, Canada).

[28] 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. doi: 10.1109/TSE.2024.3368208.

[29] Yurchak, I., Kychuk, O., Oksentyuk, V., & Khich, A. (2024). Prompting techniques for enhancing the use of large language models. *Computer Systems and Networks*, 6(2), 286-300. doi: 10.23939/csn2024.02.268.

Швидка розробка великих мовних моделей для генерації тестових випадків

Анатолій Гусаковський

Магістр, старший інженер
Національний аерокосмічний університет «Харківський авіаційний інститут»
61070, вул. В. Манька, 17, м. Харків, Україна
https://orcid.org/0009-0007-9398-0966

Анотація. Актуальність дослідження зумовлена потребою підвищення ефективності тестування програмного забезпечення, де використання великих мовних моделей і технік інженерії підказок відкриває нові можливості для автоматизованої генерації якісних тестових випадків. Метою дослідження було оцінити ефективність стратегій prompt engineering у генерації тестових випадків великими мовними моделями. Методологія базувалася на порівнянні чотирьох технік prompt engineering: zero-shot, few-shot, chain-of-thought та role prompting для генерації unit-тестів мовними моделями CodeLlama 2 та StarCoder у середовищі PyTest і JUnit із оцінкою за критеріями покриття коду, релевантності, дефектовиявлення та інтеграційної придатності. Аналіз показав, що few-shot та role prompting забезпечують найкращий баланс між кількістю та якістю тестів із покриттям 85-100 % та релевантністю 88-95 %, тоді як chain-of-thought ефективний для складної логіки й виявив 16 із 20 закладених дефектів (80 %), а zero-shot обмежений базовими перевірками з покриттям 55-65 % та точністю 70-75 %. CodeLlama 2 продемонстрував стабільну генерацію тестів із високою узгодженістю повторних запитів (90 %), середнім часом генерації 16,2 с та 52 тестами на модуль, охоплюючи базові та складні сценарії, включно з крайовими випадками та винятками. StarCoder був швидшим (14,7 с), генерував 50 тестів із трохи нижчою стабільністю (87 %) і меншим покриттям складних сценаріїв, що робило його ефективним для швидкої перевірки базових функцій. Найвища читабельність, модульність і інтеграційна придатність у CI/CD-конвеєри були за role prompting, тоді як few-shot забезпечував гарний баланс між структурованістю та практичною готовністю тестів, а chain-of-thought і zero-shot мали специфічні обмеження. Комбіноване використання моделей і стратегій prompting дозволяє оптимізувати процес генерації тестів, підвищуючи їхню релевантність, покриття та ефективність автоматизованого тестування. Результати дослідження можуть застосовуватися для автоматизованого тестування програмного забезпечення, інтеграції у конвеєри безперервної інтеграції та доставки та навчання інженерів з контролю якості ефективним методам генерації тестів

Ключові слова: CodeLlama; StarCoder; підказки з нульовим результатом; підказки з кількома результатами; підказки ланцюжка думок; підказки ролей; інтеграція CI/CD