UDC 004.43
DOI: 10.15587/2706-5448.2025.326899

Andrii Tkachuk

DETERMINING THE CAPABILITIES OF GENERATIVE ARTIFICIAL INTELLIGENCE TOOLS TO INCREASE THE EFFICIENCY OF REFACTORING PROCESS

The object of research is a source code refactoring facilitated and proctored by generative artificial intelligence tools. The paper is aimed at assessing their impact on refactoring quality while determining their practical applicability for improving software maintainability and efficiency.

The problem addressed in this research is the limitations of traditional rule-based refactoring tools, which require predefined rules and are often language-specific. Generative AI, with its advanced pattern recognition and adaptive learning capabilities, offers an alternative approach. However, its effectiveness in handling various refactoring tasks and its reliability remain undisclosed.

The research involved multiple experiments, where four AI tools – ChatGPT, Copilot, Gemini, and Claude – were tested on various refactoring tasks, including code smell detection, efficiency improvements, decoupling, and large-scale refactoring.

The results showed that Claude achieved the highest success rate (78.8%), followed by ChatGPT (76.6%), Copilot (72.8%), and Gemini (61.8%). While all tools demonstrated at least a basic understanding of refactoring principles, their effectiveness varied significantly depending on the complexity of the task. These results can be attributed to differences in model training, specialization, and underlying architectures. Models optimized for programming tasks performed better in structured code analysis, whereas more general-purpose models lacked depth in specific programming-related tasks.

The practical implications of this research highlight that while Generative AI tools can significantly aid in refactoring, human oversight remains essential. AI-assisted refactoring can enhance developer productivity, streamline software maintenance, and reduce technical debt, making it a valuable addition to modern software development workflows.

Keywords: *AI-driven refactoring, code quality improvement, automated code smell detection, generative AI tools, software optimization.*

Received: 16.01.2025
Received in revised form: 20.03.2025
Accepted: 07.04.2025
Published: 17.04.2025

How to cite
Tkachuk, A. (2025). Determining the capabilities of generative artificial intelligence tools to increase the efficiency of refactoring process. Technology Audit and Production Reserves, 3 (2 (83)), 6–11. https://doi.org/10.15587/2706-5448.2025.326899

1. Introduction

Refactoring is the process of restructuring existing code without changing its external behavior to improve its readability, maintainability, and efficiency [1]. It aims to reduce technical debt and enhance software design while preserving functionality [2].

Most refactoring systems rely only on generalized rules, which should have been written beforehand and are tightly coupled with a specific programming language [3].

Automated refactoring can be organized in multiple ways. Advanced refactoring and source code analysis systems can leverage ontology-backed knowledge bases to enhance their functionality [3]. These systems can store relationships between code entities, strictly define their types, query the knowledge base, and perform complex data reasoning. This capability enables them to modify or generate source code with the assurance that it will compile correctly, unlike AI-based approaches that rely on unspecified or fuzzy production rules [4, 5].

Recently, LLMs with their extensive capabilities got popular among software engineers. They use ChatGPT, Copilot, Gemini, and others to increase their productivity.

Large Language Models (LLMs) are transforming software engineering by automating tasks such as code generation, autocompletion, and bug detection. They assist in writing and reviewing code, generating documentation, and improving maintainability. LLMs also enable natural language interaction with codebases, facilitating code search and comprehension. Additionally, they enhance software testing by generating test cases and detecting edge cases, ultimately streamlining development workflows, and increasing productivity.

There are several ways to use AI in programming. Generative AI focuses on creating new content (e. g., text, images, and code) based on learned patterns, using models like GPT and diffusion networks [6]. Agentic AI refers to AI systems capable of autonomous decision-making, planning, and goal-oriented behavior, often employing reinforcement learning and symbolic reasoning [7]. Physical AI integrates artificial intelligence with robotics, enabling embodied systems to interact with the physical world, often utilizing sensors, actuators, and deep learning.

Generative AI has begun to play a significant role in code refactoring by automating the process of restructuring existing code to improve its readability, maintainability, and performance without altering its external behavior. Recent studies have demonstrated that models like GPT-4 can assist in refactoring tasks, leading to enhancements in code

quality [8]. Additionally, tools such as Tabnine leverage generative AI to provide code completion and refactoring suggestions, thereby streamlining the development process. However, it is important to note that while these AI-driven tools offer valuable assistance, human oversight remains essential to ensure the accuracy and functionality of the refactored code [9].

As a result, software development lifecycle may be transformed to increase productivity and quality and it may utilize GenAI tools in the spheres like code writing, refactoring, documentation, and prototyping [10].

The aim of research is to reveal the capabilities of generative AI tools in code refactoring and assess their impact on refactoring quality, while also determining their practical applicability for improving software maintainability and efficiency.

2. Materials and Methods

The object of research is a source code refactoring facilitated and proctored by generative artificial intelligence tools. To verify what tasks GenAI tools can conduct and how reliable the work is, a series of experiments were conducted.

The research methodology utilizes an experimental design to compare the performance of four generative AI tools (ChatGPT, Copilot, Gemini, and Claude) in refactoring source code. By testing these tools on a diverse set of tasks, such as code smell detection and fixing inefficiencies, the study ensures a comprehensive evaluation of their capabilities in addressing various refactoring challenges. The controlled environment, with consistent code samples, allows for a valid comparison while minimizing external influences. The inclusion of multiple tools provides a holistic view of AI-assisted refactoring, highlighting strengths and weaknesses across different contexts. The evaluation focuses on real-world outcomes like reducing technical debt and improving code readability, making the findings highly relevant to software development. The iterative feedback process mirrors real-world interactions with AI tools, enhancing the study's practical applicability. A combination of quantitative and qualitative analysis offers a deeper understanding of each tool's performance, and the reproducible methodology ensures that the study can be replicated, contributing to the broader field of AI-assisted software development.

At the *first round*, a sample code was provided to the tools. They were asked to determine whether the code snippet contains "code smells" and antipatterns which may be corrected. The example of a snippet is provided in Fig. 1.

```
func parameterList(param1: Int,
                   param2: Int,
                   param3: Int,
                   param4: Int,
                   param5: Int) -> Int {
    return param1 + param2 + param3 + param4 + param5
}
```

Fig. 1. Code snippet for first experiment

That code contains a function with 5 parameters which, according to best practices, should be moved and encapsulated in a struct.

ChatGPT accepted .swift file format easily, analyzed it quickly and provided definite answer – code contains a "code smell" named "Long Parameter List". It also provided two possible options of correcting the problem – by extracting parameters to a separate struct or by changing them to an array. Proposed code could be compiled and run.

For the Copilot and Gemini, the experiment started with an error – they cannot read the file format for Swift programming language.

Copilot was able to read a code from .txt file and find two potential problems – "Long Parameter List" and "Naming Convention" for parameters name. It also provided one possible fix – extraction of parameter to a struct. Taking into consideration the fact that Copilot cannot read code file formats and did not provide alternative solutions, the final score is lowered.

Gemini took the longest time to provide the answer. Though, it still managed to find the "Long Parameter List" "code smell" in the provided code snippet. In comparison with other tools, the response was not broad, containing just the name of the antipattern and a hint how to fix it (without any code references). To justify the ability to conduct simple code refactoring, an additional ask has been given to Gemini to conduct a refactoring. As a result, it provided 3 different ways which just 2 options with some modifications.

When the same task was provided to Claude, it was ruminating on it for a while (it accepted Swift file format), though it didn't mention that the code contains code smells at a first glance. Later, it produced information that there is may be potential "Long Parameter List" code smell, but it requires additional context to make assertive statements about that. When Claude was asked to conduct a refactoring, the code example which was generated, contained detailed explanations of what was done, how the code quality increased and an example of usage was provided.

The solution which was proposed by all four tools (to some extent) is depicted in Fig. 2.

It was observed that all three tools are capable of identifying simple antipatterns in the code and fixing it. To verify their capabilities deeper, it is needed to submit a bigger source code for analysis.

In the *second example* let's concentrate on abilities to identify explicit errors which can be predicted by compiler (interpreter) and such concepts like inefficient code and algorithms.

Example of the inefficient code is depicted in Fig. 3. The code contains sorting algorithm with $O(n^2)$ time complexity; search is conducted in the dictionary not by using hash abilities; loops does not finish execution effectively.

For the above-mentioned code, ChatGPT noticed two issues (incorrect syntax and property immutability), used more efficient sorting algorithm, fixed the problem with dictionary iteration and array lookup. After the generated code snippet, ChatGPT provided a short summary of what was done and justification of changes by applying comparison of Big O notations for the data access in dictionary and array sorting.

```
struct Parameters {
    let param1: Int
    let param2: Int
    let param3: Int
    let param4: Int
    let param5: Int
}

func parameterList(params: Parameters) -> Int {
    return params.param1 + params.param2 + params.param3 + params.param4 + params.param5
}
```

Fig. 2. Result of simple refactoring

ISSN-L 2664-9969; E-ISSN 2706-5448

```swift
func findValue(_ value: Int,
               in dictionary: [Int: Int],
               and array [Int]) -> (Int?, Int?) {
    var firstResult, secondResult: Int?

    // Let's sort the array first
    for i in 0..<(array.count - 1) {
        for j in 1..<array.count {
            if array[i] > array[j] {
                let temp = array[i]
                array[i] = array[j]
                array[j] = temp
            }
        }
    }

    // Let's find value in the dictionary
    for (key, _) in dictionary {
        if key == value {
            firstResult = key
        }
    }

    // Let's find value in the array
    for item in array {
        if item == value {
            secondResult = item
        }
    }

    return (firstResult, secondResult)
}
```

Fig. 3. Source code for the second example

The same code example was provided to Copilot system and the same task to refactor the code was given. Copilot generated its output almost instantly. The output is almost the same as the one provided by ChatGPT. Though, even the syntax issues were fixed, Copilot didn't mention that along with justifying the selected functions as substitution for the previous code. Also, no Big O notation was provided.

The next round was done using Gemini. It provided unique result in the term that it proposed not to use sorting at all as for the task in the function it is completely unnecessary. Gemini also didn't mention syntax errors in the initial source code example, though it explained why the provided code is not efficient, suggested changing variable names, and made suggestions on how the function return may be improved.

Claude found the biggest number of issues and done all the corrections which were also proposed by other systems except the variables names corrections. It justified the redundancy of sorting, using Big O notation, simplified lookup in dictionary and array.

The example of the expected code (which almost equals to the production of all systems) is depicted in Fig. 4.

In the *third example*, it is possible to concentrate on the problems such memory leak and tight coupling. In the code in the Fig. 5, there are two classes which depend on each other, have strong reference cycle between them, there is an interface which is common and may be extracted.

For the source code, ChatGPT identified strong reference cycle between classes, interface similarity and code duplications, that's why it proposed to create separate base class and a protocol. It also used abstraction to mark a reference to the other class.

Copilot coped with the task as well. It also generated a protocol to create a generic interface which may be used by both classes. Copilot also resolved strong reference cycle by applying weak references. Copilot also went ahead and applied additional refactoring actions like changed the prints for easier debugging.

Gemini did all the same things. Additionally, Gemini concentrated on usage of existential type instead of generic protocol usage (which means it keeps up with the latest language changes), improved debugging prints as well.

In this round, Claude did all the things that were done by other tools too. Additionally, Claude proposed to change names of functions to better address their purpose and to describe relations between dependent classes.

The code which was expected to be generated by tools is displayed in Fig. 6.

In the *fourth example*, a file of a Java server is sent for analysis to Generative AI systems. It is expected that the system will detect all the bottlenecks and provide recommendations with examples of how the code should be changed and fixed. A small extract of the server code is shown in Fig. 7.

When ChatGPT processed code, it identified the following drawbacks: usage of not thread-safe Singleton, excessive amount of instance variables, long constructor with repetitive code, bad exception handling with prints only, specific inefficiency related to the technology used in the code (SWRL). When ChatGPT processed code, it identified several drawbacks. They included usage of not thread-safe Singleton, excessive amount of instance variables, long constructor with repetitive code, bad exception handling with prints only, specific inefficiency related to the technology used in the code (SWRL). Despite code being in Java file, ChatGPT did not have troubles with reading it. It changed Singleton to be thread safe by using double-checked locking and Bill Pugh Singleton Pattern, encapsulated most of the properties in separate class, extracted common logic to reduce repetitive code, improved logging, optimized functions.

Copilot was asked to read the same file, though it does not have the ability to read code files. It was required to copy the code to regular text file to proceed. Though, Copilot still identified break of separation of concerns, redundant code and calls, bad error handling, lack of abstraction. It changed the Singleton to be thread-safe, encapsulated and extracted logic for properties handling, improved logging and changed IRI management.

```swift
func findValue(_ value: Int,
               in dictionary: [Int: Int],
               and array: [Int]) -> (Int?, Int?) {

    // Find value in dictionary (using Swift's native dictionary lookup)
    let firstResult: Int? = dictionary[value] != nil ? value : nil

    // Find value in array (using Swift's native contains method)
    let secondResult: Int? = array.contains(value) ? value : nil

    return (firstResult, secondResult)
}
```

Fig. 4. Code with improved efficiency

```swift
class TestClassA {
    var otherClass: TestClassB?

    private var name: String

    init() {
        name = "TestA"
    }

    func sayName() {
        print(name)
    }

    func sayOtherName() {
        otherClass?.sayName()
    }

    func printDate() {
        print(Date())
    }
}
```

a

```swift
class TestClassB {
    var otherClass: TestClassA?

    private var name: String

    init() {
        name = "TestB"
    }

    func sayName() {
        print(name)
    }

    func sayOtherName() {
        otherClass?.sayName()
    }

    func printDate() {
        print(Date())
    }
}
```

b

```swift
let first = TestClassA()
let second = TestClassB()

first.otherClass = second
second.otherClass = first
```

c

Fig. 5. Coupled classes code: *a* – first class; *b* – second class; *c* – classes usage

```swift
protocol Nameable {
    var name: String { get }
    func sayName()
    func sayOtherName()
    func printDate()
}

class BaseClass: Nameable {
    let name: String
    weak var otherClass: Nameable?

    init(name: String) {
        self.name = name
    }

    func sayName() {
        print(name)
    }

    func sayOtherName() {
        otherClass?.sayName()
    }

    func printDate() {
        print(Date())
    }
}
```

a

```swift
class TestClassA: BaseClass {
    init() {
        super.init(name: "TestA")
    }
}

class TestClassB: BaseClass {
    init() {
        super.init(name: "TestB")
    }
}

// Usage
let first = TestClassA()
let second = TestClassB()

first.otherClass = second
second.otherClass = first
```

b

Fig. 6. Decoupled and decomposed classes: *a* – base class, and a protocol; *b* – classes implementation and usage

```java
private Set<OWLAxiom> getAxiomsBetweenIndividuals(OWLNamedIndividual first, OWLNamedIndividual second) {
    Set<OWLAxiom> relevantAxioms = new HashSet<>();
    for (OWLAxiom axiom : ontology.getAxioms()) {
        if (axiom instanceof OWLObjectPropertyAssertionAxiom) {
            OWLObjectPropertyAssertionAxiom objectPropertyAssertion = (OWLObjectPropertyAssertionAxiom)axiom;
            OWLIndividual subject = objectPropertyAssertion.getSubject();
            OWLIndividual object = objectPropertyAssertion.getObject();
            if ((subject.equals(first) && object.equals(second)) ||
                (subject.equals(second) && object.equals(first))) {
                relevantAxioms.add(axiom);
            }
        }
    }
    return relevantAxioms;
}
```

Fig. 7. Example of the server-side code

ISSN-L 2664-9969; E-ISSN 2706-5448

Gemini did not provide clear justification of what was found and how "code smells" could be substituted with clear code. Also, generation of the refactored code snippet stopped almost at the end of the file, so the generated code is not completed. It provided two links at the end which led to the descriptions of issues with specific technology used in the code on GitHub. Second attempt with the same ask led to the same result. It is possible to consider this task to be not fully completed. It tried to extract duplicated code to separate functions, provide encapsulation for common objects though.

Claude took the longest time to generate the result. Claude also paused execution in the middle due to the reach maximum of words in one message, though it prompted to continue execution. It improved code structure, enhanced readability, removed "code smells", encapsulated relative code and behavior, improved error handling and performance.

It is worth mentioning that most of the fixes are the same for all the systems, though each of the tools applied its own set of "code smell" removing refactoring actions which serve the result of clean code.

3. Results and Discussion

The results obtained during the research were added to Table 1.

At each step of the experiment, the set of traits which the refactored code should have been identified and used to evaluate the success of each AI-tool.

For each of the conducted experiments, the maximum number of corrections was identified for all the AI systems collectively (as all of them provided unique useful results), and then the result produced by each system was normalized towards maximum result. That is why in some experiments none of the tools received maximum mark.

As a result, for each of the experiments, AI system received a mark from 0% to 100%.

For the first task, the task was to identify the presence of code smell, fix the naming convention, provide analysis and justification, and perform prompted refactoring. Copilot completed all the requirements, ChatGPT did 3 out of 4, Gemini and Claude were able to complete only the half.

At the second stage, ability to do the refactoring, provide several examples, comment the actions, and provide justifications for what was done were evaluated. None of the systems were able to complete all the tasks. ChatGPT, Gemini and Claude completed 3 out of 4, Copilot covered only the half.

During the third stage, ability to identify syntax issues, spot an inefficient algorithm, remove redundant actions, switch lookup method, and provide justification were taken into consideration during evaluation. Claude was super-efficient and completed all the checkmarks. ChatGPT finished 4 out of 5, while Gemini and Copilot were able to mark 3 out of 5 completed.

On the fourth stage, ability to fix strong reference cycle, extract and implement a protocol, extract and implement a base class, provide usage examples, conduct additional refactoring actions, and provide justification were evaluated. Copilot and Claude managed to achieve good result in 5 out of 6 checks, while ChatGPT and Gemini completed only 4 out of 6.

On the fifth stage, ability to identify and fix Singleton issues, find additional code smells, handle exceptions correctly, spot inefficient SWRL query (specific domain knowledge), provide working solution, mention referenced on the Internet, and complete the task fully were taken into consideration during evaluation. ChatGPT and Claude completed 6 out of 7 tasks, Copilot – 5, and Gemini did only 4.

The received result is the following: Claude gets the most credit (78.8%). ChatGPT received lower score (76.6%), though it did pretty the same job as Claude. Copilot is on the third place (72.8%). Gemini is on the fourth place (61.8%). It is worth noting that all the tools scored 50% and higher meaning that they at least understood the task, tried to solve it to the best of their model knowledge.

It is noticeable that Claude is specifically designed to work with the code as all the code snippets which were generated by the system were opened in a separate window; generated code was annotated; and usage example was provided.

By comparing the time that was consumed to solve different problems proposed in the experiments, Gemini and Claude used the biggest amount of time.

It is arguable that to achieve the same result with Copilot as with other systems, advanced and detailed prompts are required.

It is worth mentioning that at the time when the experiment may be replayed with the same input data, the achieved results may be different. It may be explained by the nature of Generative AI and LLM production, as well as the evolution of models and their training data.

As it was aimed in the research, deep understanding of Generative AI tools capabilities in refactoring may have practical application. Usage of all the mentioned tools may impact development workflows. The same outcome was achieved by the authors in [10]. They mention that by automation of repetitive tasks and fast prototyping, increase of the up-to-speed development productivity could be achieved. Integration of AI assistants may change the structure of a software development lifecycle, alter the balance of time developers spend on research of usual things and creating non-standard solutions. This is confirmed by the seen in the experiments ability to perform small, structured tasks with almost 100% accuracy. Additionally, increase in software developers' productivity is expected, which could allow to achieve the same result in less time and less investment. This statement becomes true due to the demonstrated ability of the AI-backed systems to help evaluate, restructure, and improve source code even if the success rate is not 100%. Automated refactoring assists enough to save time of developers on routines, so they can spend more efforts on adjusting the whole structure of the code.

Usage of AI assistants is always should be treated with caution. Authors in [9] mention that the output of GenAI is not always correct. As AI models have tendency to learn on the data they receive from users, sensitive information processing in the systems which are not properly isolated can cause harm to business. Additionally, the nature of content generation with artificial intelligence imposes the limitation of incorrect result. Authors in [10] also mention difficulties which are tied with the AI systems, such as ethical and security issues, technical imperfections. Users should always pay attention to what a system has generated before putting the result in use. Such notices were presented by all the systems which were used during the experiment. Also, outputs were not 100% correct or referencing the asked in the prompt task which was seen in the last experiment. Authors in [8] after the series of their experiments made the same conclusion about correctness of the production of AI-backed tools and the necessity of further refinement. This confirms the results which were received by other fellow researches and the ones achieved in the current research.

Table 1

Results of experiments

Experiment	ChatGPT (4o)	Copilot	Gemini 2.0 Flash	Claude 3.7 Sonnet
	Success %	Success %	Success %	Success %
Find "bad" code	75	100	50	50
Fix simple "code smell"	75	50	75	75
Fix inefficient code	80	60	60	100
Fix coupled repetitive code	67	83	67	83
Fix large code	86	71	57	86
Average	76.6	72.8	61.8	78.8

Though, the martial law in Ukraine does not have any direct impact on the study and its results, the discussed topic may be useful for software developers in Ukraine. It may help them to stay productive and effective while maintaining the code quality to stay on the top and be competitive. This is highly relevant in conditions of current software development market fluctuations.

Achieved in the research result may be further refined and improved by comparing other tools' traits for integration with IDE, ability to execute code in isolated environment to reduce risks of sensitive information leaks, backward compatibility.

4. Conclusions

The conducted research has demonstrated the ability of Generative AI systems to detect and resolve various code quality issues, such as code smells, inefficient algorithms, tight coupling, and poor architectural decisions. Four AI models – ChatGPT, Copilot, Gemini, and Claude – were tested on different refactoring tasks, with their performance evaluated based on their success rates in identifying and fixing issues.

The results indicate that Claude performed the best overall, achieving the highest success rate (78.8%) due to its ability to identify and correct multiple issues while providing well-structured code explanations. ChatGPT followed closely with a 76.6% success rate, demonstrating strong capabilities in detecting inefficiencies and applying suitable refactoring actions, though sometimes lacking depth in justifications. Copilot achieved a 72.8% success rate, excelling in syntactical corrections but requiring detailed prompts to reach its full potential. Gemini, with a 61.8% success rate, showed limited ability in identifying issues and generating complete refactored code, though it introduced unique optimizations not found in other models.

The qualitative traits observed during the experiments suggest that Claude is specifically designed for handling and explaining code modifications, making it a valuable tool for developers who require detailed reasoning. ChatGPT provides a balanced approach between speed and accuracy, making it useful for quick refactoring tasks. Copilot integrates well within development environments but struggles with complex refactoring unless prompted explicitly. Gemini, despite its lower performance, exhibited some innovative approaches to code optimization.

The achieved results have practical applications in software development workflows. Generative AI tools can be effectively utilized for code analysis, refactoring, and optimization, reducing the manual effort required from developers. However, human oversight remains crucial, as AI-generated refactoring actions need to be validated for correctness and maintainability. Future improvements in AI-assisted refactoring may focus on deeper contextual understanding, integration with formal verification methods, and adaptation to specific software engineering principles. Ultimately, the inclusion of Generative AI in refactoring processes has the potential to enhance code quality, maintainability, and developer productivity significantly.

Conflict of interest

The author declares that he has no conflict of interest in relation to this research, including financial, personal, authorship or any other, that could affect the research and its results presented in this article.

Financing

The research was performed without financial support.

Data availability

The manuscript has no associated data.

Use of artificial intelligence

The author used artificial intelligence technologies within permissible limits to provide their own verified data, as described in the research methodology section.

References

1. Fowler, M. (2019). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.
2. Mens, T., Tourwé, T., Demeyer, S. (2019). *Software Evolution.* Springer.
3. Tkachuk, A., Bulakh, B. (2023). Describing the Knowledge About the Source Code Using an Ontology. *Infocommunication and computer systems, 1 (5),* 123–133. https://doi.org/10.36994/2788-5518-2023-01-05-14
4. Tkachuk, A. V. (2024). Automated code refactoring using a knowledge base and logical rules. *Scientific Bulletin of UNFU, 34 (2),* 87–93. https://doi.org/10.36930/40340211
5. Põld, J., Robal, T., Kalja, A. (2013). On Proving the Concept of an Ontology Aided Software Refactoring Tool. *Frontiers in Artificial Intelligence and Applications, 249,* 84–94. https://doi.org/10.3233/978-1-61499-161-8-84
6. Bommasani, R., Hudson, D., Adeli, E., Altman, R., Arora, S., Arx, S. et al. (2021). *On the opportunities and risks of foundation models.* arXiv. https://doi.org/10.48550/arXiv.2108.07258
7. Russell, S., Norvig, P. (2020). *Artificial Intelligence: A Modern Approach.* Pearson.
8. Poldrack, R., Lu, T., Begus, G. (2023). *AI-assisted coding: Experiments with GPT-4.* arXiv. https://doi.org/10.48550/arXiv.2304.13187
9. Dhruv, A., Dubey, A. (2024). *Leveraging Large Language Models for Code Translation and Software Development in Scientific Computing.* https://doi.org/10.48550/arXiv.2410.24119
10. Sajja, A., Thakur, D., Mehra, A. (2024). Integrating Generative AI into the Software Development Lifecycle: Impacts on Code Quality and Maintenance. *International Journal of Science and Research Archive, 13 (1),* 1952–1960. https://doi.org/10.30574/ijsra.2024.13.1.1837

Andrii Tkachuk, *PhD, Assistant, Department of System Design, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-9127-6381, e-mail: andrewtkachuk@yahoo.com*