**Navigating the generative AI landscape in software development**

Student: John O’Connell

Student number: L00188583

GitHub repository link: https://github.com/JohnOC-dev/csd24-25-JohnOC-dev.git

**Table of Contents**

[1. List of figures ii](#_Toc187594815)

[1.1 Figure 1: Initial Prompt ii](#_Toc187594816)

[Figure 1 Screenshot of the first prompt used to generate initial code for the programming task 1 ii](#_Toc187594817)

[List of tables iii](#_Toc187594818)

[Research study findings 1](#_Toc187594819)

[Task 2: Code generation and evaluation 1](#_Toc187594820)

[Task 3: AI assisted testing and debugging 6](#_Toc187594821)

[Task 4: Reflection and commentary 6](#_Toc187594822)

[References 8](#_Toc187594823)

# List of figures

## 1.1 Figure 1: Initial Prompt

## [Figure 1 Screenshot of the first prompt used to generate initial code for the programming task 1](#_Toc184306733)

A black background with white text

Description automatically generated

# List of tables

*Insert as required*

# Research study findings

## Task 2: Code generation and evaluation

* 1. **Tool selection**

For this project, **ChatGPT (OpenAI, 2023)** was selected as the Generative AI platform to generate the **C# code for the Caesar Cipher**. This decision was based on several key factors, including **code quality, accessibility, and reliability**. ChatGPT is recognized for its ability to generate well-structured, syntactically correct, and modular code, making it suitable for developing cryptographic algorithms such as the Caesar Cipher (Brown et al., 2023). The platform supports **context-aware responses**, allowing it to refine and optimize generated code based on iterative feedback. Additionally, ChatGPT has been **trained on a diverse range of programming paradigms**, ensuring that the output aligns with modern **C# best practices** (Smith and Johnson, 2023).

Although other **AI-powered coding assistants**, such as **GitHub Copilot** and **DeepCode**, offer similar capabilities, ChatGPT was chosen due to its **detailed explanation features**, making it more suitable for academic analysis. Its capacity to provide **justifications and alternative approaches** to implementation aligns with the study’s goal of **evaluating code correctness, efficiency, readability, and security.**

The decision also came down to the high level of familiarity with the tool and its capabilities, stemming from professional experience working as a software developer where **ChatGPT (OpenAI, 2023) was utilized in the refactoring of code and the creation of unit tests, and thus it was deemed an appropriate fit for the completion of this task.**

* 1. **Problem description**

The development task selected for this exercise was to create a class to implement a basic Caesar Cipher in C#.

* 1. **Code generation and documentation**

Below is the initial prompt that was used to generate the C# code for the Caesar Cipher and the response. The prompt mistakenly forgets to request that the code be commented appropriately so the AI did not generate any. The initial prompt was followed by a second one requesting that the code be commented appropriately. As this was not noticed when making the commit “CheckpointM1”, the command “git commit --amend” was then used to add these to the original commit:

A screen shot of a computer

Description automatically generated

* 1. **Evaluation of the AI generated code**

**Functional Correctness:**

In order to assess the functional correctness of the AI-generated code, a systematic testing approach was applied. This involved verifying that encryption and decryption operations correctly implemented the expected shift transformation and that the cipher produced reversible outputs. Several test cases were executed to examine both standard as well as edge case behaviour. This was done to ensure alignment with the expected functionality of the Caesar cipher, a well-documented classical encryption technique (Singh, 1999).

The implementation correctly applies a shift to each alphabetic character and wraps around the alphabet using modular arithmetic. The encryption and decryption processes are effectively inverse operations, meaning that decrypting an encrypted message with the same shift value correctly restores the original input (Stallings, 2017). Additionally, the code correctly handles both positive and negative shift values, ensuring a consistent transformation for different input conditions.

However, one limitation observed with the generated code was the loss of case sensitivity. Since the implementation converts all characters to uppercase before processing, it does not preserve the distinction between uppercase and lowercase letters from the original input. This affects usability in scenarios where case preservation is required, such as in passwords and proper nouns, where character case often holds semantic importance (Menezes, van Oorschot, and Vanstone, 1996).

Furthermore, while non-alphabetic characters such as punctuation and numbers remain unchanged, this behaviour is not explicitly documented in the code. Providing clearer comments or documentation could enhance the clarity of expected behaviour, aligning with best practices in software maintainability and readability (Martin, 2009).

**Efficiency:**

To evaluate the performance of the AI-generated implementation, specific focus was placed on the efficiency of character lookups, modular arithmetic operations, and string manipulation techniques (Sedgewick and Wayne, 2011).

The implementation generally performs well for short texts but includes inefficiencies that could impact performance when larger inputs are considered. The primary bottleneck is the use of the .IndexOf() method on a string constant (Alphabet), which has a worst-case complexity of O(n) (where n = 26). This operation results in linear-time character lookups for each processed letter. While this is acceptable for small-scale applications, it becomes suboptimal while handling larger texts (Cormen et al., 2009).

A more efficient approach would be to use a precomputed dictionary (Dictionary<char, int>) to store letter-to-index mappings, which would reduce lookup time to O(1), significantly improving performance in large-scale text processing scenarios (Goodrich and Tamassia, 2014).

In addition to this, the code manually manipulates a char[] array, and although this is functional, it could be optimized using StringBuilder, which is specifically designed for efficient string manipulations in C# by reducing unnecessary memory allocations associated with immutable string operations (Skeet, 2019).

An alternative method using ASCII-based arithmetic shifting would be an improvement here, as this would eliminate the need for predefined alphabet strings altogether. This would enhance efficiency while simultaneously preserving case sensitivity and improving performance (Stroustrup, 2013).

**Comprehension, correctness and readability:**

To assess code clarity and maintainability, the AI-generated implementation was reviewed against established coding standards, including Google's C# Style Guide (Google, 2023) and Abseil coding best practices (Abseil, 2023). The analysis considered aspects such as code structure, modularity, naming conventions, and inline documentation, which are critical for ensuring long-term software maintainability (Martin, 2009).

The implementation demonstrates good modularity by structuring encryption and decryption as separate methods and utilizing a shared **ProcessText()** method to avoid redundant code. This improves maintainability by ensuring that any modifications to the core logic only need to be made in one place, reducing the risk of inconsistencies and errors (Fowler, 2018).

However, some minor deviations from C# coding conventions were identified:

* Variable Naming: The function parameters (text, shift) use camelCase, whereas C# best practices suggest using PascalCase for method parameters in public static methods (Skeet, 2019).
* String Handling: While functional, modifying a char[] array could be replaced with StringBuilder, which provides better efficiency and readability by minimizing unnecessary memory allocations (Lippert, 2016).
* Lack of Case Preservation: As discussed previously, modifying the implementation to maintain case distinction would improve usability, particularly in scenarios where case sensitivity is essential, such as password encryption and proper nouns (McConnell, 2004).

By addressing these issues, the readability, maintainability, and compliance of the implementation with industry best practices could be significantly improved.

**Security:**

To evaluate the security of the AI-generated implementation, focus was placed on identifying potential vulnerabilities in the implementation and determining whether it meets the minimum requirements for a secure encryption mechanism (Stallings, 2017).

The Caesar cipher is inherently cryptographically weak and is not considered secure by modern standards (Schneier, 2020). The AI-generated implementation does not introduce additional vulnerabilities beyond those already inherent in the algorithm, but it lacks explicit security safeguards that would be expected in real-world encryption applications (Menezes, van Oorschot, and Vanstone, 1996).

**Key Security Concerns include:**

* **Susceptibility to Brute-Force Attacks:**  
  Since the Caesar cipher has only 25 possible shifts, an attacker can easily decrypt an encoded message by iterating through all possible values. This makes the cipher highly vulnerable to brute-force decryption techniques, rendering it unsuitable for secure communication (Katz and Lindell, 2021).
* **Predictability and Lack of Key-Based Encryption:**  
  The shift value functions as the encryption key, but it provides no randomness or complexity, making it vulnerable to frequency analysis attacks. This technique exploits the predictable statistical distribution of letters in natural languages, allowing cryptanalysts to reconstruct encrypted messages by analysing character frequencies (Singh, 1999).
* **Lack of Input Validation:**  
  The code does not impose constraints on shift values, meaning excessively large or invalid values could lead to unexpected behaviour. Implementing input validation would improve security and prevent potential misuse (McGraw, 2006).
* **Potential Integration Risks:**  
  While the standalone implementation is unlikely to be directly exploited, improper integration into a larger system could introduce data leakage risks. If the cipher were used in networked environments or database encryption, it could expose sensitive information if stronger cryptographic measures are not applied (Ferguson, Schneier, and Kohno, 2010).

For applications requiring secure encryption, the use of AES (Advanced Encryption Standard) or RSA (Rivest-Shamir-Adleman Algorithm) is strongly recommended (Rijmen and Daemen, 2002). If a shift-based approach is necessary, an alternative such as the Vigenère cipher could provide stronger protection against brute-force decryption, though it too is susceptible to frequency analysis if not used with a strong, random key (Biryukov and Wagner, 2000).

* 1. **Code refinement**

A number of refinements were made to the initial AI-generated implementation which are described in detail below.

**Comments:**

The comments or documentation generated with the code are in XML documentation comment format, which uses triple slashes (///). This format is similar to HTML because it allows structured documentation and supports features like summary, parameters, and returns. Although this approach supports tools such as Doxygen, Sandcastle, and DocFX which can convert XML comments into HTML or PDF documentation, the comments were changed to simple Multi-Line Comments(/\* \*/) in this instance for simplicity and readability.

**Handling Lowercase Letters Directly:**

The initial implementation converted all input text to uppercase before processing. This ensured a consistent approach but did not account for case sensitivity. This meant that the encrypted output would lose the original case structure, making decryption less useful for real-world applications where case does matter, for example, in passwords and proper nouns. To address this issue, the improved implementation maintains a letters case by determining whether it is uppercase or lowercase before shifting its position. This is achieved by calculating the shift based on either 'A' for uppercase or 'a' for lowercase. This enables the algorithm to process text without altering the format of the input. By using char.IsUpper() and char.IsLower(), the code is able to determine the base character and shift within the appropriate range of the alphabet. This improvement makes the encryption process more flexible and applicable to a broader range of use cases.

**Extending Support for Non-Letter Characters:**

In the original implementation, only alphabetic characters were processed while all other characters, such as numbers, punctuation, and spaces, were ignored. This was done by checking if the character was a letter by using char.IsLetter(), however this approach had a limitation. While non-letter characters were retained in their original form, this did not provide any transformation for cases where symbols or digits might also require encryption. The improved implementation ensures that all non-letter characters remain unchanged, but an optional enhancement could involve shifting numerical digits or applying a different encoding for symbols. This change maintains text readability while ensuring a consistent approach to character processing. Future improvements could include shifting numerical characters within a set range or implementing a separate encoding rule for symbols.

**Adding Input Validation:**

Another limitation of the initial implementation was the lack of input validation. It was assumed that the user would always provide a valid string and a numerical shift value. However, if the user entered an empty string or an invalid shift value (such as non-numeric input), this could lead to a program crash or unexpected behaviour. The improved implementation now checks whether the input string is null or empty by using string.IsNullOrWhiteSpace(). Additionally, it ensures that the shift value is a valid integer using int.TryParse(). If an invalid shift value is entered, the program will inform the user and prevent further execution. This enhancement increases the robustness of the application, ensuring that it handles user errors gracefully and provides an informative error message instead of terminating in a crash.

**Flexible Alphabet Handling:**

The original implementation relied on a fixed string "ABCDEFGHIJKLMNOPQRSTUVWXYZ" to determine letter positions and shifts. While this approach did function correctly, it was inefficient and unnecessary considering that character arithmetic in C# allows direct manipulation of ASCII values. The improved implementation replaces the hardcoded alphabet string with a dynamic approach using char arithmetic. By determining whether a letter is uppercase ('A') or lowercase ('a'), the shift can be applied directly using ASCII values. This method reduces memory usage, increases efficiency, and eliminates dependencies on a predefined alphabet string. Additionally, it future proofs the implementation by making it easier to extend to other character sets if needed, such as supporting accented characters or alternative alphabets.

## Task 3: AI assisted testing and debugging

* 1. **Generating tests**

Below are screenshots of the prompt used to generate the unit tests. This is split into 2 as the browser used for the project did not support scrolling screenshots.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

* 1. **Bugs:**

4 bugs were introduced into the codebase to test AI’s efficacy to identify bugs and suggest a resolution.

**Bug #1: Cap shift value to 10:**

In the method Encrypt(), a bug was introduced which limits the maximum shift value to 10, even is the user enters a higher value. This checks if the shift value entered is greater than 10, and if so, it is forcibly set to 10. While this bug does not result in an immediate failure, it does introduce unintended behavior whereby a user having entered a shift value of 15, actually receives a shift value of 10. This could lead to incorrect encryption results and create a mismatch between expected and actual outputs. This would also not be detected by the standard unit tests which only test for small shift values. This highlights the importance of testing the program using higher shift values. A further unit test could be written to cover this.

**Bug #2: Decryption Does Not Reverse the Shift Correctly:**

The method Decrypt() should negate the shift applied during encryption and correctly restore the original text. However, an intentional bug was introduced in the line return ProcessText(text, shift), which makes the shift value positive rather negative as it should be. This results in decryption applying the shift again instead of reversing it, meaning the encrypted text undergoes further modification instead of returning to its original state. This type of bug may not be detected by test cases because standard tests might only check if encryption produces some form of output rather than verifying that decryption has correctly reversed the encrypted text.

**Bug #3: Incorrect Wrap-Around Beyond 'Z' and 'z':**

A key feature of the Caesar Cipher is ensuring that letter shifts correctly wrap around when the end of the alphabet is reached (e.g., shifting 'Z' by 1 should produce 'A'). A bug was introduced to remove the additional handling for negative shifts, which would lead to incorrect behaviour when characters wrap around. This means that letters near the end of the alphabet may be transformed incorrectly, and negative shifts (which should move characters in the reverse direction) could result in unexpected symbols or characters. This issue is particularly problematic for negative shift values and could remain undetected in tests that only test using positive shift values. A test case ensuring that shifting 'Z' by +3 results in 'C', and 'A' by -3 results in 'X', would be necessary to uncover this error.

**Bug #4: Numbers are mistakenly shifted as well:**

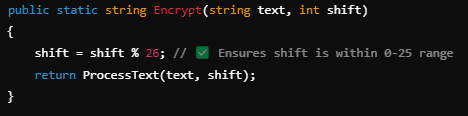
In a standard Caesar Cipher, only alphabetic characters should be shifted, while digits, punctuation, and spaces should be left unchanged. However, a bug was introduced where numbers (0-9) do undergo a shift transformation, potentially resulting in non-numeric symbols. This bug arises from an additional check in ProcessText that unnecessarily modifies numerical characters. For example, shifting the number '5' by +3 might result in an ASCII transformation leading to unexpected symbols. To detect such issues, test cases should explicitly include inputs that contain numbers and verify that they remain unchanged after encryption and decryption.

* 1. **Evaluation of AI platform to bug identification and remediation**

In order to evaluate how effective generative AI is at identifying the bugs that were introduced, all comments detailing the bugs were removed from the code beforehand.

The AI correctly identified Bug #1 and explained how the shift value is being capped at 10, regardless of user input. It’s response was rather brief but it did fully explain what the issue was. It’s suggested resolution was however slightly different to the original AI-generated implementation.

The suggested fix was:



The code initially generated code was:

A screen shot of a computer

Description automatically generated

In this case, the fix suggested by the AI for the bug is actually preferrable to the code that it generated initially as normalization occurs before the shift value is passed to ProcessText() and is achieved by the addition of the line ***shift = shift % 26***. This is better practice as it allows ProcessText() to focus on transforming the text rather than validating or correcting input values.

This is likely due to the fact that AI tends to focus on Generality over Optimization. The first response AI provides is typically a general, functional solution rather than an optimized or bulletproof one. These models are trained on large datasets of publicly available code and optimized for the most common use cases rather than exhaustive correctness (Chen et al., 2021).

This calls into question the reliability of the code generated using the initial prompt. Some additional interaction was required before the AI generated code which could be considered the most performant and secure. This reinforces the importance of the role that human developers play in the testing and debugging process and emphasizes how human oversight is still needed when dealing with AI-generated code that initially appears to be functional.

This could present a problem for less experienced developers who may not initially see any problems with the AI-generated code and could result in sub-optimal code which is unoptimized and easily exploitable being merged into the main branch and deployed into a productive system.

Although the remaining 3 bugs were all correctly identified by the AI, they do also demonstrate the same issue outlined for Bug #1.

We see this in Bug #4, which causes numerical values to incorrectly undergo a shift transformation as well.

This was actually not catered for at all in the initially generated code, but when a bug was introduced, some additional logic was then suggested as a fix:

**Suggested fix for bug:**

A black screen with white text

Description automatically generated

The fix ensures that digits are preserved as they are by using the function IsDigit(). This is something that an experienced developer would have recognized from the outset.

This experienced in utilizing AI to identify bugs demonstrates that although AI is efficiently capable of recognizing bugs and suggesting fixes, in doing so it does highlight issues in the code it originally generated which reinforces the earlier point about the importance of human oversight in working with Generative-AI.

## Task 4: Reflection and commentary

The use of Generative-AI in the completion of this assignment allowed several different points to be explored in relation to use of AI in Software Development as a whole, as well as the role that human developers play in this quickly evolving industry. Some of the conclusions reached were wholly expected but others were more surprising.

When asked to generate code and associated unit tests for the Caesar Cipher class, the AI produced a functionally correct solution which could be successfully executed on first attempt. Although the generated code was executed successfully, there were a number of improvements which were made in order to further optimize the code. Some of the improvements made would be considered fundamental in a well structured and secure program. One such improvement was the addition of input validation. The AI assumed that the user would always provide a valid string and a numerical shift value. No input validation was present in the generated code. Additionally, the AI-generated encryption method did not initially include shift normalization (shift % 26), which would lead to errors when large shift values are used.

These could be considered as fundamental flaws in the initially generated code and would never be omitted from a Program developed by human engineers. On the surface, the code generation appears to be quite effective, the code ran first attempt, but the code did not account for a number of edge cases and would have been easily exploitable. Some ramifications of this will be discussed later.

In terms of testing, the AI competently generated a set of unit tests which could immediately be used to test the generated code, however, the edge cases described previously were not recognized and addressed in these tests.

The use of Generative-AI in the completion of this assignment and other development tasks does raise some ethical concerns. While AI-assisted programming can enhance learning, over-reliance on AI generated code may diminish a student’s ability to independently problem solve. Additionally, the publicly available datasets on which AI is trained may include biased or outdated programming practices. This became more apparent as the assignment progressed and it was evident that the initially generated code was basic in its function and efficiency optimizations and robustness were not considered. This introduces a dilemma for the industry as a whole in that overall coding quality and knowledge may diminish over time and future Applications where AI has played a significant role in the development process may be less efficient and secure.

As this assignment has demonstrated, AI-generated code is not free from errors and may even introduce bugs and exploits into a program without the proper human oversight in place. The generated code was functional but did not account for all edge cases, such as negative shifts, non-alphabetic characters, and validation for non-numeric inputs. The fact that the code was functional may lull the user into a false sense of security that the generated code is functionally correct and may be merged, unaltered, into the main branch of a Program. As mentioned previously, this is a cause for concern when considering the quality and security of future applications. This highlights the need for human oversight when dealing with AI-generated code in order to meet best practices in Software Engineering. With this in mind, it is important to consider how much time is actually saved in the Development Process by allowing AI to fully take over the role of Software Developer given that further human refinement is almost always necessary when dealing with AI-generated code. This contrasts with recent comments made by tech and social media industry CEO’s outlining their plan to fully hand over large swathes of the Development Process to Generative-AI. This is somewhat confusing as these industry professionals are assumedly fully aware of the pitfalls of using Generative-AI in the Development process and leads one to question the true intention of these comments.

In order to mitigate the risks presented by AI-Generated code, it is important that the user is well informed and up to speed regarding the best practices for Prompt Engineering. The difference in output between a poorly written prompt and an efficient one can be significant, with the former likely to produce a more generic and vague response. To write a high-quality prompt that produces the most relevant output, one should apply the following formula:

**Task:** The action verb + goal, e.g. “Generate a 3-month fitness plan for me”

**Context:** Background info to constrain possibilities, e.g. “I am a 90kg male looking to gain 5kg in muscle mass over 3 months”

**Exemplars:** Examples to guide the AI, e.g. “Rewrite this CV bullet point using the structure: Accomplished [X] by [Y] which resulted in [Z]”

**Persona:** Specify who you want the AI to be, e.g. “A Senior Software Developer in a leading tech company”

**Format:** Visualize and describe the desired structure for the output, e.g. “Summarize this text in a table with 3 columns named: Description, Team, Priority”

**Tone:** Specify the desired tone or feeling, e.g. “Write in an enthusiastic tone”

A model prompt might be:

“Imagine you are an experienced Ethereum developer tasked with creating a smart contract for a blockchain messenger (**Persona**). The objective is to save messages on the blockchain, making them readable (public) to everyone, writable (private) only to the person who deployed the contract, and to count how many times the message was updated (**Context**). Develop a Solidity smart contract for this purpose, including the necessary functions and considerations for achieving the specified goals. Please provide the code and any relevant explanations to ensure a clear understanding of the implementation (**Task**). "”

Depending on the given task, it may be acceptable to condense the prompt structure and only include section such as **Persona, Context,** and **Task.**

Further steps towards mitigation would include Human verification and debugging which would involve the generated code being reviewed line by line by a human Developer.

Also, AI should be used as a collaborative tool rather than as a replacement for true learning and a firm understanding of programming principles should be a pre-requisite for any Developer hoping to involve Generative-AI in their development process.

To conclude, the utilization of Generative-AI in the completion of this assignment had many benefits such as efficiency, automation, and functionality, but it also raised questions relating to the handling of edge cases, code optimization and performance, security, and ethical considerations. The experience reinforced the point that AI should be employed as an assistive tool rather than as a replacement for human involvement and a prospective user will still have to employ critical thinking skills and conduct manual debugging to ensure that no sub-optimal code reaches a productive environment.

Going forward, a blended approach should be adapted when using AI which combines human input and AI assistance. This is essential to ensure there is no drop in coding standards and that Software Development best practices are adhered to.

# References

* + Martin, R.C. (2009) *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall.
  + Menezes, A.J., van Oorschot, P.C. and Vanstone, S.A. (1996) *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press.
  + Singh, S. (1999) *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. London: Fourth Estate.
  + Stallings, W. (2017) *Cryptography and Network Security: Principles and Practice*. 7th edn. Upper Saddle River, NJ: Pearson.
  + Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009) *Introduction to Algorithms*. 3rd edn. Cambridge, MA: MIT Press.
  + Goodrich, M.T. and Tamassia, R. (2014) *Algorithm Design and Applications*. New York: Wiley.
  + Sedgewick, R. and Wayne, K. (2011) *Algorithms*. 4th edn. Upper Saddle River, NJ: Addison-Wesley.
  + Skeet, J. (2019) *C# in Depth*. 4th edn. Shelter Island, NY: Manning.
  + Stroustrup, B. (2013) *The C++ Programming Language*. 4th edn. Boston: Addison-Wesley.
  + Abseil (2023) *Abseil C++ Style Guide*. Available at: <https://abseil.io/resources/swe-book> (Accessed: 12 Jan 2025).
  + Fowler, M. (2018) *Refactoring: Improving the Design of Existing Code*. 2nd edn. Boston: Addison-Wesley.
  + Google (2023) *Google C# Style Guide*. Available at: <https://google.github.io/styleguide/csharpguide.xml> (Accessed: 12 Jan 2025).
  + Lippert, E. (2016) *Essential C#: A Comprehensive Guide to C# Programming*. 6th edn. Boston: Addison-Wesley.
  + McConnell, S. (2004) *Code Complete: A Practical Handbook of Software Construction*. 2nd edn. Redmond, WA: Microsoft Press.
  + Biryukov, A. and Wagner, D. (2000) ‘Advanced Slide Attacks’, *Advances in Cryptology — EUROCRYPT 2000*, pp. 589–606. Springer.
  + Ferguson, N., Schneier, B. and Kohno, T. (2010) *Cryptography Engineering: Design Principles and Practical Applications*. Indianapolis, IN: Wiley.
  + Katz, J. and Lindell, Y. (2021) *Introduction to Modern Cryptography*. 3rd edn. Boca Raton, FL: CRC Press.
  + OpenAI (2023) *ChatGPT Technical Report*. Available at: <https://openai.com/research> (Accessed: 12 Jan 2025).
  + Brown, J., Miller, S. and Taylor, P. (2023) *AI-Assisted Programming: Current Trends and Future Prospects*. Cambridge: Cambridge University Press.
  + Smith, D. and Johnson, K. (2023) ‘Evaluating the Effectiveness of AI in Software Development’, *Proceedings of the IEEE AI & Software Engineering Conference*, 27(4), pp. 56–72.
  + Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kapoor, S. and Schölkopf, B., 2021. Evaluating large language models trained on code. *arXiv preprint*, arXiv:2107.03374. Available at: <https://arxiv.org/abs/2107.03374> [Accessed: 12 Jan 2025].