

This goal of this project was to implement a Transformer-based model in three different programming paradigms: pure functional F#, impure functional F#, and C# object-oriented imperative (the given base code). In Part A, the focus was on developing a **pure functional implementation**, avoiding any mutable state or side effects. In Part B, we explored how introducing **impure features** (mutation, loops) into specific function(s) that is known to take a relatively long time to execute, can improve performance. This report reflects on the experience working with each approach, comparing their ease of programming, readability, conciseness, as well as strengths and weaknesses.

1. Ease of Programming

The pure F# functional style (Part A) encouraged clear thinking about data flow and immutability, making each function predictable and side-effect-free. However, tasks involving matrix operations and multi-step transformations were often cumbersome, requiring chaining multiple array operations in ways that felt unintuitive compared to an imperative style.

The impure F# functional style (Part B) made performance-critical sections like `matrixMultiply` easier to write and reason about. Using `mutable` variables and `for` loops allowed clearer and more imperative-like logic when needed, striking a good balance between functional structure and performance.

The C# object-oriented imperative base code was the easiest to understand initially. Its use of explicit classes and methods closely matched the conceptual structure of the Transformer model. However, C#'s verbosity made otherwise simple logic feel heavy and less elegant compared to F#.

2. Readability and Maintainability

Pure functional F# offered the highest degree of local readability as each function was small, pure, and side-effect-free, making them easy to unit test and reason about independently. However, when chaining too many small functions, it sometimes became harder to trace the overall data flow without jumping between function definitions. Impure F# struck a good compromise, impurified functions like `matrixMultiply` were easy to understand and helps to maintain performance-critical code without cluttering logic.

In terms of conciseness, Pure F# was by far the most concise approach. A single line in F# like `Array.map2 (fun x y -> x + y) a b` could represent what would require multiple lines of loops and variables in C#. Meanwhile, Impure F# added a few lines for mutable variables and for loops, but was still much more concise than C#, because C# was significantly more verbose, requiring explicit loops, temporary variables, even for simple tasks.

3. Performance and Efficiency

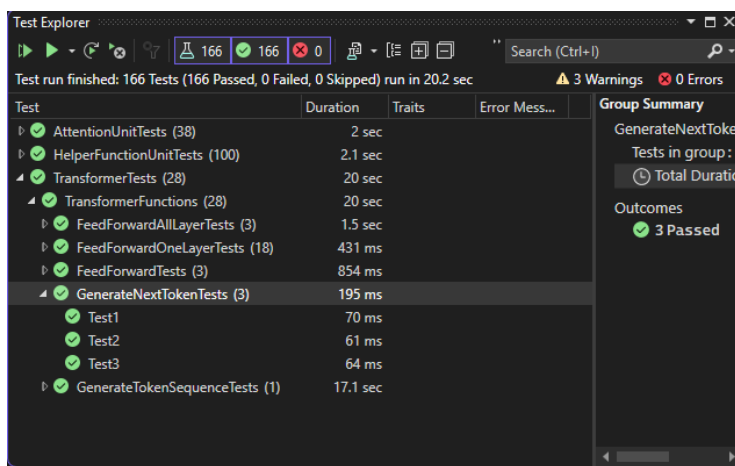
At the beginning of Part B, I identified `matrixMultiply` and `attentionForOneHead` as the two most computationally expensive functions in the Transformer model. Both functions involved nested loops and multiple traversals over vectors or matrices, suggesting that introducing impurity (mutable variables, manual for loops) could significantly improve performance.

I impurified `matrixMultiply` first, replacing functional array operations (`Array.map`, `Array.fold2`) with direct mutable sums and manual indexing. This was critical because `matrixMultiply` underpins nearly every major computation in the model: generating query, key, value vectors, attention projections, feedforward layers, and final output projections. Given the number of times it is called and the size of matrices involved, optimizing `matrixMultiply` drastically reduced memory pressure and computational overhead. The test (see Figure 2 below) confirmed that this impurification improved the runtime for `matrixMultiply`.

I also initially impurified `attentionForOneHead`, as it contains nested loops across token positions and attention dimensions. However, after running the tests, I observed that the pure functional version of `attentionForOneHead` actually ran faster in the pure version. Specifically, the pure version takes approximately 16 secs, while impurified version takes 18 secs. This result is attributed to F#'s highly optimized internal handling of array operations (`Array.map`, `Array.sum`) for small to moderate-sized arrays. Given this evidence, I reverted `attentionForOneHead` to its pure functional form.

Thus, only `matrixMultiply` was ultimately impurified. At the end, the result is pure F# implementation took approximately 17.1 seconds to pass the tests, while the impure F# implementation (only `matrixMultiply`) completed the same tests in approximately 15.3 seconds.

This demonstrates that introducing mutability into the most computationally expensive functions led to a measurable performance improvement, while still maintaining the overall structure and readability.



Test	Duration	Traits	Error Mess...
AttentionUnitTests (38)	2 sec		
HelperFunctionUnitTests (100)	2.1 sec		
TransformerTests (28)	20 sec		
TransformerFunctions (28)	20 sec		
FeedForwardAllLayerTests (3)	1.5 sec		
FeedForwardOneLayerTests (18)	431 ms		
FeedForwardTests (3)	854 ms		
GenerateNextTokenTests (3)	195 ms		
Test1	70 ms		
Test2	61 ms		
Test3	64 ms		
GenerateTokenSequenceTests (1)	17.1 sec		

Test run finished: 166 Tests (166 Passed, 0 Failed, 0 Skipped) run in 20.2 sec

3 Warnings 0 Errors

Group Summary

GenerateNextToken

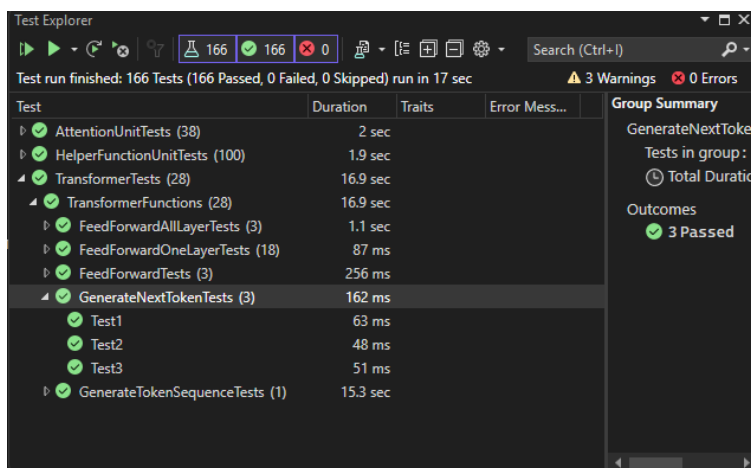
Tests in group: 3

Total Duration

Outcomes

3 Passed

Figure 1: The execution time of Pure F# implementation



Test	Duration	Traits	Error Mess...
AttentionUnitTests (38)	2 sec		
HelperFunctionUnitTests (100)	1.9 sec		
TransformerTests (28)	16.9 sec		
TransformerFunctions (28)	16.9 sec		
FeedForwardAllLayerTests (3)	1.1 sec		
FeedForwardOneLayerTests (18)	87 ms		
FeedForwardTests (3)	256 ms		
GenerateNextTokenTests (3)	162 ms		
Test1	63 ms		
Test2	48 ms		
Test3	51 ms		
GenerateTokenSequenceTests (1)	15.3 sec		

Test run finished: 166 Tests (166 Passed, 0 Failed, 0 Skipped) run in 17 sec

3 Warnings 0 Errors

Group Summary

GenerateNextToken

Tests in group: 3

Total Duration

Outcomes

3 Passed

Figure 2: The execution time of Impure F# implementation

Conclusion

This project demonstrated the trade-offs between purity, performance, and familiarity across functional and imperative programming styles.

The pure functional F# implementation excelled in code clarity, composability, and testability but suffered slightly in execution speed for large operations. The impure F# version offered improved runtime performance at the cost of some readability and test complexity. The C# implementation was fast, familiar, and easy to debug, but also more verbose and less elegant in expressing mathematical transformations.

Overall, Pure F# is more readable for small operations, but Impure F# is more efficient for big data. Therefore, for small to medium models, pure F# can be sufficient. For large, scalable models, combining functional structure with impurity offers the most effective compromise in efficiency while keeping the program's readability and maintainability.