From .NET Core to .NET 8: A Comprehensive Analysis of Performance, Features, and Migration Pathways

Branimir Cvijić[1], Pero Ranilović[2]
[1]*Endava, Banja Luka, RS, BiH, cvijic.branimir@gmail.com*
[2]*Pan-European University "Apeiron", Banja Luka, RS, BiH, pero.d.ranilovic@apeiron-edu.eu*

Review paper
https://doi.org/10.7251/JIT2401069C UDC: 004.432.2C#:004.738-.1

Abstract: This analysis embarks on a comprehensive exploration of the .NET ecosystem's evolution, with a spotlight on the transition from .NET Core to the unified .NET platform, culminating in the release of .NET 8. It meticulously examines the performance enhancements, feature evolutions, and migration strategies that underscore this transition, providing a lens through which the future trajectory of .NET, including the anticipation of .NET 9, can be discerned. By offering a deep dive into the comparative performance metrics and the introduction of novel features across versions, this paper caters to IT professionals, students, and technology aficionados seeking to grasp the full extent of .NET's capabilities and its strategic direction. The findings aim to not only delineate the technical advancements but also to contextualize the platform's ongoing innovation within the broader software development ecosystem.

Keywords: .NET core, Unified .NET platform, Migration strategies, Performance benchmarking

Introduction

Before the advent of .NET Core, the .NET Framework dominated Microsoft's development landscape, tailored primarily for Windows applications. As global software development trends shifted towards more agile, scalable, and cross-platform solutions, the need for a more flexible framework became evident. In response, Microsoft introduced .NET Core, fundamentally designed to address these modern computing requirements with support for modular, cross-platform development across Windows, Linux, and macOS. Launched as an open-source framework, .NET Core represented a significant departure from the traditional .NET Framework. [1] The evolution of the .NET ecosystem, particularly the transition from .NET Core to the unified .NET platform, marks a pivotal shift in the landscape of software development. This journey, which extends to the advent of .NET 8, is not merely a series of technological advancements but a comprehensive strategic realignment towards creating an inclusive, performance-optimized, and feature-rich ecosystem. The cornerstone of this evolutionary path was .NET Core 3.1, celebrated as the last Long-Term Support (LTS) version within the .NET Core series. It established a robust foundation that catalyzed the seamless transition to subsequent versions, each introducing significant enhancements and capabilities.

This paper aims to dissect the intricate progression from .NET Core 3.1 to .NET 8, providing a granular analysis of performance improvements, feature augmentations, and the nuances of migration strategies. By delving into the evolution of the platform, the analysis is tailored to offer IT professionals, students, and technology enthusiasts a detailed comprehension of .NET's expansive capabilities and its trajectory towards future developments. Furthermore, a preliminary overview of .NET 9 is included, offering insights into the continuing innovation within the .NET ecosystem.

A focus on detailed performance benchmarks and feature analysis will underscore the platform's developmental milestones, illuminating the strategic insights gleaned through its evolution. This paper is committed to mapping out .NET's transformative journey, highlighting the technological and strategic milestones that have underpinned its growth and continue to shape its future.

.NET Core overview

.NET Core has represented a significant shift in the .NET framework's development, aiming to provide a more modular, cross-platform development experience. [1] It was designed from the ground up to support the development of applications for Windows, Linux, and macOS, thereby broadening the .NET ecosystem beyond its traditional Windows-centric roots. The introduction of .NET Core was a response to the evolving needs of the software development community, emphasizing performance, scalability, and the ability to run in diverse environments.

Genesys and evolution of .NET Core

.NET Core's journey began as a lean and composable framework that sought to address the emerging trends in software development, including cloud-based applications and microservices architectures. It introduced a side-by-side installation feature, allowing different versions of .NET Core to coexist on the same machine and thus enabling greater flexibility in application deployment and maintenance. Over its lifecycle, .NET Core saw rapid iteration and improvement, with each version bringing performance enhancements, expanded API sets, and better tooling.

Spotlight on .NET Core 3.1

When .NET Core 3.1 was released in December 2019, it was declared the final and most polished version of the .NET Core series. As an LTS version, it was guaranteed support from Microsoft for three years, making it a stable base for developers looking to build high-performance web and cloud applications. .NET Core 3.1 brought several key features and improvements: [2]

- Enhanced Performance: Continuing the .NET Core tradition, 3.1 improved on its predecessors' performance metrics, offering more efficient memory usage, faster algorithms, and optimizations in the core libraries.
- Desktop application support: With the introduction of Windows Forms and WPF (Windows Presentation Foundation) on Windows, .NET Core 3.1 bridged the gap between modern web application development and traditional desktop application development.
- Improved Container Support: Recognizing the importance of containerization and microservices, .NET Core 3.1 enhanced its capabilities to run more efficiently in containers, including smaller image sizes and more customizable runtime images. [3]
- Platform expansion: - It maintained support for a broad array of operating systems, further solidifying .NET Core's position as a versatile, cross-platform framework.

Transition to the unified .NET platform

.NET Core 3.1 set the stage for the transition to the unified .NET platform, starting with .NET 5. This transition aimed to bring together the best of .NET Core, .NET Framework, Xamarin, and Mono under a single platform, simplifying the .NET landscape and offering a unified path forward for all types of .NET development. The move represented not just an evolutionary step in terms of features and performance but also a unification of the .NET ecosystems, streamlining the development experience across application types and platforms.

Prelude to AI integration in .NET 8

Looking beyond .NET Core 3.1, the .NET platform continues to evolve, with .NET 8 introducing AI and machine learning capabilities as a testament to the framework's adaptability and forward-thinking design. These integrations signal a future where .NET is not just about building applications but also about incorporating intelligent features and data-driven insights directly into those applications.

Key Feature Evolution in the .NET platform

Exploring the significant enhancements and new capabilities introduced with each successive version of the .NET platform, from the foundational .NET Core 3.1 through to the innovative .NET 8, reveals a trajectory of continuous improvement and expansion. This journey reflects Microsoft's dedication to addressing the evolving needs of developers and organizations, through the provision of a versatile and powerful framework capable of supporting the development of modern applications. Here's a closer look at how these features have evolved, emphasizing the platform's adaptability and forward-looking approach.

From .NET Core 3.1 to Unified Platform

Unified platform transition: .NET 5 marked a pivotal milestone, merging .NET Core, .NET Framework,

Xamarin, and Mono into a single, streamlined framework, simplifying the development ecosystem. [4]

C# 9 and F# 5 Enhancements

Introduced significant language features like records in C# for immutable data models and pattern matching improvements, alongside new F# features to enhance functional programming.

Blazor WebAssembly: Facilitated the development of full-stack web applications with C#, running client-side logic in the browser via WebAssembly, thereby broadening the scope of .NET in web development.

Progression with .NET 6: Enhancing Productivity and Performance

Minimal APIs: Introduced to simplify the creation of HTTP APIs, these APIs reduce the boilerplate code necessary for setting up microservices and small web applications.

C# 10 Innovations: Brought global using directives and record structs, among other features, making code more concise and improving developer productivity.

MAUI Preview: .NET Multi-platform App UI (MAUI) previewed, offering a path toward building cross-platform mobile and desktop apps from a single codebase.

.NET 7 Connectivity, Cloud Optimization and Code Refinement

Language Advancements: C# 11 and F# 6 introduced further enhancements, such as list patterns in C# for more expressive code and syntax improvements in F#.

ASP.NET Core and Blazor Improvements: Significant advancements in web development capabilities, including better Blazor components and SignalR client reconnections, underscored .NET's commitment to web technologies.

.NET MAUI official release: Delivered a robust framework for developing native applications across Android, iOS, macOS, and Windows, streamlining cross-platform development.

.NET 8, Integrating AI and expanding capabilities

AI and Machine Learning Integration: Demonstrated .NET's adaptability by incorporating AI tooling and libraries, empowering developers to build intelligent, data-driven applications. [5]

Continuous Enhancements: Ongoing improvements in C# and core platform features focused on elevating developer productivity, optimizing application performance, and enhancing cross-platform support.

Improved Native Interop: Made it easier to integrate with native libraries, critical for applications requiring high performance and direct access to underlying system capabilities.

Performance Enhancements in .NET Core

.NET Core has undergone significant performance optimizations across its lifecycle, with particular focus on JIT Compiler improvements and Garbage Collector (GC) enhancements. These efforts have been crucial in ensuring that .NET Core remains a robust and efficient framework suitable for a wide range of applications, from desktop and web applications to cloud-based services and microservices architectures. The enhancements to the JIT compiler and garbage collector across the .NET Core versions underscore Microsoft's commitment to performance, efficiency, and the modernization of application development. [6]

Besides that, performance trajectory of the .NET Core platform has been marked by continuous advancements aimed at optimizing startup times, memory usage, computational efficiency, and ensuring consistent performance across multiple operating systems. These enhancements have solidified .NET Core's position as a high-performance, efficient framework suitable for a diverse range of applications.

JIT Compiler Enhancements

- Optimized Code Generation:

Across its evolution, .NET Core's JIT compiler has continually improved its code generation strategies. This includes better inlining of methods (where the code of a called method is inserted into the caller's body), which can significantly reduce call overhead and improve execution speed.

- Tiered Compilation:

Introduced and refined over time, tiered compilation helps achieve a balance between fast startup times and optimized application performance. Initially, methods are compiled for quick execution, and as they are identified as frequently used ("hot" methods), they are recompiled with optimizations.

- Dynamic PGO (Profile-Guided Optimization):

Enhanced in later versions, Dynamic PGO utilizes runtime performance data to optimize code paths, significantly boosting the efficiency of JIT compilation and overall application performance.

Garbage Collector (GC) Improvements

- Efficiency and Throughput:

The GC has been optimized for high efficiency and throughput, with specific enhancements aimed at reducing pause times. This means applications experience fewer interruptions for GC, leading to smoother performance.

- Container Support:

As .NET Core embraced containerization, the GC received optimizations for running in container environments. This includes scaling to the available resources and constraints within a container, ensuring applications perform well even in memory-limited situations.

- Concurrent Garbage Collection:

Enhancements to concurrent GC operations have minimized the impact on application responsiveness. This allows the GC to reclaim memory in the background, reducing pauses in the application's execution

Cross-Platform Performance Consistency

Ensuring consistent performance across diverse operating systems is a cornerstone of .NET Core's design philosophy. The framework's cross-platform capabilities are supported by comprehensive optimizations to the runtime and framework libraries, which are crucial for optimal performance on Windows, Linux, and macOS. These enhancements focus on modularizing system dependencies and improving access to native APIs, ensuring applications perform efficiently across all platforms. [7]

Real-World Impact and Benchmarks

Benchmark tests across various versions of .NET Core consistently demonstrate improvements in startup times, memory efficiency, and overall performance. These enhancements contribute to a more robust and scalable application performance, meeting and often exceeding modern application performance expectations.

This approach to performance optimization not only maintains .NET Core's competitiveness but also anticipates future challenges in software development, preparing the platform with advanced capabilities like Blazor for web assembly and .NET MAUI, and integrating AI tools in .NET 8. These innovations underscore .NET's readiness to embrace emerging technologies and adapt to the evolving landscape of software development.

Practical Performance Analysis in the .NET Ecosystem

A. Definition of Benchmarking Environment

This section focuses on the practical implementation of performance benchmarking within the .NET ecosystem. Providing accurate results requires a detailed definition of the hardware and software characteristics used in the benchmarking process. This segment provides basic information about the hardware and software configuration to enable precise comparison of performance across different versions of the .NET framework.

The specification of the laptop used for benchmarking is as follows:

- 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
- 16GB DDR4 RAM
- SK Hynix PC711 512GB
- Windows 10 Pro

In the benchmarking process, three key versions of the .NET framework were used for performance comparison. Versions .NET Core 3.1, .NET 6.0, .NET 7.0, and .NET 8.0 were analyzed. Each version was carefully selected to explore potential performance differences and any optimizations implemented in newer versions.

BenchmarkDotNet was used as the benchmarking tool, widely accepted and recognized tool in the .NET community for measuring performance. BenchmarkDotNet provides a flexible and reliable framework for conducting benchmark tests, enabling automatic management of many details such as measurement stability, error handling, and precise result comparison. Thanks to its simplicity and power, BenchmarkDotNet is an ideal tool for performing detailed performance analysis in the .NET ecosystem. BenchmarkDotNet has already been adopted by over 19,100 GitHub projects, including .NET Runtime, .NET Compiler, .NET Performance, and many others. [8]

Benchmarking process included the following steps:

Environment preparation: System configuration was standardized to ensure test consistency. This involved setting up appropriate versions of the .NET framework and ensuring all relevant software and hardware parameters remained constant throughout all tests.

Test execution: Each version of the .NET framework was tested using the same set of tests, which included various types of loads and scenarios to obtain a comprehensive performance picture.

Results analysis: Results were collected and analyzed using BenchmarkDotNet, which provides detailed reports including metrics such as execution time, memory consumption, and performance stability.

Limitations and potential biases

While the methodology was carefully designed, there are certain limitations and potential biases to consider:

Hardware limitations: Performance may vary depending on hardware configuration. Although using the same system for all tests reduced variability, results may differ on other systems.

Software variables: Operating system versions, drivers, and other software components can affect performance. Testing was conducted on Windows 10 Pro, but different configurations may yield different results.

Benchmarking tool: Although BenchmarkDotNet is a reliable tool, every benchmarking tool has its limitations and may introduce certain biases into results. For example, optimizations specific to BenchmarkDotNet may affect real-world application performance.

External variables: External factors such as background processes and system state can influence test results, despite efforts to minimize such influences.

B. Time Comparison of Compute-Intensive Operations

This section explores the time difference between the execution of compute-intensive operations, including computing SHA256 and MD5 hash values, on different versions of .NET Framework: .NET Core 3.1, .NET 6.0, .NET 7.0 and .NET 8.0. The aim is to compare the performance of these operations across different framework versions and identify potential differences in speed between them.

SHA256 (Secure Hash Algorithm 256-bit) and MD5 (Message Digest Algorithm 5) are cryptographic hash algorithms used to generate unique digital signatures or "hash" values from input data. These algorithms apply mathematical functions to input data to generate a unique string of bits representing the digital fingerprint or "hash" of the original data. For testing purposes, a bit array value of 10,000 was used.

```
[SimpleJob(RuntimeMoniker.NetCoreApp31)]
[SimpleJob(RuntimeMoniker.Net60)]
[SimpleJob(RuntimeMoniker.Net70)]
[SimpleJob(RuntimeMoniker.Net80)]
1 reference
public class Md5VsSha256
{
    private const int N = 10000;
    private readonly byte[] data;

    private readonly SHA256 sha256 = SHA256.Create();
    private readonly MD5 md5 = MD5.Create();

    0 references
    public Md5VsSha256()
    {
        data = new byte[N];
        new Random(42).NextBytes(data);
    }

    [Benchmark]
    0 references
    public byte[] Sha256() => sha256.ComputeHash(data);

    [Benchmark]
    0 references
    public byte[] Md5() => md5.ComputeHash(data);
}
```

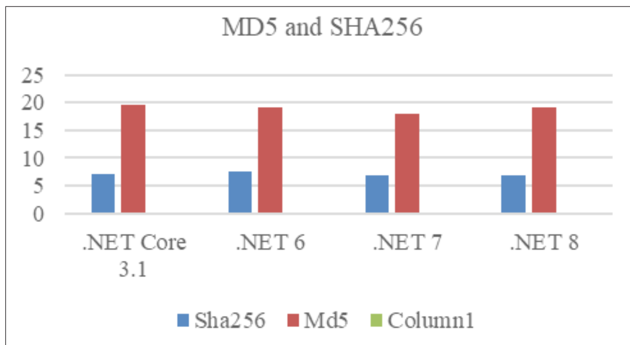
Figure 1. *MD5 vs SHA256 benchmark code*

When the SHA256 or MD5 algorithm is applied to a dataset, the resulting hash value will be unique for that dataset. Even the slightest change in the input data will produce a completely different hash value.

Table 1. *Comprehensive overview of the results of execution of MD5 and SHA256 algorithms on different versions of .NET*

Method	Job	Runtime	Mean	Error	StdDev	Median
Sha256	.NET 6.0	.NET 6.0	7.401 μs	0.2467 μs	0.7118 μs	7.447 μs
Md5	.NET 6.0	.NET 6.0	19.447 μs	0.6093 μs	1.7579 μs	19.075 μs
Sha256	.NET 7.0	.NET 7.0	7.138 μs	0.1528 μs	0.4432 μs	6.940 μs
Md5	.NET 7.0	.NET 7.0	17.879 μs	0.3436 μs	0.3375 μs	17.911 μs
Sha256	.NET 8.0	.NET 8.0	6.889 μs	0.1256 μs	0.1880 μs	6.816 μs
Md5	.NET 8.0	.NET 8.0	19.079 μs	0.3755 μs	0.3856 μs	19.167 μs
Sha256	.NET Core 3.1	.NET Core 3.1	7.030 μs	0.1207 μs	0.1983 μs	6.996 μs
Md5	.NET Core 3.1	.NET Core 3.1	20.064 μs	0.6292 μs	1.8254 μs	19.674 μs

Results show that the performance of SHA256 and MD5 algorithms has improved with each new version of the .NET framework. The average execution time of the SHA256 algorithm decreases from .NET 6.0 to .NET 8.0, suggesting continuous performance improvement. A similar trend is observed with the MD5 algorithm. However, in all versions of the .NET framework, the SHA256 algorithm demonstrates faster execution time compared to the MD5 algorithm. This indicates superior performance of the SHA256 algorithm in this specific benchmark test.



Graph 1. *Comprehensive overview of the results of execution of MD5 and SHA256 algorithms on different versions of .NET*

The standard deviations for both algorithms are low, suggesting consistent performance and reliable results. Overall, the results demonstrate a positive evolution of performance for cryptographic algorithms in the .NET ecosystem over time.

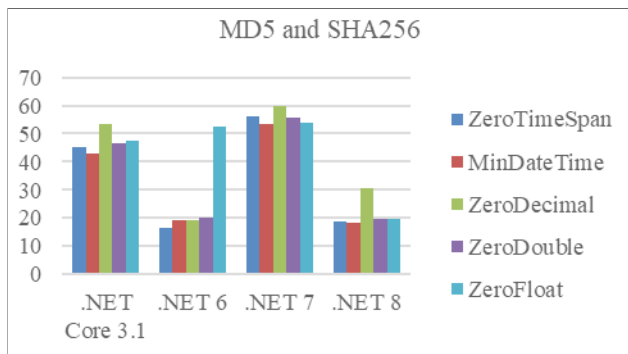
C. Time Comparison of Asynchronous Operations Execution

In this section of the research, we will focus on the comparative analysis of executing asynchronous operations in different versions of the .NET framework. The asynchronous approach enables efficient management of execution time for operations that require waiting for I/O operations or processing long-running tasks. Through this research, we aim to assess how the performance of asynchronous operations differs between different versions of the .NET framework, as well as to identify any improvements or changes in execution time in newer framework versions. This is important to understand the impact of the evolution of the .NET platform on the efficiency of asynchronous programming and potential opportunities for optimizing coding.

```
[MarkdownExporter, AsciiDocExporter, HtmlExporter, CsvExporter, RPlotExporter]
[SimpleJob(RuntimeMoniker.Net60)]
[SimpleJob(RuntimeMoniker.Net70)]
[SimpleJob(RuntimeMoniker.Net80)]
[MemoryDiagnoser(true)]
1 reference
public class Tests
{
    0 references
    [Benchmark] public async Task<TimeSpan> ZeroTimeSpan() => TimeSpan.Zero;
    0 references
    [Benchmark] public async Task<DateTime> MinDateTime() => DateTime.MinValue;
    0 references
    [Benchmark] public async Task<Guid> EmptyGuid() => Guid.Empty;
    0 references
    [Benchmark] public async Task<DayOfWeek> Sunday() => DayOfWeek.Sunday;
    0 references
    [Benchmark] public async Task<decimal> ZeroDecimal() => 0m;
    0 references
    [Benchmark] public async Task<double> ZeroDouble() => 0;
    0 references
    [Benchmark] public async Task<float> ZeroFloat() => 0;
    0 references
    [Benchmark] public async Task<Half> ZeroHalf() => (Half)0f;
    0 references
    [Benchmark] public async Task<(int, int)> ZeroZeroValueTuple() => (0, 0);
}
```

Figure 2. *Async methods for benchmark*

Benchmark tests represent asynchronous methods that return different types of data with fixed values (Figure 2.). Each method returns a unique data type with its corresponding fixed value, such as TimeSpan.Zero, DateTime.MinValue, Guid.Empty, DayOfWeek.Sunday, 0m, 0, 0f, (Half)0f, and (0, 0) for the value type Tuple. The goal of these tests is to measure the execution time of asynchronous operations that return constant values. This approach enables the assessment of the efficiency of asynchronous execution for different data types and can provide insights into performance and potential optimizations in working with this data in the .NET ecosystem.



Graph 2. *Comprehensive overview of the results of execution async operation*

Table 2. *Comprehensive overview of the results of execution async operation*

Method	Job	Runtime	Mean	Error	StdDev	Median	Gen0	Allocated
ZeroTimeSpan	.NET 6.0	.NET 6.0	16.71 ms	0.322 ms	0.251 ms	16.65 ms	0.0013	72 B
MinDateTime	.NET 6.0	.NET 6.0	19.90 ms	0.655 ms	1.900 ms	19.22 ms	0.0013	72 B
EmptyGuid	.NET 6.0	.NET 6.0	19.73 ms	0.714 ms	2.060 ms	19.40 ms	0.0015	80 B
Sunday	.NET 6.0	.NET 6.0	15.68 ms	0.264 ms	0.441 ms	15.58 ms	0.0013	72 B
ZeroDecimal	.NET 6.0	.NET 6.0	19.07 ms	0.119 ms	0.266 ms	19.08 ms	0.0015	80 B
ZeroDouble	.NET 6.0	.NET 6.0	22.04 ms	2.066 ms	5.862 ms	20.28 ms	0.0013	72 B
ZeroFloat	.NET 6.0	.NET 6.0	50.35 ms	3.051 ms	8.948 ms	52.36 ms	0.0013	72 B
ZeroZeroValueTuple	.NET 6.0	.NET 6.0	48.09 ms	1.037 ms	2.856 ms	47.85 ms	0.0013	72 B
ZeroTimeSpan	.NET 7.0	.NET 7.0	56.11 ms	1.277 ms	3.764 ms	55.99 ms	0.0013	72 B
MinDateTime	.NET 7.0	.NET 7.0	54.24 ms	1.147 ms	2.769 ms	53.63 ms	0.0013	72 B
EmptyGuid	.NET 7.0	.NET 7.0	58.56 ms	1.242 ms	3.564 ms	57.68 ms	0.0014	80 B
Sunday	.NET 7.0	.NET 7.0	54.30 ms	1.431 ms	4.221 ms	53.64 ms	0.0013	72 B
ZeroDecimal	.NET 7.0	.NET 7.0	59.73 ms	1.257 ms	3.440 ms	56.65 ms	0.0014	80 B
ZeroDouble	.NET 7.0	.NET 7.0	56.13 ms	1.195 ms	2.998 ms	55.65 ms	0.0013	72 B
ZeroFloat	.NET 7.0	.NET 7.0	54.54 ms	1.177 ms	3.470 ms	53.94 ms	0.0013	72 B
ZeroZeroValueTuple	.NET 7.0	.NET 7.0	57.69 ms	1.223 ms	3.390 ms	57.48 ms	0.0013	72 B
ZeroTimeSpan	.NET 8.0	.NET 8.0	18.97 ms	0.460 ms	1.127 ms	18.54 ms	-	-
MinDateTime	.NET 8.0	.NET 8.0	18.65 ms	0.438 ms	0.886 ms	18.40 ms	-	-
EmptyGuid	.NET 8.0	.NET 8.0	21.04 ms	0.501 ms	1.151 ms	20.73 ms	-	-
Sunday	.NET 8.0	.NET 8.0	18.15 ms	0.374 ms	0.548 ms	18.13 ms	-	-
ZeroDecimal	.NET 8.0	.NET 8.0	29.73 ms	1.125 ms	3.245 ms	30.51 ms	-	-
ZeroDouble	.NET 8.0	.NET 8.0	20.03 ms	0.469 ms	1.141 ms	19.71 ms	-	-
ZeroFloat	.NET 8.0	.NET 8.0	19.59 ms	0.475 ms	0.856 ms	19.55 ms	-	-
ZeroZeroValueTuple	.NET 8.0	.NET 8.0	27.59 ms	0.614 ms	0.603 ms	27.56 ms	-	-
ZeroTimeSpan	.NET Core 3.1	.NET Core 3.1	45.44 ms	0.963 ms	1.444 ms	45.32 ms	0.0012	72 B
MinDateTime	.NET Core 3.1	.NET Core 3.1	43.29 ms	0.725 ms	0.864 ms	42.91 ms	0.0012	72 B
EmptyGuid	.NET Core 3.1	.NET Core 3.1	55.04 ms	1.140 ms	1.561 ms	55.01 ms	0.0013	80 B
Sunday	.NET Core 3.1	.NET Core 3.1	46.49 ms	2.005 ms	5.818 ms	47.45 ms	0.0012	72 B
ZeroDecimal	.NET Core 3.1	.NET Core 3.1	53.17 ms	1.131 ms	2.160 ms	53.21 ms	0.0013	80 B
ZeroDouble	.NET Core 3.1	.NET Core 3.1	47.21 ms	1.019 ms	2.518 ms	46.79 ms	0.0012	72 B
ZeroFloat	.NET Core 3.1	.NET Core 3.1	47.66 ms	1.020 ms	2.196 ms	47.47 ms	0.0012	72 B
ZeroZeroValueTuple	.NET Core 3.1	.NET Core 3.1	47.94 ms	1.020 ms	2.013 ms	47.70 ms	0.0012	72 B

By analyzing the obtained results, we can conclude that the average execution time of asynchronous operations is significantly lower when using the .NET 8.0 version. Let's compare the execution times of the ZeroFloat method across different versions of .NET. When executed on version 8.0, the time obtained was 19.55ns, on version 7.0 it was 53.94ns, on version 6.0 it was 52.36ns, while on version .NET Core 3.1 it was 47.47ns. We see that the difference in time is significant. The situation is similar with other methods used in the measurement. ZeroDecimal method can also be singled out, which had the shortest time when using .NET 6.0 version - 19.08ns, .NET 7.0 version – 59.65ns, .NET 8.0 version – 30.51ns, .NET Core 3.1 – 53.21ns. ZeroDecimal is the method that gave the longest execution time during measurement using .NET 8.0.

D. Use of GC advancements in .NET 8 version

In .NET 8, the GC (Garbage Collection) server now supports a dynamic heap count. In .NET 8, it is generally off by default but can be enabled by adding the <GarbageCollectionAdaptationMode>1</GarbageCollectionAdaptationMode> property within MSBuild. The employed algorithm can increase and decrease the heap count over time, aiming to maximize its view of throughput while maintaining a balance between that and the overall memory footprint. A scenario demonstrating how the GC operates within .NET 8 is depicted in Figure 3.

```
for (int i = 0; i < 32; i++)
{
    new Thread(() =>
    {
        while (true) Array.ForEach(new byte[1], b => { });
    }).Start();
}
using Process process = Process.GetCurrentProcess();

while (true)
{
    process.Refresh();
    Console.WriteLine($"{process.WorkingSet64:N0}");
    Thread.Sleep(1000);
}
```

Figure 3. *Threads and GC usage code*

The example creates a multitude of threads that continuously allocate, and then repeatedly prints out the working set of memory. In Figures 4a, b, c and d we can see the working sets without the GarbageCollectionAdaptationMode property, while in Figure 4e, the property is set and enabled. There we can observe a significant decrease in the working set.

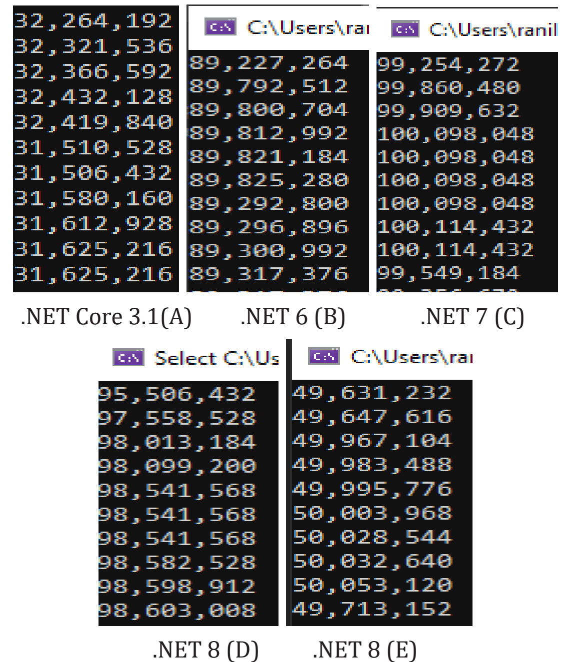


Figure 4. *Result of execute code (GC)*

In conclusion, activating and configuring the GarbageCollectionAdaptationMode property in .NET 8 significantly reduces the working set of memory. This indicates more efficient memory management and potentially better application performance, especially in situations where the number of heaps dynamically adjusts. Therefore, properly configuring the GarbageCollectionAdaptationMode property can contribute to performance optimization and reduce the resources required for application execution.

Upon analysis of the practical work, it becomes evident that .NET 8 introduces numerous performance enhancements, some of which have not been thoroughly explored. Further investigation into additional practical cases not covered in this study reveals significant differences in the utilization of .NET 8 compared to older versions.

E. Limitations of the Study

Scope of Tests: Benchmarking was conducted using a specific set of tests and scenarios. This means that certain edge cases or specific workloads that might impact performance in real-world applications were not covered. The tests focused on general

performance rather than specific scenarios that may have different requirements.

Statistical Analysis: The number of test iterations and variability in the results were controlled, but a small number of iterations can lead to less reliable conclusions. Increasing the number of test iterations could provide more stable and representative performance data.

Update Frequency: The .NET framework is regularly updated, which means the tested versions can quickly become outdated. This can limit the relevance of the results over time, as newer versions with different performance characteristics are released.

F. Upcoming .NET 9 release

Platform for Cloud-Native Development:

.NET 9 will further improve runtime performance and application monitoring, making applications faster and more stable. With .NET Aspire, cloud application development becomes less complex and more cost-effective. Optimizing applications for Native AOT and trimming will reduce application size and improve execution speed.

Tools for Cloud-Native Development:

Visual Studio and Visual Studio Code will gain support for Native AOT, enabling developers to more easily compile and deploy applications across various platforms. Enhanced integration with Azure Container Apps will simplify scaling applications and managing resources in a cloud environment.

Integration with Artificial Intelligence:

.NET 9 will enable developers to more easily integrate AI functionality into their applications through new libraries and documentation for working with OpenAI and OSS models. Collaboration on projects such as Semantic Kernel and Azure SDK will ensure a rich experience in developing intelligent applications.

Backlog and Future Plans:

The .NET team will regularly update the backlog and release notes, introducing new features and optimizations based on feedback from the community and industry partners. Ongoing experiments may become part of future releases, ensuring the platform's continuous evolution.

Impact on the Broader Software Development Landscape:

These improvements will increase developer productivity, enabling faster development and deployment of applications. Enhancements in performance and scalability will result in more efficient and responsive applications, while advanced security solutions will help protect data and ensure regulatory compliance. Broader support for cross-platform development will lower entry barriers and enable more developers to use .NET for developing diverse applications in various environments.

G. Potential for Further Research

Long-term Performance: Additional research that includes long-term performance and stability of different .NET framework versions could provide deeper insights into the efficiency and reliability of the platform.

Real-world Scenarios: Including case studies from the real world and various industrial applications can help understand how specific features and improvements impact performance in real conditions.

Comparative Analysis: Further research that includes comparisons with other development platforms and languages can provide a broader context and help assess the relative advantages and disadvantages of the .NET ecosystem.

Conclusion

The journey from .NET Core to .NET 8 represents a remarkable chapter in the evolution of the .NET ecosystem, marked by significant advancements in performance, a broadening of features, and the simplification of migration pathways for developers. Through the analysis of performance improvements, feature enhancements, and migration considerations, this paper has illuminated the strategic and technological evolution that has taken place within the .NET framework, culminating in the release of .NET 8.

.NET Core 3.1 laid the groundwork with its robust foundation, setting a high standard for performance and reliability. From there, each subsequent version of .NET has introduced enhancements and new capabilities, demonstrating Microsoft's commitment to innovation and its responsiveness to the needs of the development community. The integration of AI and machine learning capabilities in .NET 8 is particularly

notable, signaling a forward-looking approach to application development.

Looking ahead, the upcoming release of .NET 9 promises to continue this trajectory of innovation. With the .NET ecosystem now firmly established as a unified platform, future versions are poised to delve deeper into the realms of AI, machine learning, and other cutting-edge technologies. The emphasis on performance optimization, feature richness, and ease of migration will undoubtedly remain central, ensuring that .NET continues to be a leading framework for developers worldwide.

In conclusion, the transition from .NET Core to .NET 8 and beyond exemplifies the dynamic and evolving nature of the .NET ecosystem. This evolution reflects a broader trend in software development towards more efficient, flexible, and intelligent solutions. As the .NET framework continues to advance, it will undoubtedly play a pivotal role in shaping the future of software development, empowering developers to create innovative applications that address the complex challenges of the digital age.

References

[1] Microsoft. (2016). Introducing .NET Core. .NET Blog. Accessed April 15, 2024. [Online]. Available: https://devblogs.microsoft.com/dotnet/introducing-net-core/

[2] Microsoft. (2019). "Announcing .NET Core 3.1". Accessed April 18, 2024. [Online]. Available: https://devblogs.microsoft.com/dotnet/announcing-net-core-3-1/.

[3] Microsoft. (2019). "Performance improvements in .NET Core 3.1". Accessed April 18, 2024. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-1.

[4] Microsoft. (2020). "The journey to one .NET". Accessed April 18, 2024. [Online]. Available: https://devblogs.microsoft.com/dotnet/the-journey-to-one-net/.

[5] Microsoft. (2022). "Machine Learning with ML.NET". Accessed April 20, 2024. [Online]. Available: https://dotnet.microsoft.com/en-us/apps/machinelearning-ai/ml-dotnet.

[6] Microsoft. (2021). "Performance improvements in .NET 6". Accessed April 18, 2024. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-6.

[7] Microsoft. (2021). "Performance best practices with ASP.NET Core". Accessed April 18, 2024. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/performance/performance-best-practices.

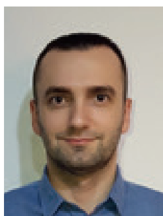
[8] BenchmarkDotNet. "Home - BenchmarkDotNet Documentation." BenchmarkDotNet, Accessed March 24, 2024. [Online]. Available: https://benchmarkdotnet.org/.

[9] Akinshin, A. (2018). "Pro .NET Benchmarking: The Art of Performance Measurement." Apress.

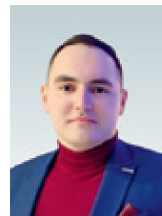
[10] Roth, D., & Price, M. J. (2018). "Migrating to .NET Core: Rebuilding Enterprise Applications." Apress.

Received: May 13, 2024
Accepted: May 24, 2024

About the authors



Branimir Cvijić received the B.Sc. and M.Sc. degrees in computer engineering and informatics from the University of Banja Luka (Banja Luka, Bosnia and Herzegovina) in 2009 and 2014, respectively. He worked in Lanaco d.o.o, Banja Luka, as senior software architect from 2009 to 2021. Since 2021 works in Endava as Software development consultant - Development lead. His specialisation is in software and databases development using enterprise tools. His research interests are Internet of things using enterprise tools for development and integration in different areas of industry.



Pero Ranilović was born in the city of Prijedor (Republic of Srpska, BiH). Graduated from high school in Novi Grad. Bachelor's degree in Programming and Software Engineering earned at the Faculty of Information Technology at Pan-European University "APEIRON" in Banja Luka.

Master's degree pursued at the Faculty of Information Technology at Pan-European University "APEIRON" in Banja Luka. Since 2018, he has been employed as a software developer at Lanaco. Employed as a software developer in Lanaco, since 2018. In addition, hired as a teaching assistant at Pan-European University "APEIRON".

For citation

Branimir Cvijić, Pero Ranilović, From .NET Core to .NET 8: A Comprehensive Analysis of Performance, Features, and Migration Pathways, *JITA – Journal of Information Technology and Applications, Banja Luka*, Pan-Europien University APEIRON, Banja Luka, Republika Srpska, Bosna i Hercegovina, JITA 14(2024)1:69-77, (UDC: 004.432.2C#:004.738-.1), (DOI: 10.7251/JIT2401069C, Volume 14, Number 1, Banja Luka, June (1-88), ISSN 2232-9625 (print), ISSN 2233-0194 (online), UDC 004