



DOI 10.28925/2663-4023.2026.35.1323

УДК 004.415.53:004.422

Третяк Аліна Петрівна

здобувачка вищої освіти,
Волинський національний університет імені Лесі Українки, Луцьк, Україна
ORCID: 0000-0002-1776-5339
Tretyak.Alina@vnu.edu.ua

Булатецька Леся Віталіївна

кандидат фіз.-мат. наук, доцент,
доцент кафедри комп'ютерних наук та кібербезпеки
Волинський національний університет імені Лесі Українки, Луцьк, Україна
ORCID: 0000-0002-7202-826X
Bulatetska.Lesya@vnu.edu.ua

Булатецький Віталій Вікторович

кандидат фіз.-мат. наук, доцент,
доцент кафедри комп'ютерних наук та кібербезпеки
Волинський національний університет імені Лесі Українки, Луцьк, Україна
ORCID: 0000-0002-9883-4550
Bulatetsky.Vitaly@vnu.edu.ua

Павленко Юлія Степанівна

старший викладач кафедри комп'ютерних наук та кібербезпеки
Волинський національний університет імені Лесі Українки, Луцьк, Україна
ORCID: 0000-0002-4065-045X
Pavlenko.Yulya@vnu.edu.ua

ОЦІНКА ВПЛИВУ JIT ТА NATIVE AOT-КОМПІЛЯЦІЇ НА ПРОДУКТИВНІСТЬ І РЕСУРСОЕФЕКТИВНІСТЬ ВЕБСЕРВІСІВ У СЕРЕДОВИЩІ .NET 9

Анотація. У статті представлено результати дослідження впливу методів компіляції Just-In-Time (JIT) та Native Ahead-Of-Time (Native AOT) на продуктивність і ресурсоефективність вебсервісів у середовищі .NET 9. Актуальність роботи зумовлена активним використанням контейнеризованих і безсерверних архітектур, у яких важливими характеристиками є скорочення часу ініціалізації, підвищення пропускну здатності та оптимізація використання системних ресурсів. Експериментальне дослідження виконано на основі тестового вебсервісу, розробленого з використанням платформи ASP.NET Core та полегшеного шаблону `WebApplication.CreateSlimBuilder()`. Для мінімізації впливу сторонніх компонентів доступ до локальної бази даних SQLite організовано без використання ORM через ADO.NET, а серіалізацію даних реалізовано засобами `System.Text.Json`. Порівняння виконано за сценаріями точкового читання даних за первинним ключем і пакетної вибірки 10^3 записів із подальшою JSON-серіалізацією. Навантажувальне тестування проводилося в діапазоні від 1 до 200 паралельних потоків. За результатами експериментів встановлено, що Native AOT забезпечує суттєву перевагу під час «холодного» запуску вебсервісу. Час обробки першого запиту в сценарії точкового читання скорочено з приблизно 668 мс для JIT-версії до 44 мс для Native AOT, що відповідає майже 15-кратному зменшенню затримки. Дослідження пропускну здатності показало, що за навантаження 4 паралельними потоками Native AOT забезпечує продуктивність близько 667 зап./с проти 626 зап./с у JIT-версії. Аналіз використання оперативної пам'яті продемонстрував близький рівень споживання ресурсів у стабілізованому стані: близько 26.13 МБ для JIT та 26.55 МБ для Native AOT. Отримані результати можуть бути використані для обґрунтованого вибору способу компіляції під час проєктування сучасних високонавантажених вебсервісів.

Ключові слова: .NET 9, вебсервіс, база даних SQLite, операційна система, JIT-компіляція, Native AOT, ASP.NET Core, пропускна здатність, затримка відгуку, оперативна пам'ять, IL Trimming.



ВСТУП

Постановка проблеми. Тенденції розвитку хмарних обчислень зумовлюють перехід від монолітних архітектур до мікросервісів, контейнеризації та безсерверних обчислень. Тому до вебсервісів висуваються вимоги не лише щодо високої пропускної здатності під стаціонарним навантаженням, але й до споживання обчислювальних ресурсів процесу та мінімального часу ініціалізації (затримки першого відгуку).

Платформа .NET зазвичай виконує код у два кроки. Спочатку програма перетворюється на проміжний IL-код (Intermediate Language, або проміжна мова), а вже під час запуску JIT-компілятор (Just-In-Time) «на льоту» перекладає його в машинні команди. Така динамічна компіляція дозволяє програмі працювати дуже швидко після розігріву. Проте при цьому система витрачає додатковий час і оперативну пам'ять на запуск віртуальної машини та налаштування внутрішніх процесів .NET.

Вихід платформи .NET 9 відкрив ширші можливості для застосування технології Native AOT (Ahead-Of-Time), яка передбачає повну статичну компіляцію високорівневого IL-коду безпосередньо в нативний машинний код цільової архітектури операційної системи без використання класичного середовища виконання .NET.

Перехід від виконання застосунку в середовищі CLR до використання автономного виконуваного файлу, сформованого засобами Native AOT, змінює принципи функціонування програмної системи та потребує оцінки наслідків таких змін. Зокрема, важливо визначити, як статичний аналіз коду, механізм видалення невикористовуваних компонентів (IL Trimming) і відсутність JIT-компіляції впливають на продуктивність вебсервісу за інтенсивного навантаження, взаємодію з підсистемою введення-виведення бази даних, а також ефективність використання ресурсів операційної системи.

Аналіз останніх досліджень і публікацій. Практичний інтерес до компіляції з випередженням часу суттєво зріс із появою офіційної підтримки Native AOT у сучасних вебфреймворках, зокрема в ASP.NET Core [1], де цей підхід позиціонується як ключовий засіб мінімізації часу запуску та зменшення обсягу споживання пам'яті. Водночас емпіричні дослідження підтверджують відмінності в характеристиках продуктивності Native AOT та JIT-компіляції. Зокрема, у роботі [2] на прикладі JavaScript показано, що хоча JIT-двигуни перевершують Native AOT у стаціонарному режимі на стандартних контрольних тестах, статична компіляція є цілком конкурентоспроможною. На розширеній вибірці тестів застосунки, скомпільовані в режимі Native AOT, не поступалися JIT-версіям за швидкістю, забезпечуючи при цьому вищу швидкість запуску та суттєво менше споживання оперативної пам'яті.

Автори [3] виконали подібне порівняння для середовища виконання Python, підкресливши відмінності в енергоефективності та системній ресурсоемності. Експериментальний аналіз восьми спеціалізованих компіляторів (зокрема Codon, PyPy та Numba) показав, що перехід від класичного інтерпретатора до компіляційних підходів дозволяє досягти понад 90 % приросту швидкості обчислень і відповідного скорочення енерговитрат, підтверджуючи вирішальну роль компілятора у забезпеченні обчислювальної ефективності програмних систем.

Окрему увагу в наукових працях приділено проблемам формування компактних нативних дистрибутивів. Автори роботи [4] досліджували шляхи оптимізації Native AOT-компіляції у середовищі GraalVM Native Image, запропонувавши гібридний підхід до формування нативних виконуваних файлів. Для вирішення проблеми надмірного розміру результуючих бінарних файлів автори запропонували виключати з монолітної компіляції методи, які рідко використовуються, залишаючи їх у формі байт-коду для виконання легковаговим інтерпретатором. Експерименти довели, що таке рішення дозволяє скоротити розмір нативного файлу на величину до 36.9 % без втрати пікової продуктивності та без погіршення швидкості запуску застосунку [4].

У роботі [5] проведено емпіричне дослідження компромісів між AOT- та JIT-компіляцією. Автори показали, що AOT забезпечує майже миттєвий запуск окремих тестів (від 0.015 мс) та зменшує споживання оперативної пам'яті на 18—50 % порівняно з JIT. Водночас JIT-версії потрібно від 3 до 6 ітерацій для «розігріву», після чого завдяки адаптивній оптимізації під час виконання вона досягає продуктивності AOT або навіть перевищує її. Автори зазначають, що жоден із підходів не є універсальним, і рекомендують використовувати гібридні стратегії, які поєднують швидкий запуск AOT та високу адаптивну продуктивність JIT під час тривалої роботи.

Важливим аспектом оцінки продуктивності є методологія вимірювань та врахування фази «розігріву». У роботі [6] автори проаналізували специфіку мікротестування продуктивності Java-застосунків, зосередившись на динаміці стаціонарного стану виконання. Автори показали, що фаза «розігріву» через оптимізації віртуальної машини супроводжується значними коливаннями швидкодії, причому значна частина тестувань узагалі не досягає стабільного стану. Дослідження підкреслює, що суб'єктивні оцінки тривалості ініціалізації розробниками часто є не точними, що призводить до



викривлення результатів. Це доводить необхідність враховувати часову нестабільність та застосовувати автоматизовані методи контролю для об'єктивної оцінки продуктивності середовищ виконання.

Результати аналізу сучасних досліджень показують відсутність універсального рішення у виборі моделі компіляції. АОТ-компіляція забезпечує високу передбачуваність виконання та ефективніше використовує системні ресурси, тоді як JIT-компіляція має перевагу завдяки можливості динамічно оптимізувати код під час виконання. Поява .NET 9 із розширеними можливостями Native AOT зумовила необхідність комплексного порівняння різних моделей компіляції та виконання застосунків у межах цієї платформи.

Метою статті є порівняльне оцінювання впливу JIT- та Native AOT-компіляції на продуктивність і ефективність використання системних ресурсів вебсервісів у середовищі .NET 9.

МЕТОДИКА ДОСЛІДЖЕННЯ

Експериментальне дослідження виконували на базі робочої станції Lenovo ThinkPad T450, оснащеної процесором Intel Core i5-5300U, 8 ГБ оперативної пам'яті (RAM) та накопичувачем SSD (dual level cell, INTEL SSDSC2BF180A5L, об'ємом 180 ГБ, заповненого на 60 %, із інтерфейсом SATA-3), під керуванням 64-розрядної операційної системи MS Windows 10 Pro.

Для оцінювання впливу підходів до виконання коду JIT та Native AOT було розроблено тестовий вебсервіс на базі ASP.NET Core. У якості програмного середовища виконання та розгортання використано платформу .NET 9 (x64). З метою усунення впливу накладних витрат, пов'язаних із використанням засобів об'єктно-реляційного відображення (ORM), експериментальний вебсервіс реалізовано на основі полегшеного шаблону WebApplication.CreateSlimBuilder(). Взаємодія з базою даних організована через ADO.NET (Microsoft.Data.Sqlite) без застосування ORM, а серіалізацію об'єктів виконано засобами System.Text.Json із використанням генераторів вихідного коду.

Для зберігання експериментальних даних використовувалася локальна база даних SQLite. Після створення база автоматично заповнювалася 10^4 тестовими записами, що забезпечило однакові початкові умови для всіх експериментів. Основу структури бази даних становила таблиця Data:

```
CREATE TABLE IF NOT EXISTS Data (  
    Id INTEGER PRIMARY KEY AUTOINCREMENT,  
    Name TEXT NOT NULL,  
    Value REAL NOT NULL,  
    CreatedAt TEXT NOT NULL  
);
```

Тестування продуктивності охоплювало два сценарії роботи з базою даних.

Перший сценарій моделював читання окремого запису за первинним ключем. Для цього виконувався параметризований SQL-запит через вебмаршрут `/api/data/single/1`. Кожен експеримент складався з серії 10^4 послідовних HTTP-запитів.

```
SELECT Id, Title, Value, CreatedAt  
FROM Data  
WHERE Id = @id;
```

Другий сценарій моделював пакетне читання даних. Через вебмаршрут `/api/data/batch` виконувалася вибірка 10^4 записів одним SQL-запитом із подальшою їх серіалізацією у формат JSON та передаванням клієнту. Кожен експеримент включав серію з 500 HTTP-запитів.

```
SELECT Id, Title, Value, CreatedAt  
FROM Data LIMIT 1000;
```

Навантажувальне тестування було здійснене за допомогою спеціалізованого скрипту PowerShell із викликом паралельних асинхронних завдань. Для оцінки стабільності та масштабованості системи обидва сценарії моделювалися за таких рівнів навантаження: 1, 2, 3, 4, 5, 10, 50, 100 та 200 паралельних потоків. Основними показниками оцінювання були пропускна здатність системи (RPS, Requests Per Second), середній час обробки запитів, обсяг використаної оперативної пам'яті за показником робочого набору процесу (Working Set), а також час «холодного» запуску сервісу та отримання першого відгуку (Cold Start Latency).



РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Аналіз параметрів компіляції та загальний розмір файлів збірки. На початковому етапі експерименту вебсервіс зібрали у двох режимах: із динамічною JIT-компіляцією та Native AOT. Час побудови Native AOT-версії становив близько 60 секунд, тоді як для JIT-версії не перевищував 2 секунд. Більший час компіляції Native AOT обумовлений тим, що система виконує статичний аналіз залежностей, видаляє невикористовуваний код (IL Trimming) та генерує нативний машинний код безпосередньо під час збірки, а не під час запуску. [7]

Після компіляції було визначено підсумковий розмір пакетів розгортання та основних виконуваних файлів (табл. 1). Розмір файлу WebDatabaseTest.exe для JIT-версії становив 0.15 МБ, тоді як нативний виконуваний файл, сформований за допомогою Native AOT, мав розмір 9.20 МБ. Така відмінність зумовлена принциповими особливостями архітектури виконання. У JIT-варіанті виконуваний файл містить мінімальний код запуску, тоді як основна частина логіки представлена у вигляді проміжного коду (IL) та залежних бібліотек (Microsoft.Data.Sqlite.dll, SQLitePCLRaw тощо), виконання яких забезпечується середовищем .NET CoreCLR. На відміну від цього, Native AOT формує автономний виконуваний файл, що містить попередньо скомпільований машинний код і необхідні компоненти середовища виконання для незалежного запуску застосунку. [7]

Таблиця 1.

Порівняльна характеристика параметрів та загального розміру файлів збірки

Збірка (Configuration)	Час компіляції (сек)	Розмір .exe файлів (МБ)	Загальний розмір папки без .PDB (МБ)	Архітектурна особливість
JIT (publish-jit)	~2	0.15	2.35	Залежить від встановленого середовища
Native AOT (publish-aot)	~60	9.20	10.88	Повністю автономний бінарний файл

Для коректного порівняння розмірів опублікованих застосунків було визначено загальний обсяг пакетів публікації без урахування налагоджувальних символів (.pdb). Файл WebDatabaseTest.pdb обсягом 47.45 МБ не враховувався під час вимірювання, оскільки такі файли містять інформацію для налагодження та зазвичай не включаються до пакета розгортання застосунку у виробничому середовищі. Загальний розмір папки JIT-збірки становив 2.35 МБ за умови конфігурації із залежністю від середовища виконання, коли застосунок використовує попередньо встановлене середовище .NET 9 Runtime у цільовій системі. Натомість, Native AOT формує самодостатній пакет розгортання загальним обсягом 10.88 МБ, який містить скомпільований машинний код, статично скомпоновані бібліотеки та необхідні компоненти середовища виконання для автономного запуску застосунку. [7],[8]

У випадку використання JIT-моделі у режимі автономного розгортання (self-contained deployment) до пакета застосунку включаються необхідні компоненти .NET Runtime, що призводить до збільшення його загального розміру порівняно з варіантом, залежним від встановленого середовища виконання [8].

Згідно з документацією [7], [8], [9], Native AOT виконує попередню компіляцію коду застосунку в нативний машинний код, використовуючи статичний аналіз залежностей та механізми скорочення невикористовуваного коду (trimming), що дозволяє сформувати автономний виконуваний файл без необхідності JIT-компіляції під час виконання. У проведеному експерименті такий підхід дозволив сформувати автономний пакет розгортання обсягом 10.88 МБ, який може виконуватися без попереднього встановлення середовища виконання .NET.

Оцінка часу первинної ініціалізації та «холодного» старту. Окрім розмірів пакетів публікації, важливою характеристикою мікросервісної архітектури є час первинного відгуку вебсервісу після його запуску. Для вимірювання часу «холодного» старту було проведено серію з 5 незалежних перезапусків процесів вебсервісу з вимірюванням часу обробки першого HTTP-запиту, а також двох наступних послідовних запитів для обох досліджуваних сценаріїв запитів. Зведені результати наведено на рис. 1.

Аналіз даних, наведених на рис. 1, демонструє перевагу технології Native AOT у сценарії «холодного» запуску вебсервісу. Для операції точкового читання використання Native AOT дозволило скоротити час первинного відгуку з ~668 мс до ~44 мс, що відповідає приблизно 15-кратному скороченню часу обробки першого запиту порівняно з JIT-версією. Така перевага зумовлена відсутністю необхідності виконання JIT-компіляції проміжного коду (IL) під час запуску та скороченням кількості етапів ініціалізації середовища виконання .NET.[7]

У сценарії пакетного читання перевага Native AOT при першому запуску знижується до ~30 % (~260 мс проти ~374 мс у JIT). Таке зближення показників може бути обумовлене тривалими операціями дискового введення-виведення та JSON-серіалізацією 10^4 записів.



Рис. 1. Порівняльна характеристика часу відгуку при «холодному» та «гарячому» старті (мс)

Під час обробки другого та третього запитів (у «гарячому» стані) значення затримок обох конфігурацій стають близькими. Для сценарію точкового читання вони становлять ~21—29 мс, а для пакетного читання – ~215—234 мс, що свідчить про зменшення впливу початкової ініціалізації та вирівнювання продуктивності систем після прогрівання середовища виконання.

Дослідження пропускної здатності та затримок обробки запитів. За результатами серії тестувань (з підтвердженою повторюваністю експерименту та похибкою вимірювань < 4 %) було отримано усереднені показники пропускної здатності системи, середньої затримки обробки та динаміку споживання оперативної пам'яті за показником робочого набору.

Характер залежності пропускної здатності вебсервісу (RPS) від рівня конкурентного навантаження детально відображено далі на рис. 2 (у логарифмічному масштабі).

Характер кривих на рис. 2 дозволяє визначити дві фази поведінки системи. У першій фазі активного масштабування (з кількістю паралельних потоків від 1 до 4) спостерігається стрімкий підйом пропускної здатності від послідовного виконання (1 потік) до максимуму при 4 паралельних потоках. Для точкового читання даних значення пропускної здатності зростає з ~311—335 зап./с до пікових ~667 зап./с у режимі Native AOT та ~626 зап./с для JIT [5]. Для пакетного читання показник підвищується з 177—182 зап./с до пікових 370—379 зап./с. Це, імовірно, зумовлено повним використанням апаратного потенціалу CPU (2 фізичних ядра / 4 логічні потоки) без виникнення затримок у черзі запитів [10].

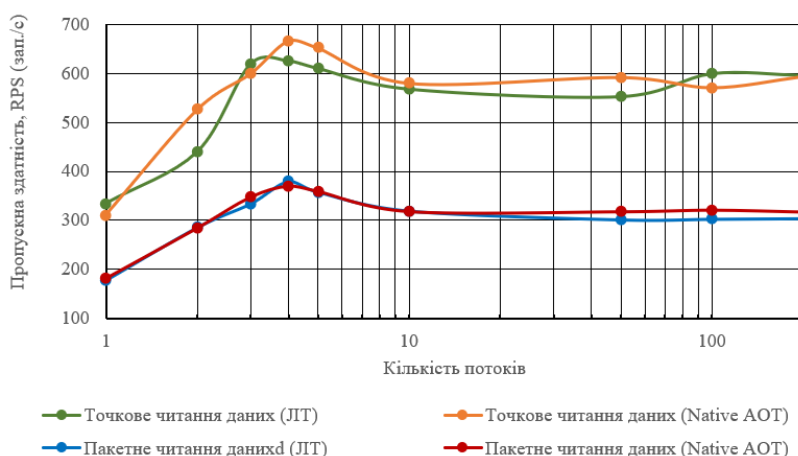


Рис. 2. Залежність пропускної здатності (RPS, зап./с) вебсервісу від кількості паралельних потоків у логарифмічному масштабі



Друга фаза – це фаза насичення, що охоплює рівні від 5 до 100 паралельних потоків. Оскільки кількість конкуруючих потоків запитів перевищує число логічних ядер CPU, виникають накладні витрати на перемикання контексту в операційній системі. Через це пропускна здатність стабілізується на рівні ~570—600 зап./с для точкових та ~300—320 зап./с для пакетних операцій. Графік у цьому інтервалі стає майже горизонтальним і зберігає таку форму аж до максимального досліджуваного навантаження у 200 паралельних потоків. Це свідчить про високу стійкість системи управління потоками, відсутність взаємних блокувань та раціональне використання ресурсів [11].

Динаміку середньої затримки обробки запитів відображено на рис. 3. Цей показник відображає сукупні накладні витрати на мережеву взаємодію, затримки в системі управління потоками, виконання алгоритмів бізнес-логіки та дискові операції введення-виведення (I/O).

Для сценарію точкового читання даних затримка залишається мінімальною та майже незмінною протягом усього експерименту (від ~0.7 мс при 1 потоці до не більше ~1.8—2.2 мс при 200 потоках). У сценарії пакетного читання даних зростання затримки з ~3.2 мс до ~8.8—9.0 мс має плавний характер і зумовлене обсягом серіалізації 10^3 об'єктів БД під час формування JSON-відповіді [12]. В сценарії точкового читання режим Native AOT забезпечує вищу пропускну здатність у точці максимального навантаження (4 потоки), перевищуючи JIT приблизно на 6 %, а також демонструє дещо меншу затримку в умовах високого навантаження.

На рис. 4 наведено порівняльний аналіз пропускної здатності вебсервісу для конфігурацій JIT та Native AOT для трьох характерних режимів навантаження: послідовного виконання (1 потік), режиму максимальної пропускної здатності (4 паралельні потоки) та режиму насичення (100 паралельних потоків). Під час виконання операції точкового читання за навантаження 4 паралельними потоками конфігурація Native AOT забезпечує максимальну пропускну здатність на рівні ~667 зап./с. Це приблизно на 6 % перевищує результат JIT-конфігурації (~626 зап./с). Такий приріст може бути зумовлений відсутністю необхідності виконання JIT-компіляції під час роботи застосунку, що дає змогу ефективніше використовувати процесорний час при обробці короткотривалих запитів. [5], [7]

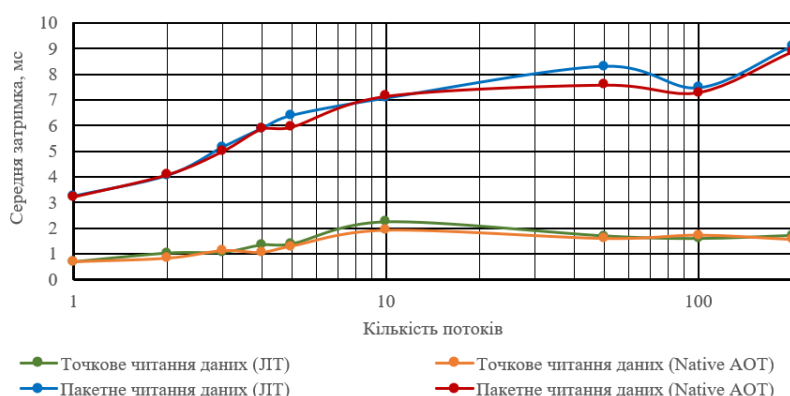


Рис. 3. Залежність середньої затримки обробки запитів від кількості паралельних потоків

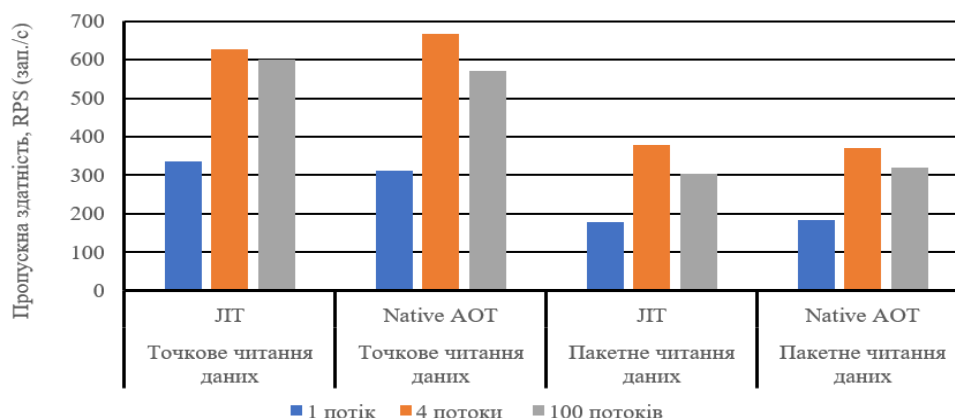


Рис. 4. Порівняння пропускної здатності (RPS, запитів/с) режимів JIT та Native AOT для точкового та пакетного читання даних при 1, 4 та 100 паралельних потоках



У сценарії пакетного читання даних обидва режими компіляції демонструють практично однаковий рівень пропускної здатності. За навантаження 4 паралельними потоками JIT забезпечує ~380 зап./с проти ~370 зап./с у Native AOT, тоді як у режимі насичення (100 потоків) Native AOT досягає ~321 зап./с порівняно з ~302 зап./с у JIT. В обох випадках різниця не перевищує 5 %. Отримані результати можна пояснити тим, що основний внесок у час обробки запиту формують операції доступу до бази даних SQLite та JSON-серіалізації великого обсягу даних, тоді як вплив способу компіляції на загальну продуктивність стає менш вираженим.

Аналіз динаміки споживання оперативної пам'яті. Вимірювання робочого набору оперативної пам'яті проводилося на трьох ключових етапах життєвого циклу процесів: у режимі очікування перед початком випробувань, під час максимального навантаження та у фазі повної релаксації системи після завершення тестування. Отримані кількісні показники наведено в табл. 2.

Таблиця 2.

Динаміка використання оперативної пам'яті (WorkingSet / RAM)

Етап вимірювання	JIT (publish-jit)	Native AOT (publish-aot)
1. До тесту (чистий запуск)	~22.90 МБ	~22.85 МБ
2. Під час пікового навантаження	~34.09 МБ	~34.89 МБ
3. Повна релаксація (після тесту)	~26.13 МБ	~26.55 МБ

Після завершення навантажувального тестування споживання оперативної пам'яті повертається майже до початкового рівня, що не вказує на наявність витоків пам'яті. Зростання обсягу робочого набору пам'яті з ~22.9 МБ до ~34.5 МБ під час пакетного читання пояснюється короточасним виділенням ресурсів під буфери JSON-серіалізації та HTTP-контексти. Після завершення тестування обсяг споживання оперативної пам'яті знижується до ~26.3 МБ, що свідчить про ефективне вивільнення тимчасово виділених ресурсів.

Така динаміка не виявила ознак неефективного керування пам'яттю та узгоджується з коректною роботою механізму автоматичного вивільнення невикористовуваних об'єктів. У стабілізованому стані різниця в обсязі використаної оперативної пам'яті між JIT-версією (26.13 МБ) та Native AOT (26.55 МБ) становить лише 0.42 МБ (менше 1.6 %), що є незначною. Це свідчить про подібний рівень використання оперативної пам'яті обома конфігураціями.

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

У результаті проведеного експериментального дослідження здійснено порівняльне оцінювання впливу JIT- та Native AOT-компіляції на продуктивність і ресурсоемність вебсервісів у середовищі .NET 9 під керуванням ОС MS Windows 10 x64. Встановлено, що головною перевагою Native AOT є майже 15-кратне скорочення часу «холодного» старту (з ~668 мс до ~44 мс) за рахунок відсутності накладних витрат на ініціалізацію CLR та динамічну компіляцію IL-коду. У режимі максимальної пропускної здатності (4 паралельні потоки) Native AOT перевищує JIT на ~6 % (~667 проти ~626 зап./сек), однак у фазі насичення та в сценаріях із переважанням операцій введення-виведення й серіалізації показники обох режимів вирівнюються, демонструючи відхилення не більше 5 %. Водночас обидва підходи виявилися однаково ресурсоефективними щодо використання оперативної пам'яті у стабілізованому стані (~26.13 МБ у JIT проти ~26.55 МБ у Native AOT). Попри більший обсяг автономного пакета розгортання (10.88 МБ проти 2.35 МБ у залежної JIT-збірки), Native AOT є оптимальним вибором для мікросервісів та контейнеризованих середовищ завдяки відсутності зовнішніх залежностей і миттєвому першому відгуку, тоді як JIT залишається ефективним для монолітних довготривалих систем. Перспективи подальших досліджень полягають у проведенні порівняльних тестів на кросплатформених ОС (Linux/Alpine), оцінці роботи сервісу під високим мережевим навантаженням та аналізі ефективності гібридних методів оптимізації коду.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft. (2026). *ASP.NET Core support for Native AOT*. Microsoft Learn. <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/native-aot?view=aspnetcore-10.0>
2. Serrano, M. (2021). Of JavaScript AOT compilation performance. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 1–30. <https://doi.org/10.1145/3473575>



3. Stoico, V., Dragomir, A. C., & Lago, P. (2025). *An empirical study on the performance and energy usage of compiled Python code* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2505.02346>
4. Pichler, C., Bohn, M., Mössenböck, H., & Wimmer, C. (2025). Fast and compact: Reducing size of AOT-compiled Java code without sacrificing performance. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25)* (pp. 12–22). Association for Computing Machinery. <https://doi.org/10.1145/3759426.3760976>
5. Ugwumba, N. K., & Jaja, P. S. (2025). *Ahead-of-Time vs. Just-in-Time Compilation Trade-offs: Empirical performance studies* [Preprint]. Research Square. <https://doi.org/10.21203/rs.3.rs-7915532/v1>
6. Traini, L., Cortellessa, V., Di Pompeo, D., & Tucci, M. (2022). Towards effective assessment of steady state performance in Java software: Are we there yet? *Empirical Software Engineering*, 28(1), Article 13. <https://doi.org/10.1007/s10664-022-10247-x>
7. Microsoft. (2026, January 8). *Native AOT deployment overview*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/>
8. Microsoft. (2025, October 28). *.NET application publishing overview*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/core/deploying/>
9. TeamDev. (2026). *Trimming & Native AOT*. DotNetBrowser Documentation. <https://teamdev.com/dotnetbrowser/docs/guides/gs/aot-support/>
10. Kokosa, K. (2018). *Pro .NET Memory Management: For Better Code, Performance, and Scalability*. Apress. <https://doi.org/10.1007/978-1-4842-4027-4>
11. Microsoft. (2024). *The managed thread pool*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/standard/threading/the-managed-thread-pool>
12. Microsoft. (2025) How to serialize JSON in C# - .NET. Microsoft Learn: Build with answers in reach. <https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to>

**Alina P. Tretyak**

Higher Education Applicant,
Lesya Ukrainka Volyn National University, Lutsk, Ukraine
ORCID: 0000-0002-1776-5339
Tretyak.Alina@vnu.edu.ua

Lesia V. Bulatetska

PhD, Associate Professor, Associate Professor at the Department of Computer Science and Cybersecurity
Lesya Ukrainka Volyn National University, Lutsk, Ukraine
ORCID: 0000-0002-7202-826X
Bulatetska.Lesya@vnu.edu.ua

Vitalii V. Bulatetskyi

PhD, Associate Professor, Associate Professor at the Department of Computer Science and Cybersecurity,
Lesya Ukrainka Volyn National University, 13 Volia Avenue, Lutsk, 43025, Ukraine,
ORCID: 0000-0002-9883-4550
Bulatetsky.Vitaly@vnu.edu.ua

Yuliia S. Pavlenko

Senior Lecturer of the Department of Computer Science and Cybersecurity
Lesya Ukrainka Volyn National University, Lutsk, Ukraine
ORCID: 0000-0002-4065-045X
Pavlenko.Yulya@vnu.edu.ua

EVALUATION OF THE IMPACT OF JIT AND NATIVE AOT COMPILATION ON THE PERFORMANCE AND RESOURCE EFFICIENCY OF WEB SERVICES IN THE .NET 9 ENVIRONMENT

Abstract. This paper presents the findings of a study evaluating the impact of Just-In-Time (JIT) and Native Ahead-Of-Time (Native AOT) compilation techniques on the performance and resource efficiency of web services running in the .NET 9 environment. The relevance of the research is driven by the widespread adoption of containerized and serverless architectures, where reducing initialization time, increasing throughput, and optimizing system resource usage are critical requirements. The experimental study was conducted using a test web service developed with the ASP.NET Core platform and the lightweight WebApplication.CreateSlimBuilder() template. To minimize the overhead of third-party components, access to a local SQLite database was implemented via ADO.NET without using an ORM, and data serialization was handled using native System.Text.Json. The comparison evaluated two scenarios: single-record lookup by primary key and batch retrieval of 103 records followed by JSON serialization. Load testing was performed across a concurrency range of 1 to 200 parallel threads. The experimental results indicate that Native AOT provides a significant advantage during the cold start of the web service. The processing time for the initial request in the single-record lookup scenario dropped from approximately 668 ms for the JIT version to 44 ms for Native AOT, representing an nearly 15-fold decrease in latency. Throughput testing demonstrated that under a load of 4 parallel threads, Native AOT achieved approximately 667 req/s compared to 626 req/s for the JIT version. Memory analysis revealed comparable resource consumption in a stabilized state, with approximately 26.13 MB for JIT and 26.55 MB for Native AOT. The findings can be applied to inform the selection of compilation strategies when designing modern high-load web services.

Keywords: .NET 9, web service, SQLite database, operating system, JIT compilation, Native AOT, ASP.NET Core, throughput, response latency, RAM, IL Trimming.

REFERENCES

1. Microsoft. (2026). *ASP.NET Core support for Native AOT*. Microsoft Learn. <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/native-aot?view=aspnetcore-10.0>
2. Serrano, M. (2021). Of JavaScript AOT compilation performance. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 1–30. <https://doi.org/10.1145/3473575>



3. Stoico, V., Dragomir, A. C., & Lago, P. (2025). *An empirical study on the performance and energy usage of compiled Python code* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2505.02346>
4. Pichler, C., Bohn, M., Mössenböck, H., & Wimmer, C. (2025). Fast and compact: Reducing size of AOT-compiled Java code without sacrificing performance. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25)* (pp. 12–22). Association for Computing Machinery. <https://doi.org/10.1145/3759426.3760976>
5. Ugwumba, N. K., & Jaja, P. S. (2025). *Ahead-of-Time vs. Just-in-Time Compilation Trade-offs: Empirical performance studies* [Preprint]. Research Square. <https://doi.org/10.21203/rs.3.rs-7915532/v1>
6. Traini, L., Cortellessa, V., Di Pompeo, D., & Tucci, M. (2022). Towards effective assessment of steady state performance in Java software: Are we there yet? *Empirical Software Engineering*, 28(1), Article 13. <https://doi.org/10.1007/s10664-022-10247-x>
7. Microsoft. (2026, January 8). *Native AOT deployment overview*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/>
8. Microsoft. (2025, October 28). *.NET application publishing overview*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/core/deploying/>
9. TeamDev. (2026). *Trimming & Native AOT*. DotNetBrowser Documentation. <https://teamdev.com/dotnetbrowser/docs/guides/gs/aot-support/>
10. Kokosa, K. (2018). *Pro .NET Memory Management: For Better Code, Performance, and Scalability*. Apress. <https://doi.org/10.1007/978-1-4842-4027-4>
11. Microsoft. (2024). *The managed thread pool*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/standard/threading/the-managed-thread-pool>
12. Microsoft. (2025) How to serialize JSON in C# - .NET. *Microsoft Learn: Build with answers in reach*. <https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to>

Отримано редакцією журналу / Received: 03.06.26

Прорецензовано / Revised: 16.06.26

Схвалено до друку / Accepted: 26.10.26



This work is licensed under Creative Commons Attribution-noncommercial-sharelike 4.0 International License.