Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)
Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів продуктивності .NET застосунків залежно від вибору контейнера Dependency Injection
(тема)

Виконав:
Здобувач 2 року навчання
групи ІПЗм-23-1
Іван КОРОБОВ
(власне ім'я, прізвище)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова
Освітня програма Інженерія програмного забезпечення
(повна назва освітньої програми)
Керівник доц. Олексій НАЗАРОВ
(посада, власне ім'я, прізвище)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (власне ім'я, прізвище)

2025

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма
(освітньо-професійна або освітньо-наукова)

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___»_____ 2025р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Коробову Івану Руслановичу
(прізвище, ім’я, по батькові)

1. Тема роботи: «Дослідження методів продуктивності .NET застосунків залежно від вибору контейнера Dependency Injection»

Затверджена наказом по університету від 15.04.2025р. № 290 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 12.06 .2025 р.

3. Вихідні дані до роботи: *аналіз життєвих циклів об'єктів у Dependency Injection (Transient, Scoped, Singleton), огляд існуючих DI-контейнерів (.NET), розроблена методологія тестування продуктивності контейнерів у різних сценаріях, визначення архітектурних підходів до створення тестового застосунку та рекомендації щодо вибору DI-контейнера залежно від завдань.*

4. Перелік питань, що потрібно опрацювати в роботі: *вступ, аналіз предметної галузі, огляд літературних джерел, постановка задачі, дослідження життєвих циклів у DI, порівняння контейнерів Dependency Injection, методологія проведення тестів, аналіз результатів, висновки, перелік джерел посилання, додатки.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання	16.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	17.04.2025 - 18.04.2025	*виконано*
3	Огляд та порівняння популярних контейнерів DI	19.04.2025 – 22.04.2025	*виконано*
4	Аналіз життєвих циклів об’єктів та їх впливу на продуктивність	23.04.2025 – 25.04.2025	*виконано*
5	Розробка тестового застосунку для моделювання навантаження	26.04.2025 – 30.04.2025	*виконано*
6	Проведення серії експериментів для різних DI-контейнерів і життєвих циклів	01.05.2025 – 04.05.2025	*виконано*
7	Підготовка до апробації результатів дослідження. Публікація матеріалів	05.05.2025 – 09.05.2025	*виконано*
8	Аналіз отриманих результатів та формування висновків	10.05.2025 – 11.05.2025	*виконано*
9	Підготовка пояснювальної записки	12.05.2025 – 21.05.2025	*виконано*
10	Підготовка презентації та доповіді	22.05.2025 – 25.05.2025	*виконано*
11	Перевірка на плагіат	29.05.2025	*виконано*
12	Нормоконтроль	30.05.2025	*виконано*
13	Рецензування	06.06.2025	*виконано*
14	Попередній захист	09.06.2025	*виконано*
15	Занесення диплома в електронний архів	10.06.2025	*виконано*
16	Допуск до захисту у зав. кафедри	10.06.2025	*виконано*

Дата видачі завдання 16 квітня 2025р.

Здобувач _____ Іван КОРОБОВ
(підпис)

Керівник роботи _____ доц. Олексій НАЗАРОВ
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 118 стор., 23 рис., 5 табл., 24 джерела.

ВИСОКОНАВАНТАЖЕНІ СИСТЕМИ, ЖИТТЄВІ ЦИКЛИ, МАСШТАБОВАНІСТЬ, ОПТИМІЗАЦІЯ, ТЕСТУВАННЯ ПРОДУКТИВНОСТІ, DI-КОНТЕЙНЕРИ, AUTOFUC, C#, DRYIOC, MS DI, .NET, NINJECT.

Об’єктом дослідження є механізми впровадження залежностей у .NET-застосунках, які впливають на продуктивність програмного забезпечення.

Предметом дослідження виступають життєві цикли об'єктів та DI-контейнери, такі як MS DI, Autofac, Ninject, DryIoc, їхній вплив на швидкодію та ефективність роботи застосунків.

Метою роботи є аналіз і визначення оптимальних комбінацій DI-контейнерів та життєвих циклів об'єктів для підвищення продуктивності .NET-застосунків.

Методи розробки базуються на аналізі існуючих DI-контейнерів, створенні тестового застосунку, проведенні експериментів із використанням інструментів для тестування продуктивності, таких як BenchmarkDotNet.

У результаті роботи було реалізовано тестовий застосунок для проведення навантажувального моделювання, виконано серію експериментів для оцінювання продуктивності популярних DI-контейнерів у різних сценаріях та за різних конфігурацій життєвих циклів. На основі зібраних даних проаналізовано ефективність кожного контейнера з урахуванням швидкодії та споживання ресурсів, побудовано узагальнені метрики продуктивності та сформульовано практичні рекомендації щодо вибору DI-контейнера залежно від типу навантаження та вимог до системи.

HIGH-LOAD SYSTEMS, LIFECYCLES, OPTIMIZATION, PERFORMANCE TESTING, SCALABILITY, DI CONTAINERS, AUTOFAC, C#, DRYIOC, MS DI, .NET, NINJECT.

The object of the research is the dependency injection mechanisms in .NET applications that influence software performance, resource utilization, and scalability.

The subject of the study includes lifecycle management and DI containers such as MS DI, Autofac, Ninject, and DryIoc, focusing on their impact on speed and efficiency in high-load applications.

The aim of the study is to analyze and determine the optimal combinations of DI containers and object lifecycles to enhance the performance of .NET applications while reducing resource consumption and improving user experience.

The research methods are based on analyzing existing DI containers, developing a test application, and conducting experiments using performance testing tools such as BenchmarkDotNet.

The study resulted in the implementation of a test application for load modeling and the execution of a series of experiments to evaluate the performance of popular DI containers under various scenarios and lifecycle configurations. Based on the collected data, the efficiency of each container was analyzed in terms of execution speed and resource usage. Generalized performance metrics were constructed, leading to practical recommendations on selecting DI containers depending on the system’s load profile and functional requirements.

Завідувачу кафедри ПІ
проф. Кирилу СМЕЛЯКОВУ

ЗАЯВА

щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Коробов Іван Русланович, здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи ІПЗм-23-1 кафедра програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Дослідження методів продуктивності .NET застосунків залежно від вибору контейнера Dependency Injection», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

20.05.2025  Іван КОРОБОВ

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AOP – Aspect-Oriented Programming (аспектно-орієнтоване програмування)

API – Application Programming Interface (інтерфейс програмування застосунків)

CPU – Central Processing Unit (центральний процесор)

CQRS – Command and Query Responsibility Segregation (розподіл відповідальності за команди і запити)

CRUD – Create, Read, Update, Delete (створення, читання, оновлення, видалення)

DI – Dependency Injection (впровадження залежностей)

GC – Garbage Collector (збирач сміття)

IOC – Inversion of Control (інверсія керування)

MS DI – Microsoft.Extensions.DependencyInjection

NET – платформа Microsoft для розробки програмного забезпечення

UML – Unified Modeling Language (уніфікована мова моделювання)

VSA – Vertical Slice Architecture (вертикальна slice-архітектура)

ВСТУП

В умовах стрімкого розвитку інформаційних технологій та зростання складності програмних систем забезпечення високої продуктивності й раціонального використання ресурсів набуває ключового значення. Підвищені вимоги до швидкодії, масштабованості та надійності спонукають розробників застосовувати ефективні архітектурні підходи, серед яких важливе місце займає патерн Dependency Injection (DI). Його використання сприяє зменшенню зв’язності компонентів, покращує тестованість та гнучкість програмного коду.

DI забезпечує ефективне управління залежностями між компонентами, що особливо актуально в умовах мікросервісної архітектури. Проте неправильний вибір контейнера або конфігурації життєвих циклів об'єктів може негативно вплинути на продуктивність. Вибір між контейнерами DI, такими як MS DI, Autofac, Ninject чи DryIoc, і життєвими циклами (Transient, Scoped, Singleton) суттєво впливає на швидкодію та споживання ресурсів.

Актуальність теми дослідження полягає в необхідності глибокого розуміння впливу вибору контейнера DI та конфігурації життєвих циклів об'єктів на продуктивність .NET-застосунків. У контексті зростаючих вимог до продуктивності та ефективності програмного забезпечення, розробники стикаються з вибором оптимальних інструментів та підходів для досягнення поставлених цілей. Недостатнє врахування цих аспектів може призвести до зниження швидкодії застосунку, підвищення споживання пам'яті та ресурсів процесора, що негативно впливає на користувацький досвід та масштабованість системи.

Метою роботи є теоретичне дослідження впливу вибору контейнера Dependency Injection та конфігурації життєвих циклів об'єктів на продуктивність та споживання ресурсів .NET-застосунків. Для досягнення цієї мети необхідно виконати такі завдання:

- аналіз механізмів DI у .NET і їхнього впливу на архітектуру;
- дослідження особливостей MS DI, Autofac, Ninject, DryIoc;

- дослідження впливу життєвих циклів об'єктів (Transient, Scoped, Singleton);
- порівняльний аналіз продуктивності контейнерів DI;
- розробку рекомендацій для вибору контейнерів і налаштувань життєвих циклів.

Об’єктом дослідження є механізми впровадження залежностей у .NET-застосунках, які впливають на архітектуру програмного забезпечення та його продуктивність.

Предметом дослідження виступає вплив вибору контейнера Dependency Injection та конфігурації життєвих циклів об'єктів на продуктивність і споживання ресурсів .NET-застосунків.

Методи дослідження базуються на аналізі наукової та технічної літератури, систематизації отриманих знань та порівняльному аналізі різних підходів до впровадження залежностей. У роботі використовуються теоретичні методи дослідження, що дозволяють обґрунтувати вплив різних факторів на продуктивність застосунків та розробити рекомендації щодо оптимізації їх роботи.

Наукова новизна роботи полягає у комплексному аналізі впливу вибору контейнера Dependency Injection та життєвих циклів об'єктів на продуктивність .NET-застосунків. В роботі вперше систематизовано знання про особливості різних контейнерів DI у контексті їх впливу на швидкодію та ефективність використання ресурсів, а також розроблено рекомендації щодо оптимального налаштування життєвих циклів об'єктів.

Практичне значення отриманих результатів полягає у можливості використання теоретичних висновків та рекомендацій розробниками програмного забезпечення для підвищення продуктивності та оптимізації .NET-застосунків. Результати дослідження можуть бути застосовані при виборі контейнера Dependency Injection та налаштуванні життєвих циклів об'єктів у реальних проєктах, що сприятиме оптимізації споживання ресурсів та покращенню користувацького досвіду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз проблемної області дослідження

Розробка програмного забезпечення на платформі .NET займає вагоме місце в сучасній індустрії інформаційних технологій. Платформа .NET, розроблена компанією Microsoft, є кросплатформеною екосистемою, яка надає розробникам широкий спектр інструментів та бібліотек для створення різноманітних застосунків.

Ключовими принципами розробки на .NET є модульність, повторне використання коду, масштабованість та об’єктно-орієнтований підхід. Модульність дозволяє розбивати застосунок на незалежні компоненти, що спрощує його підтримку та розширення. Повторне використання коду зменшує час розробки та підвищує якість програмного забезпечення. Масштабованість забезпечує адаптацію системи до зростаючих навантажень, а об'єктно-орієнтований підхід сприяє кращій структуризації коду.

У сучасних умовах швидкого розвитку технологій та зростання вимог до програмного забезпечення особливої уваги набуває концепція слабкого зв'язування компонентів системи. Слабке зв'язування дозволяє зменшити залежність між компонентами, що спрощує модифікацію та тестування системи. У цьому контексті патерн Dependency Injection (DI) стає ключовим інструментом для досягнення цієї мети. DI є реалізацією більш загального принципу інверсії керування (Inversion of Control, IoC), який полягає в передачі контролю за створенням об'єктів з класу, що їх використовує, зовнішньому контейнеру або фреймворку.

Dependency Injection дозволяє ін'єктувати залежності об'єктів ззовні, замість того щоб створювати їх всередині класів. Це зменшує зв'язність коду та підвищує його тестованість, оскільки залежності можуть бути легко замінені на мок-об'єкти під час тестування. Використання DI сприяє дотриманню принципу інверсії залежностей (Dependency Inversion Principle) – останнього з п'яти принципів SOLID, який стверджує, що модулі верхнього рівня не повинні залежати від модулів нижнього рівня; обидва повинні залежати від абстракцій.

Принцип Dependency Inversion вказує на необхідність будувати системи так, щоб вони залежали від абстракцій, а не від конкретних реалізацій. Це дозволяє легко замінювати реалізації без зміни коду, що використовує ці абстракції. DI допомагає впровадити цей принцип на практиці, забезпечуючи ін'єкцію залежностей у вигляді інтерфейсів або абстрактних класів.

Схема взаємодії компонентів у патерні Dependency Injection включає три основні елементи (див. рис. 1.1): клієнтський, серверний та інжекторний класи. Ця схема відображає архітектурний підхід до побудови слабозв’язаних компонентів системи, де створення та конфігурація об'єктів делегується зовнішньому механізму, що сприяє покращенню тестованості, гнучкості та підтримуваності застосунків.

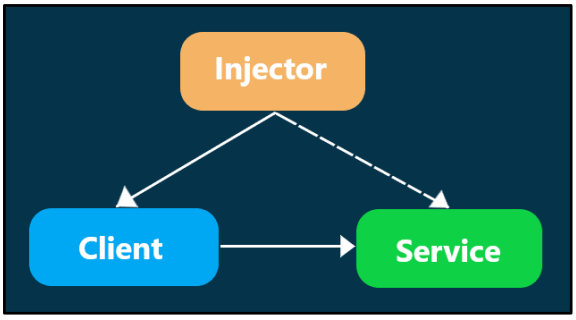

Рисунок 1.1 – Схема взаємодії компонентів у патерні Dependency Injection (за даними [1])

Клієнтський клас (Client Class) залежить від певної служби або функціональності. Він використовує сервіси, але не відповідає за їх створення. У контексті .NET, клієнтським класом часто виступають контролери (Controllers), які отримують залежності через конструктор або властивості.

Сервісний клас (Service Class) надає певну функціональність або сервіс, необхідний клієнтському класу. Він реалізує інтерфейс або абстракцію, яку використовує клієнтський клас.

Інжекторний клас (Injector Class) – це механізм або контейнер, який відповідає за створення та впровадження залежностей у клієнтський клас. У .NET для цього зазвичай використовується вбудований контейнер MS DI, який підтримує базові механізми впровадження залежностей.

Роль Dependency Injection у сучасній розробці програмного забезпечення важко переоцінити. У .NET-середовищі використання DI стало стандартом де-факто, особливо з появою ASP.NET Core, де вбудований контейнер DI є невід'ємною частиною фреймворку. Це дозволяє розробникам легко впроваджувати DI у свої проєкти без додаткових зусиль.

Згідно з дослідженням [2], подальший розвиток методів оцінки програмного забезпечення на етапі проєктування сприятиме виявленню потенційних вузьких місць та підвищенню ефективності роботи системи. Це також дозволить більш усвідомлено підходити до налаштування та вибору DI-контейнерів, оптимізуючи їх під конкретні вимоги архітектури та продуктивності.

Попри переваги, використання DI може створювати виклики, зокрема щодо продуктивності. Неправильний вибір контейнера або конфігурація життєвих циклів об’єктів здатні спричинити затримки ініціалізації, надмірне споживання пам’яті й зниження ефективності. Як зазначено у роботі [3], адаптивна оптимізація у розподілених потокових системах допомагає краще керувати ресурсами, що особливо актуально при використанні DI.

Виклики, пов'язані з використанням DI, включають необхідність балансування між гнучкістю та ефективністю. Використання складних контейнерів DI з багатим набором функцій може додати оверхед до процесу резолюції залежностей. Крім того, неправильне налаштування життєвих циклів об'єктів може спричинити проблеми з витоком пам'яті або надмірним створенням об'єктів, що навантажує систему.

У сучасних високонавантажених застосунках навіть незначні затримки можуть мати суттєвий вплив на користувацький досвід. Тому розуміння того, як різні контейнери DI та їхні налаштування впливають на продуктивність, є критично важливим для розробників, які прагнуть оптимізувати свої системи. Вибір правильного контейнера та налаштування життєвих циклів об'єктів може суттєво підвищити ефективність застосунку та зменшити витрати ресурсів.

1.2 Огляд існуючих підходів та обмежень

Традиційно в .NET-застосунках управління залежностями здійснювалося без використання спеціалізованих контейнерів DI. Розробники вручну створювали екземпляри необхідних класів, що призводило до високого ступеня зв'язності коду та ускладнювало тестування та підтримку. Такий підхід обмежував гнучкість системи та робив її менш адаптивною до змін.

З появою концепції Dependency Injection ця проблема була частково вирішена шляхом впровадження патернів ін'єкції залежностей через конструктори, властивості або методи. Це дозволило розробникам делегувати створення залежностей зовнішнім механізмам та спростило модульне тестування. Однак, без використання спеціалізованих контейнерів DI, управління залежностями все ще залишалося складним завданням.

Вбудований контейнер Dependency Injection у .NET Core, Microsoft.Extensions.DependencyInjection (MS DI), став стандартним інструментом для управління залежностями у сучасних .NET-застосунках. Він надає базовий набір функціональності, достатній для більшості сценаріїв. MS DI підтримує різні життєві цикли сервісів – Singleton, Scoped, Transient – що дозволяє розробникам контролювати час життя об'єктів.

Проте його можливості можуть бути обмеженими у складних випадках, де потрібна розширена конфігурація або специфічні функції, такі як інтерсептори, аспектно-орієнтоване програмування або модульність. У таких випадках розробники звертаються до сторонніх контейнерів DI.

Сторонні контейнери DI, такі як Autofac, DryIoc та Ninject, пропонують більш широкий набір можливостей:

– Autofac підтримує гнучку конфігурацію, модульність, інтерсептори та аспектно-орієнтоване програмування, що дозволяє будувати більш складні архітектури;
– DryIoc вирізняється високою продуктивністю та гнучкістю, підтримує умовне інжектування та складні життєві цикли, а також легко інтегрується з .NET Core;

– Ninject відомий своєю простотою та можливістю розширення через модулі, що спрощує управління залежностями в великих проєктах.

Використання сторонніх контейнерів впровадження залежностей (DI) може ускладнити архітектуру застосунку та негативно вплинути на його продуктивність через додатковий оверхед. Цей аспект особливо важливий у високонавантажених системах, де навіть незначні втрати ефективності можуть мати відчутні наслідки. Водночас відсутність чіткої та порівняльної документації щодо переваг і недоліків різних контейнерів значно ускладнює їхній обґрунтований вибір і налаштування життєвих циклів об'єктів. У результаті розробники можуть припускатися помилок, що знижують загальну продуктивність. Як підтверджується у дослідженні [4], ефективна робота з великими обсягами даних вимагає не лише вдало обраного DI-контейнера, а й продуманої архітектури доступу до ресурсів і структури зберігання, що разом забезпечує стабільність і високу швидкодію системи.

Обмеження та проблеми, пов'язані з різними контейнерами та життєвими циклами об'єктів, включають:

– продуктивність резолюції залежностей. Деякі контейнери можуть бути повільнішими у порівнянні з іншими, особливо при великій кількості зареєстрованих сервісів або складних графах залежностей. Це може призводити до затримок при запуску застосунку або при створенні нових об'єктів;
– споживання пам'яті. Неправильне налаштування життєвих циклів може призвести до витоків пам'яті або надмірного споживання ресурсів. Наприклад, якщо об'єкти з великим обсягом даних зареєстровані як Singleton, вони залишатимуться в пам'яті протягом всього часу роботи застосунку, що може бути недоцільним;
– потокобезпека. Використання Singleton-об'єктів в багатопотокових середовищах може спричинити проблеми з потокобезпекою, якщо не забезпечити відповідних механізмів синхронізації. Це може призвести до

непередбачуваної поведінки застосунку та помилок, які важко діагностувати;

– складність конфігурації. Сторонні контейнери можуть мати складніші механізми конфігурації, що вимагає від розробників додаткових зусиль для їхнього освоєння та правильного налаштування. Це може збільшити час розробки та ускладнити підтримку коду.

У роботі [5] розглянуто різні архітектурні стилі для побудови розподілених систем та проведено їхній порівняльний аналіз. На основі аналізу можна зробити висновок, що незалежно від обраної архітектури, правильна конфігурація DI-контейнерів залишається важливим елементом для забезпечення стабільної продуктивності та масштабованості системи.

Враховуючи ці обмеження, розробникам важливо ретельно обирати DI-контейнер та налаштовувати його відповідно до потреб проєкту. Потрібно балансувати між необхідною функціональністю та потенційними ризиками, пов'язаними з продуктивністю та складністю конфігурації. Розуміння переваг та недоліків кожного підходу дозволяє прийняти обґрунтоване рішення та забезпечити ефективну роботу застосунку.

1.3 Тенденції та перспективи розвитку

Сучасні тенденції у використанні Dependency Injection у .NET свідчать про зростаючу потребу в гнучких та ефективних механізмах управління залежностями. З появою .NET Core та переходом до кросплатформенності роль DI стала ще більш вагомою. Вбудований контейнер MS DI постійно розвивається, додаючи нові можливості та оптимізації, щоб задовольнити потреби спільноти розробників.

Розвиток контейнерів DI спрямований на підвищення продуктивності та зручності використання. Наприклад, зусилля спрямовуються на зменшення оверхеду при резолюції залежностей, покращення механізмів кешування та оптимізацію роботи з життєвими циклами об'єктів. Нові версії контейнерів

надають кращу підтримку асинхронних операцій, що стає все більш важливим у сучасних застосунках.

Перспективи оптимізації продуктивності через налаштування DI полягають у більш глибокому розумінні того, як різні налаштування впливають на роботу застосунку. Правильний вибір життєвого циклу для кожного сервісу може суттєво знизити навантаження на систему. Наприклад, використання Singleton для сервісів без стану дозволяє уникнути надмірного створення об'єктів, тоді як для сервісів зі станом краще використовувати Scoped або Transient, щоб уникнути проблем з потокобезпекою.

Також з'являються нові інструменти та підходи для аналізу та оптимізації продуктивності DI-контейнерів. Інтеграція з засобами профілювання та моніторингу дозволяє розробникам виявляти вузькі місця та оптимізувати конфігурацію своїх застосунків. Це сприяє підвищенню ефективності та надійності програмного забезпечення.

У майбутньому очікується подальший розвиток контейнерів DI з акцентом на продуктивність та масштабованість. Можливе впровадження нових патернів та підходів, спрямованих на зменшення оверхеду та підвищення ефективності роботи з залежностями. Наприклад, можуть бути розроблені контейнери, оптимізовані для використання в мікросервісній архітектурі або хмарних середовищах.

Крім того, зростаючий інтерес до хмарних технологій та мікросервісної архітектури вимагає від DI-контейнерів підтримки розподілених систем та спеціалізованих сценаріїв використання. Це включає можливість інтеграції з різними сервісами, підтримку контейнеризації та оркестрації, а також забезпечення безпеки та надійності в розподілених середовищах.

Розробники все більше звертають увагу на принципи SOLID та важливість дотримання принципу інверсії залежностей. Це сприяє створенню більш стійких та гнучких систем, які легко адаптуються до змін та масштабуються відповідно до потреб бізнесу.

2 ОГЛЯД Й АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Огляд літературних джерел

У процесі дослідження впливу вибору контейнера Dependency Injection (DI) та життєвих циклів об'єктів на продуктивність .NET-застосунків було проведено систематизований огляд літературних та наукових джерел. Метою цього огляду було ідентифікувати актуальні дослідження, теоретичні концепції та практичні підходи, що стосуються теми. Відбір джерел здійснювався за критеріями авторитетності, актуальності, об'єктивності та достовірності інформації.

Авторитетність джерел забезпечувалася шляхом включення робіт визнаних експертів у галузі розробки програмного забезпечення та публікацій у рецензованих наукових журналах і конференціях. Зокрема, було враховано праці Мартіна Фаулера, Марка Сімена, а також статті, опубліковані в IEEE Xplore, MDPI та на офіційному порталі Microsoft. Актуальність джерел підтверджується їхньою відповідністю сучасним тенденціям у сфері DI та .NET-технологій. Обрані джерела охоплюють період від становлення концепції DI до найсучасніших досліджень, що дозволяє врахувати еволюцію підходів та інновації у цій галузі.

Об'єктивність забезпечувалася шляхом відбору джерел, які надають неупереджену інформацію, базовану на теоретичних дослідженнях та емпіричних даних. Перевага надавалася роботам, що містять критичний аналіз та відсутність комерційного впливу. Достовірність інформації гарантувалася використанням джерел з перевірених наукових платформ, таких як IEEE Xplore, MDPI та офіційна документація Microsoft, що забезпечує точність та надійність представленої інформації.

Проведений огляд був структурований за тематичними напрямами, що відповідають аспектам дослідження. Перший напрям включав фундаментальні концепції Dependency Injection та Inversion of Control, розкриті в роботах Мартіна Фаулера та Марка Сімена, які пояснюють принципи зменшення зв'язаності між компонентами та підвищення гнучкості архітектури програмного забезпечення. Другий напрям зосереджувався на практичному впровадженні DI у .NET, зокрема в книзі Марино Посадаса, яка детально розглядає реалізацію DI у .NET Core та

надає практичні рекомендації щодо використання вбудованого DI-контейнера та управління життєвими циклами об'єктів.

Третій напрям стосувався впливу DI на підтримуваність та продуктивність програмного забезпечення. Стаття “Examining the Influence of Dependency Injection on Software Maintainability” досліджує, як DI впливає на підтримуваність та продуктивність, підкреслюючи важливість правильного налаштування DI-контейнерів для підвищення надійності та ефективності .NET-застосунків. Також було розглянуто статтю “Measuring Impact of Dependency Injection on Software Maintainability”, яка вводить нові метрики для оцінки зв'язаності та аналізує вплив DI на ці показники.

Четвертий напрям охоплював офіційну документацію та практичні рекомендації. Документація Microsoft надає актуальну інформацію про впровадження DI у .NET, включаючи детальні інструкції та рекомендації щодо вибору життєвих циклів об'єктів для оптимізації продуктивності.

Таким чином, обрані джерела забезпечують всебічне розуміння теоретичних та практичних аспектів теми дослідження, створюючи міцну основу для подальшого аналізу та експериментальної роботи.

2.2 Аналіз літератури

В основі розуміння Dependency Injection (DI) лежить принцип інверсії керування (Inversion of Control, IoC), детально описаний Мартіном Фаулером у статті “Inversion of Control Containers and the Dependency Injection pattern” [6]. Фаулер заклав теоретичний фундамент для DI, пояснюючи, як цей патерн зменшує зв'язаність між компонентами системи та підвищує гнучкість архітектури. Він підкреслює, що DI сприяє побудові масштабованих і підтримуваних програмних систем, що є критично важливим у сучасній розробці.

Марк Сімен у книзі “Dependency Injection in .NET” [7] розширює ці концепції, зосереджуючись на практичному застосуванні DI у .NET-середовищі. Він детально аналізує різні патерни ін'єкції залежностей (через конструктор, властивості, методи) та їх вплив на архітектуру застосунку. Особливу увагу Сімен

приділяє вибору життєвого циклу об'єктів (Transient, Scoped, Singleton) і підкреслює, що неправильне налаштування може призвести до проблем із продуктивністю, таких як надмірне споживання пам'яті або зниження швидкодії. Проте детального аналізу впливу кожного життєвого циклу на продуктивність у контексті різних DI-контейнерів у його роботі немає.

У роботі Марино Посадаса “Dependency Injection in .NET Core 2.0” [8] детально розглядається впровадження DI у .NET Core. Посадас акцентує увагу на вбудованому DI-контейнері MS DI, його можливостях та обмеженнях. Він надає практичні рекомендації щодо управління залежностями та підкреслює важливість правильного налаштування життєвих циклів об'єктів для оптимізації продуктивності застосунків. Однак питання впливу вибору різних DI-контейнерів та їх комбінацій з життєвими циклами на продуктивність не є центральним у його дослідженні.

Стаття “Examining the Influence of Dependency Injection on Software Maintainability” [9] досліджує вплив DI на підтримуваність та продуктивність програмного забезпечення. Автори доводять, що DI не лише зменшує зв'язаність між компонентами, а й підвищує ефективність тестування та полегшує оновлення коду. Вони підкреслюють, що правильне налаштування DI-контейнерів, таких як MS DI або сторонніх бібліотек, сприяє підвищенню надійності та продуктивності .NET-застосунків. Проте детального аналізу впливу конкретних життєвих циклів у поєднанні з різними DI-контейнерами на продуктивність не було проведено.

Стаття “Measuring Impact of Dependency Injection on Software Maintainability” [10] також досліджує вплив DI на підтримуваність програмного забезпечення, вводячи нові метрики для вимірювання зв'язаності та аналізуючи, як DI впливає на ці показники. Автори пропонують метрики Dependency Injection-weighted Afferent Couplings (DCE) та Dependency Injection-weighted Coupling Between Objects (DCBO) для оцінки пропорції використання DI в проєктах. Дослідження показує, що DI може зменшити зв'язаність між компонентами та підвищити підтримуваність системи. Проте автори зазначають, що їхні метрики

не враховують вплив різних життєвих циклів об'єктів та специфіки різних DI-контейнерів на продуктивність.

Офіційна документація Microsoft [11] виступає важливим джерелом практичних знань про реалізацію DI у .NET. Вона надає детальні інструкції щодо використання вбудованого DI-контейнера та рекомендації з вибору життєвих циклів об'єктів. Документація підкреслює, що правильне налаштування життєвих циклів, наприклад використання Singleton для безстанових сервісів, допомагає уникнути витоків пам'яті та покращує продуктивність системи. Проте вона не надає детального порівняння різних DI-контейнерів та їх взаємодії з життєвими циклами об'єктів.

Аналіз літератури дозволяє зробити кілька ключових висновків. По-перше, DI сприяє зменшенню зв'язаності між компонентами та підвищує підтримуваність програмного забезпечення, що підтверджено багатьма дослідженнями. По-друге, правильне налаштування DI-контейнерів може покращити продуктивність застосунків, проте детальний аналіз впливу життєвих циклів об'єктів та вибору різних DI-контейнерів залишається недостатнім. По-третє, існує прогалина в дослідженнях щодо детального аналізу впливу конкретних життєвих циклів у поєднанні з різними DI-контейнерами на продуктивність .NET-застосунків. Методи та підходи, застосовані в попередніх дослідженнях, такі як теоретичний аналіз, створення метрик, емпіричне тестування та порівняльні дослідження, не дають змоги здійснити комплексний аналіз комбінацій життєвих циклів з різними DI-контейнерами.

2.3 Оцінка актуальності та новизни

Актуальність представленої інформації підтверджується зростаючими вимогами до продуктивності та підтримуваності програмних систем. Використання DI є стандартною практикою, але питання оптимізації його використання залишається актуальним. Аналіз літератури виявив прогалину у дослідженнях: відсутність детального аналізу впливу вибору життєвих циклів об'єктів у поєднанні з різними DI-контейнерами на продуктивність .NET-

застосунків. Ніхто в спільноті розробників не створив статтю, яка б комплексно досліджувала це питання.

Наукова новизна полягає в можливості проведення дослідження, яке детально аналізує, як кожен життєвий цикл об'єкта впливає на продуктивність застосунку при використанні різних DI-контейнерів. Такий аналіз дозволить розробникам краще розуміти, коли використовувати той чи інший життєвий цикл у поєднанні з конкретним DI-контейнером для досягнення оптимальної продуктивності.

Потенційний внесок у галузь полягає у заповненні прогалини в знаннях шляхом проведення детального аналізу впливу комбінацій життєвих циклів та DI-контейнерів на продуктивність. Це надасть розробникам практичні рекомендації щодо оптимального вибору DI-контейнерів та життєвих циклів об'єктів, що сприятиме покращенню продуктивності .NET-застосунків через обґрунтований вибір технологій. Таким чином, проведення такого дослідження є обґрунтованим та необхідним, оскільки воно заповнить існуючу прогалину та сприятиме розвитку практичних підходів до оптимізації продуктивності програмних систем на базі .NET.

2.4 Висновки з огляду

Проведений огляд літератури дозволяє зробити кілька ключових висновків, що безпосередньо впливають на тему дослідження. По-перше, Dependency Injection (DI) визнається фундаментальним підходом для зменшення зв'язаності між компонентами та підвищення підтримуваності програмних систем. Роботи Мартіна Фаулера та Марка Сімена підтверджують важливість DI як теоретично, так і практично, особливо у контексті .NET-середовища.

По-друге, сучасні дослідження свідчать про позитивний вплив DI на підтримуваність та надійність програмного забезпечення. Статті “Examining the Influence of Dependency Injection on Software Maintainability” та “Measuring Impact of Dependency Injection on Software Maintainability” підкреслюють, що правильне налаштування DI-контейнерів та використання відповідних патернів ін'єкції

залежностей сприяє покращенню ефективності тестування та спрощує процес оновлення коду. Це має велике значення для довгострокової продуктивності та масштабованості великих проєктів.

Однак, незважаючи на значний внесок цих робіт, виявлено суттєву прогалину в існуючих дослідженнях. Жодне з розглянутих джерел не надає детального аналізу впливу вибору життєвих циклів об'єктів (Transient, Scoped, Singleton) у поєднанні з різними DI-контейнерами на продуктивність .NET-застосунків. Це питання залишається мало вивченим, що створює необхідність у подальших дослідженнях.

Таким чином, основні висновки огляду літератури підкреслюють актуальність та важливість теми дослідження. Визначена прогалина у знаннях обґрунтовує потребу в проведенні власного дослідження, спрямованого на детальний аналіз впливу комбінацій життєвих циклів та DI-контейнерів на продуктивність. Таке дослідження дозволить розробникам краще розуміти, як оптимально використовувати DI для підвищення ефективності своїх застосунків. Це не лише заповнить існуючу прогалину, але й сприятиме розвитку практичних рекомендацій, що підвищать якість та продуктивність програмного забезпечення на платформі .NET.

3 ПОСТАНОВКА ЗАДАЧІ

Розробка високопродуктивних і масштабованих систем є ключовою задачею сучасної індустрії програмного забезпечення. Завдяки гнучкості та кросплатформенності, платформа .NET стала популярною для створення складних застосунків, а патерн Dependency Injection (DI) є одним із основних підходів для підвищення їхньої підтримуваності та масштабованості.

Попри те, що концепція DI широко відома та активно використовується у .NET-розробці, низка аспектів потребує глибшого вивчення. Зокрема, недостатньо досліджено, як вибір конкретного DI-контейнера та налаштування життєвих циклів об'єктів впливають на продуктивність застосунку. У високонавантажених системах навіть незначні затримки можуть суттєво погіршити користувацький досвід та ефективність застосунку в цілому. Відсутність детальних досліджень створює проблему для розробників, які прагнуть оптимізувати свої застосунки без належної інформації для обґрунтованого вибору технологій та конфігурацій.

Мета цього дослідження полягає в тому, щоб заповнити цю прогалину в знаннях, провівши комплексний аналіз впливу вибору контейнера Dependency Injection та життєвих циклів об'єктів на продуктивність .NET-застосунків. Це дозволить розробникам більш усвідомлено підходити до вибору технологій та налаштування DI, оптимізуючи їх під конкретні вимоги архітектури та продуктивності. Досягнення цієї мети передбачає вирішення кількох ключових завдань, включаючи аналіз існуючих DI-контейнерів, визначення впливу різних життєвих циклів об'єктів, розробку методики експериментального дослідження, створення тестового застосунку, проведення серії експериментів, аналіз отриманих результатів та розробку практичних рекомендацій.

Аналіз існуючих DI-контейнерів, таких як MS DI, Autofac, Ninject та DryIoc є першим кроком у цьому дослідженні. Ці контейнери широко використовуються в .NET-спільноті та надають різні можливості для управління залежностями. Вивчення їх архітектури та механізмів роботи дозволить зрозуміти, як вони впливають на процес резолюції залежностей та які фактори можуть впливати на

продуктивність. Особливу увагу буде приділено тому, як кожен контейнер реалізує різні життєві цикли об'єктів і які можливості оптимізації вони надають.

Наступним етапом є визначення впливу різних життєвих циклів об'єктів (Transient, Scoped, Singleton) на продуктивність застосунку. Життєві цикли визначають тривалість існування об'єктів у пам'яті та частоту їх створення. Неправильне налаштування життєвих циклів може призвести до надмірного споживання пам'яті, проблем з потокобезпекою та зниження загальної продуктивності. Аналіз цих аспектів дозволить виявити оптимальні комбінації життєвих циклів для різних типів сервісів.

Важливим та критичним кроком є розробка методики експериментального дослідження для отримання достовірних результатів. Планується створити тестовий застосунок на платформі .NET, який моделюватиме реальні умови роботи та міститиме сервіси, контролери й базу даних. Він буде достатньо складним для типових сценаріїв, але керованим для аналізу результатів.

Ключовим етапом дослідження є експерименти з різними комбінаціями DI-контейнерів і життєвих циклів об'єктів. Планується тестування конфігурацій із категоризацією сервісів за функціональністю. Для збору даних використовуватиметься .NET Benchmark [12] для моделювання навантаження та профайлери продуктивності для аналізу часу відгуку, споживання ресурсів і пропускної здатності. Аналіз використання ресурсів під різними навантаженнями та конфігураціями допоможе виявити вузькі місця й оптимізувати систему.

Обґрунтування вибору методів дослідження, програмних рішень та засобів базується на необхідності отримання кількісних даних про вплив конкретних факторів на продуктивність Вибір платформи .NET зумовлений її широким використанням в індустрії та значущістю для сучасних досліджень у сфері програмного забезпечення. Використання ASP.NET Core для розробки тестового застосунку дозволяє моделювати веб-застосунки, які є поширеними в реальних проектах. Вибір DI-контейнерів зумовлений їхньою популярністю та різними можливостями, що дає змогу провести порівняльний аналіз.

Одним із обмежень дослідження є те, що воно зосереджується на чотирьох DI-контейнерах і не охоплює всі доступні варіанти. Крім того, тестовий застосунок, хоча і розроблений для моделювання реальних умов, не може відобразити всю різноманітність можливих сценаріїв використання. Апаратні ресурси, на яких будуть проводитися експерименти, можуть обмежувати масштабування навантаження, що слід враховувати при інтерпретації результатів.

Для реалізації експерименту необхідні ресурси включають апаратне забезпечення з достатньою потужністю для проведення навантажувальних тестів, програмне забезпечення (платформа .NET, СУБД, DI-контейнери, інструменти для тестування та профілювання), а також доступ до літератури та документації. Часові ресурси передбачають період для розробки, налаштування, проведення експериментів та аналізу даних.

Очікувані результати включають отримання детальних даних про вплив вибору DI-контейнера та життєвих циклів об'єктів на продуктивність .NET-застосунків, визначення оптимальних комбінацій для різних типів сервісів та сценаріїв використання, а також розробку практичних рекомендацій для розробників. Це сприятиме підвищенню ефективності програмного забезпечення та кращому використанню ресурсів.

Значущість дослідження полягає в заповненні прогалини у знаннях щодо впливу DI-контейнерів та життєвих циклів об'єктів на продуктивність. Результати будуть корисними як для академічних досліджень, так і для практичного застосування в індустрії, особливо в умовах зростаючих вимог до швидкодії та масштабованості застосунків.

Проведення цього дослідження сприятиме розвитку знань у сфері оптимізації .NET-застосунків, допоможе розробникам приймати обґрунтовані рішення щодо вибору технологій та конфігурацій, а також підвищить загальну якість та ефективність програмного забезпечення. Це відповідає сучасним тенденціям розвитку індустрії програмного забезпечення, де важливість продуктивності та масштабованості систем постійно зростає.

4 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

4.1 Життєві цикли об'єктів у Dependency Injection

У контексті патерну Dependency Injection (DI) життєві цикли об'єктів відіграють ключову роль у керуванні створенням та існуванням екземплярів класів. Життєвий цикл визначає, як довго об'єкт існує в пам'яті та коли створюється новий екземпляр. Правильне налаштування життєвих циклів сприяє ефективному використанню ресурсів, забезпечує потокобезпеку та оптимізує продуктивність застосунків. Історично концепція життєвих циклів виникла разом із розвитком DI-контейнерів, коли з'явилася потреба в гнучкому керуванні створенням та утилізацією об'єктів залежно від контексту їх використання [13].

Життєві цикли вирішують проблему керування станом об'єктів та їхньою доступністю в різних частинах застосунку. Без належного управління життєвими циклами можуть виникати проблеми з надмірним споживанням пам'яті, витоками ресурсів або конфліктами при доступі з різних потоків. У середовищі з високим навантаженням ці проблеми можуть суттєво вплинути на стабільність та продуктивність системи.

4.1.1 Життєвий цикл Transient

Життєвий цикл Transient передбачає створення нового екземпляра сервісу при кожному запиті до нього. Це означає, що кожного разу, коли об'єкт залежності запитується у DI-контейнера, створюється новий екземпляр класу. Такий підхід забезпечує повну ізоляцію стану між різними використаннями сервісу, що є корисним, коли об'єкт має короткочасний стан або потрібна чиста копія без залишків від попередніх операцій.

У реальних проєктах Transient-життєвий цикл використовується для сервісів, які не зберігають стан між викликами або коли необхідно гарантувати, що кожен клієнт отримує унікальний екземпляр об'єкта. Наприклад, сервіси, що виконують операції форматування, обробки даних без збереження контексту, можуть бути зареєстровані як Transient.

Однак у високонавантажених системах неправильне використання Transient-життєвого циклу може призвести до надмірного споживання ресурсів. Кожне створення нового об'єкта вимагає додаткового часу процесора та пам'яті для ініціалізації. Якщо сервіси, що є ресурсомісткими або часто використовуються, зареєстровані як Transient, це може спричинити зниження продуктивності та підвищене навантаження на систему збору сміття (Garbage Collector).

Наприклад, у веб-застосунку, який обслуговує тисячі запитів на секунду, використання Transient для сервісу, що здійснює доступ до бази даних, призведе до створення великої кількості екземплярів цього сервісу. Це може спричинити затримки в обробці запитів та збільшити час відгуку системи.

4.1.2 Життєвий цикл Scoped

Життєвий цикл Scoped передбачає створення одного екземпляра сервісу на область (scope), яка зазвичай відповідає життєвому циклу запиту в веб-застосунку. У контексті ASP.NET Core область починається на початку HTTP-запиту і завершується після його обробки. Це означає, що всі залежності, зареєстровані як Scoped, будуть спільними в межах одного запиту, але ізольовані від інших запитів.

Scoped-життєвий цикл ідеально підходить для сервісів, які зберігають стан, специфічний для поточного запиту. Наприклад, сервіси, що працюють з контекстом користувача, транзакціями бази даних або логуванням дій у межах одного запиту, можуть бути зареєстровані як Scoped. Це дозволяє забезпечити цілісність даних та уникнути конфліктів між паралельними запитами.

У високонавантажених системах неправильне використання Scoped-життєвого циклу може призвести до витоків пам'яті або надмірного споживання ресурсів, якщо область не завершується коректно. Наприклад, якщо об'єкти Scoped-сервісів утримуються в статичних змінних або передаються в Singleton-сервіси, вони можуть залишатися в пам'яті після завершення запиту. Це порушує

концепцію області та може спричинити неконтрольоване зростання споживання пам'яті.

Крім того, слід бути обережним при використанні Scoped-сервісів у фонових задачах або потоках, які виходять за межі області запиту. У таких випадках область може бути недоступною, що призведе до винятків або непередбачуваної поведінки застосунку.

4.1.3 Життєвий цикл Singleton

Життєвий цикл Singleton означає, що DI-контейнер створює один екземпляр сервісу при першому запиті і використовує його протягом всього часу роботи застосунку. Це дозволяє спільно використовувати ресурси та зменшити накладні витрати на створення об'єктів. Singleton підходить для сервісів, які є безстановими або потребують зберігати спільний стан, доступний для всіх частин застосунку.

У реальних проєктах Singleton-життєвий цикл використовується для сервісів конфігурації, кешування, роботи з файловою системою або інших ресурсів, де необхідно зберігати спільний стан або забезпечити єдиний доступ до ресурсу. Використання Singleton може підвищити продуктивність, оскільки зменшує кількість створюваних об'єктів та навантаження на систему.

Однак у багатопотокових застосунках Singleton-сервіси можуть спричинити проблеми з потокобезпекою, якщо вони зберігають змінний стан. Оскільки один екземпляр сервісу використовується всіма потоками, відсутність належної синхронізації може призвести до конфліктів та непередбачуваної поведінки. Це особливо критично у високонавантажених системах, де велика кількість потоків одночасно звертається до Singleton-сервісу.

Наприклад, якщо Singleton-сервіс зберігає дані сесії користувача без належної ізоляції, дані можуть змішуватися між різними користувачами. Це не лише спричинить помилки в роботі застосунку, але й може стати причиною серйозних проблем з безпекою. Тому важливо забезпечити, щоб Singleton-сервіси були або безстановими, або реалізовували механізми синхронізації та ізоляції стану.

4.1.4 Висновки щодо вибору життєвих циклів

Правильний вибір життєвого циклу для кожного сервісу є критичним для забезпечення оптимальної продуктивності та надійності застосунку. Вибір залежить від характеру сервісу, його стану та вимог до ізоляції даних. У таблиці 4.1 узагальнено рекомендації щодо вибору життєвого циклу залежно від типу сервісу.

Таблиця 4.1 – Рекомендації щодо вибору життєвого циклу (за даними [14])

Тип сервісу	Рекомендований життєвий цикл
Безстановий сервіс	Singleton
Ресурсоємний сервіс, що часто використовується	Singleton
Сервіс зі станом запиту	Scoped
Сервіс з короткочасним станом	Transient
Сервіс, що потребує унікальний екземпляр	Transient

Неправильне налаштування життєвих циклів може призвести до проблем з продуктивністю, таких як надмірне створення об’єктів, витоки пам’яті або конфлікти при доступі з різних потоків. Також можливі конфлікти при одночасному паралельному доступі до сервісів з різних потоків, особливо у випадках, коли не враховано потокобезпеку. У високонавантажених системах ці проблеми можуть стати критичними, спричиняючи зниження продуктивності або навіть вихід з ладу.

Розробникам необхідно ретельно аналізувати функціональні вимоги до кожного сервісу, щоб обрати відповідний життєвий цикл. Такий підхід забезпечує оптимальну продуктивність, ефективне використання ресурсів та мінімізує ризики, пов’язані з помилками конфігурації або неочікуваною поведінкою застосунку.

4.2 Огляд контейнерів впровадження залежностей у .NET

У сучасній розробці програмного забезпечення на платформі .NET контейнер впровадження залежностей (DI) є невід'ємним компонентом для побудови гнучкої та масштабованої архітектури. Незалежно від вибраної бібліотеки, основним аспектом їхньої роботи є управління створенням та життєвим циклом об'єктів, що впроваджуються. Кожен контейнер забезпечує різні механізми для реєстрації та резолюції залежностей, але загальними для всіх є такі життєві цикли:

– singleton: об'єкт створюється один раз і використовується протягом усього часу роботи застосунку;
– scoped: створюється окремий екземпляр для кожної області, наприклад, для кожного HTTP-запиту;
– transient: при кожному запиті створюється новий екземпляр.

Існує декілька популярних DI-контейнерів, кожен з яких має свої особливості, переваги та недоліки. У цьому розділі проведемо детальний аналіз основних DI-контейнерів: MS DI, Autofac, Ninject та DryIoc.

Перед тим як перейти до аналізу, доцільно звернути увагу на загальну популярність та динаміку використання цих бібліотек серед .NET спільноти. На рисунку 4.1 представлено графік, що демонструє кількість завантажень кожного з розглянутих контейнерів упродовж останніх шести років згідно з даними NuGet.

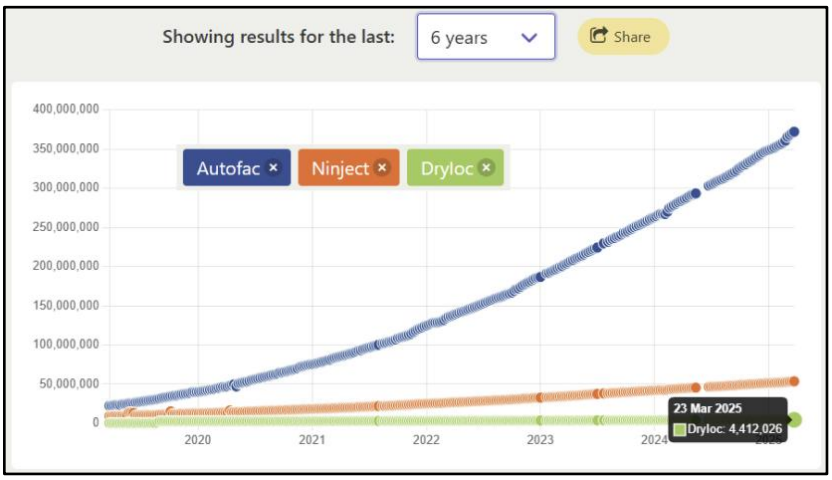

Рисунок 4.1 – Динаміка підключень Autofac, Ninject та DryIoc за останні шість років (за даними [15])

З графіка видно, що найбільш активно використовується бібліотека Autofac, яка демонструє стабільне експоненційне зростання, значно випереджаючи інших конкурентів. Ninject зберігає помірне, але стабільне зростання, що може свідчити про його застосування в довготривалих проєктах або наявність існуючих рішень, які все ще підтримуються. У той же час DryIoc, попри технічну досконаліть і високу продуктивність, залишається відносно нішовим рішенням із невисокою динамікою росту популярності. Така різниця в тенденціях використання може бути пов'язана з рівнем документації, кривою навчання, підтримкою з боку спільноти та сумісністю з екосистемою .NET.

4.2.1 Огляд Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.DependencyInjection (MS DI) є вбудованим DI-контейнером платформи .NET [16]. Він забезпечує базову функціональність для впровадження залежностей і є оптимізованим для продуктивності. MS DI підтримує основні життєві цикли об'єктів: Singleton, Scoped та Transient.

MS DI інтегрований у фреймворк ASP.NET Core, що робить його природним вибором для розробників, які використовують цю платформу. Контейнер орієнтований на простоту використання та високу продуктивність, що досягається завдяки попередній компіляції графа залежностей під час старту застосунку. Це дозволяє зменшити час резолюції залежностей під час виконання, що особливо важливо для високонавантажених систем.

MS DI відзначається високою продуктивністю завдяки оптимізованій архітектурі та попередній компіляції графа залежностей. Інтеграція з .NET Core робить його простим у використанні, не вимагаючи додаткових пакетів або налаштувань.

Проте MS DI має певні обмеження. Контейнер не підтримує такі функції, як інтерсептори, аспектно-орієнтоване програмування (AOP) та складні конфігурації життєвих циклів. Це обмежує його застосування в складних сценаріях, де потрібне більш гнучке управління залежностями.

4.2.2 Огляд Autofac

Autofac є потужним та гнучким DI-контейнером, який надає розробникам широкий спектр можливостей для управління залежностями. Він підтримує модульність, інтерсептори, аспектно-орієнтоване програмування та різноманітні життєві цикли. Autofac дозволяє створювати складні конфігурації залежностей, що робить його придатним для великих та комплексних застосунків [17].

Autofac інтегрується з .NET Core через спеціальні розширення, що дозволяє використовувати його замість або разом з MS DI. Він також підтримує автоматичну реєстрацію сервісів, сканування збірок та інші розширені функції.

Autofac відзначається високою гнучкістю та розширеними можливостями конфігурації. Підтримка аспектно-орієнтованого програмування та інтерсепторів дозволяє реалізовувати складні сценарії, такі як логування, обробка помилок та кешування на рівні контейнера. Однак, ця гнучкість може впливати на продуктивність, оскільки складніша внутрішня архітектура може призводити до збільшення часу резолюції залежностей. Крім того, використання Autofac вимагає більшого часу на освоєння та налаштування, що може збільшити складність проєкту.

4.2.3 Огляд Ninject

Ninject відомий своєю простотою використання та можливістю розширення через модулі. Він надає інтуїтивний синтаксис та дозволяє розробникам легко налаштовувати залежності. Ninject підтримує аспектно-орієнтоване програмування через розширення та може бути інтегрований з різними фреймворками [18].

Однак, Ninject не має вбудованої підтримки Scoped життєвого циклу, що може ускладнити його використання у веб-застосунках. Для вирішення цієї проблеми можна використовувати додаткові пакети, такі як Ninject.Web.Common, які додають підтримку Scoped життєвого циклу через InRequestScope.

Перевагою Ninject є простота та інтуїтивність конфігурації, що дозволяє швидко розпочати роботу з контейнером. Можливість розширення через модулі

робить його гнучким для різних потреб. Однак, значним недоліком є відносно низька продуктивність через використання рефлексії та динамічної генерації проксі. Це може стати проблемою у високонавантажених системах. Крім того, обмежена підтримка Scoped життєвого циклу вимагає додаткових зусиль для налаштування [19].

4.2.4 Огляд DryIoc

DryIoc орієнтований на забезпечення високої продуктивності та низького споживання пам'яті. Він підтримує розширені можливості конфігурації, включаючи роботу з відкритими генериками, інтерсепторами та умовною реєстрацією. DryIoc прагне поєднати переваги MS DI з гнучкістю сторонніх контейнерів [20].

DryIoc використовує агресивне кешування та генерацію виразів для прискорення резолюції залежностей. Це дозволяє досягти високої швидкодії навіть у складних графах залежностей. DryIoc відзначається високою продуктивністю та гнучкістю, що робить його придатним для високонавантажених систем. Однак, багаті можливості контейнера можуть ускладнити його освоєння, особливо для розробників, які не мають досвіду з DI-контейнерами.

Попри свою відносну менш популярність, DryIoc демонструє дуже добрі результати в бенчмарках продуктивності, що робить його перспективним кандидатом для розгляду як потенційної заміни у нових системах , особливо для мікросервісних архітектур. Водночас менш детальна документація може створити додаткові труднощі для розробників.

4.3 Архітектурний підхід до моделювання тестового застосунку

Vertical Slice Architecture (VSA) є сучасним підходом до побудови програмного забезпечення, що забезпечує логічний поділ системи на незалежні функціональні блоки, кожен з яких охоплює всі необхідні рівні для реалізації певної бізнес-логіки (див. рис. 4.2). Цей підхід демонструє значну гнучкість, що

робить його ідеальним для дослідження роботи Dependency Injection (DI) контейнерів у різних умовах [21].

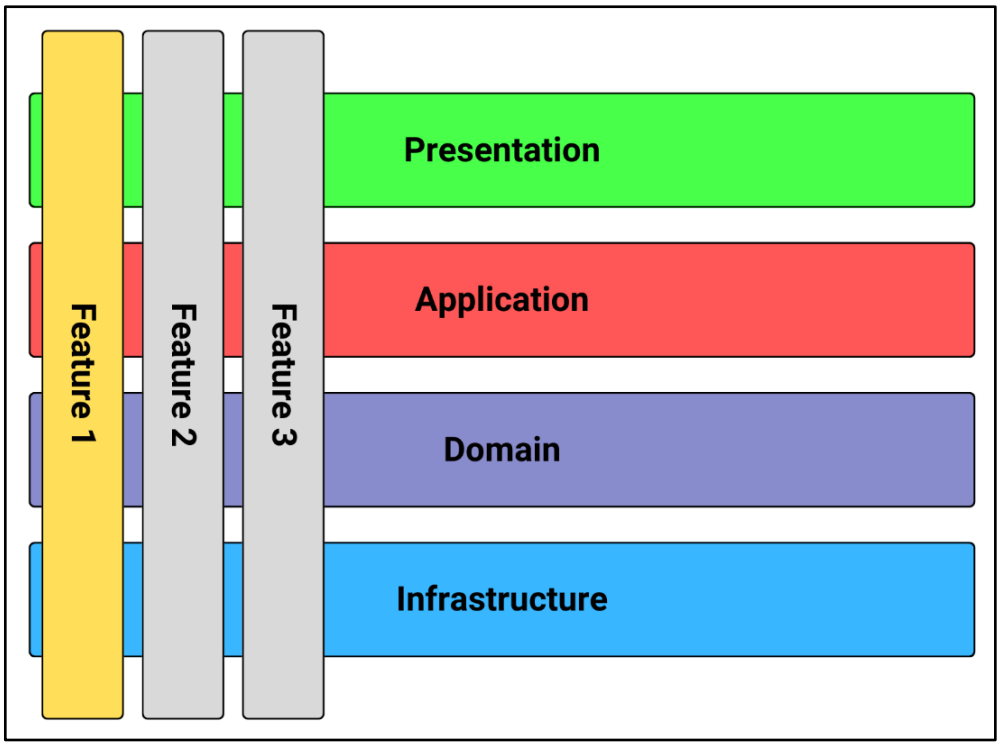

Рисунок 4.2 – Структура Vertical Slice Architecture (за даними [21])

Останніми роками VSA стала популярною завдяки здатності масштабуватися, інтегрувати сучасні підходи, такі як Command and Query Responsibility Segregation (CQRS), та адаптуватися до різноманітних бізнес-вимог. Обраний підхід забезпечує не лише структурну гнучкість, але й мінімізує ризик впливу змін у одному модулі на інші.

Основною концепцією VSA є поділ системи на самодостатні вертикальні зрізи, які охоплюють усі етапи обробки запиту: від контролера до взаємодії з базою даних. Це відрізняє VSA від традиційної багатошарової архітектури, де логіка об'єднана за рівнями (UI, бізнес-логіка, доступ до даних). У VSA кожен зріз є самодостатнім модулем, що дозволяє ізолювати будь-які зміни в межах цього модуля, мінімізуючи ризик негативного впливу на інші частини системи [21].

VSA органічно поєднується з CQRS, розділяючи запити і команди для підвищення читабельності та тестованості. Використання бібліотеки MediatR як посередника спрощує реалізацію модульної архітектури [22]. Кожен зріз

відповідає за окрему бізнес-дію, що сприяє простоті розробки й тестування (див. рис. 4.3).

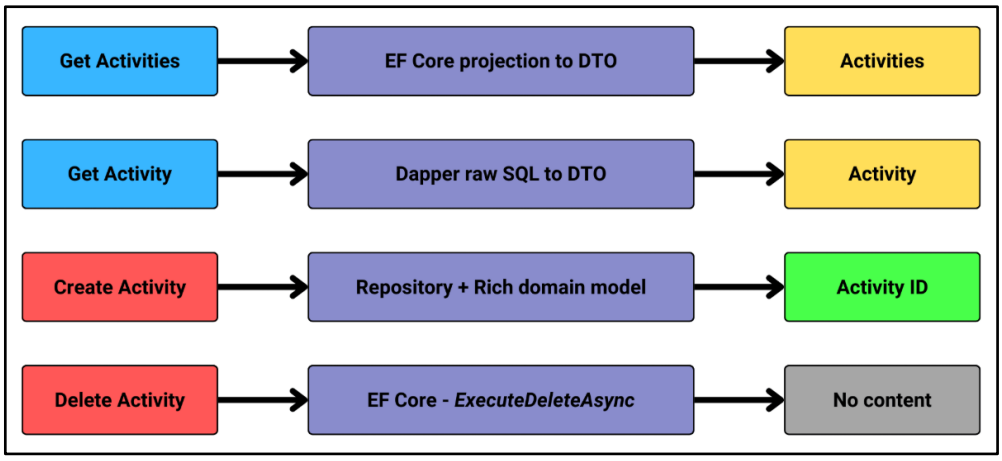

Рисунок 4.3 – Реалізація патерну CQRS із використанням різних підходів до обробки команд та запитів (за даними [22])

Масштабування застосунку відбувається шляхом додавання нових зрізів, які інтегруються незалежно від існуючих. Це дозволяє експериментувати з різними DI-контейнерами, такими як MS DI, Autofac, Ninject, або DryIoc, і порівнювати їхню продуктивність у реальних умовах. Завдяки цьому можна реалізувати і тестувати різні життєві цикли об'єктів (Transient, Scoped, Singleton), уникаючи конфліктів між компонентами інших зрізів. Наприклад, сервіс роботи з базою даних у одному зрізі може використовувати Scoped-життєвий цикл, щоб забезпечити унікальний контекст для кожного запиту. У той час як сервіс кешування у іншому зрізі буде використовувати Singleton, щоб уникнути дублювання в обробці даних.

VSA сприяє автоматизації тестування. Кожен зріз можна перевіряти окремо за допомогою юніт-тестів для бізнес-логіки, інтеграційних тестів для доступу до даних і навантажувальних тестів для перевірки продуктивності.

В результаті було побудовано діаграму тестового застосунку (див. рис. 4.4) з використанням VSA, що забезпечує гнучкість у роботі з різними DI-контейнерами. Це дозволяє порівнювати продуктивність контейнерів у реальних умовах через зміну життєвих циклів об'єктів у залежності від ролі сервісу.

Для зберігання даних обрано Microsoft SQL Server, який забезпечує високу продуктивність і сумісність із сучасними технологіями. Для перевірки продуктивності використовуються xUnit, BenchmarkDotNet, dotTrace та dotMemory.

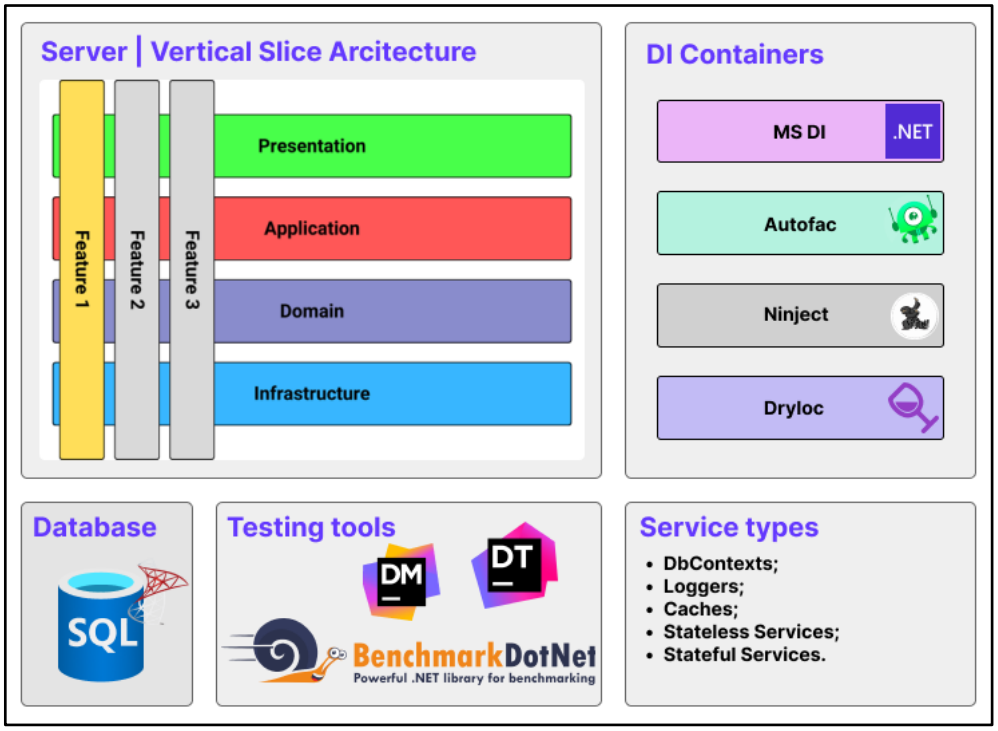

Рисунок 4.4 – Архітектура тестового застосунку (рисунок виконаний самостійно)

Таким чином, VSA є надійним підґрунтям для розробки і тестування програмного забезпечення з використанням різних DI-контейнерів. Завдяки модульній структурі, що поєднує гнучкість, масштабованість і підтримку сучасних підходів, таких як CQRS і MediatR, ця архітектура дозволяє забезпечити ізоляцію компонентів, спрощуючи їх розробку, тестування та підтримку. Обраний підхід є оптимальним для дослідження продуктивності DI-контейнерів у різноманітних сценаріях та створення адаптивних рішень для складних бізнес-завдань.

4.4 Методологія розробки сценаріїв для дослідження DI-контейнерів

Проєктування тестових сценаріїв є важливим етапом у створенні експериментального середовища, спрямованого на аналіз роботи Dependency

Injection (DI) контейнерів у поєднанні з різними життєвими циклами сервісів. Головною метою цього етапу є імітація реальних умов експлуатації програмного забезпечення, що включає варіативність у конфігураціях життєвих циклів сервісів і рівнях навантаження на систему. Це дозволить отримати глибоке розуміння впливу DI-контейнерів на продуктивність, стабільність і ефективність управління ресурсами.

У межах дослідження передбачається охоплення широкого спектру варіантів конфігурації залежностей і сервісів. Основний акцент робиться на створенні сценаріїв, що відображають різноманітність ролей сервісів у реальних проєктах. Сервіси поділяються на кілька категорій залежно від їхньої функціональності: без стану, зі станом, репозиторії, контексти бази даних, а також сервіси кешування та конфігурації. Такий підхід дозволяє розробити універсальні сценарії, які підходять для різних типів застосунків, від невеликих локальних систем до масштабованих корпоративних платформ.

Для моделювання реалістичних умов експлуатації було обрано підхід з використання бібліотеки BenchmarkDotNet, що дає змогу проводити серію замірів з різною кількістю ітерацій, включно з етапами розігріву (warmup) та додатковими налаштуваннями. У цьому дослідженні параметрами для кількості виконуваних операцій було обрано значення 1000, 5000 і 10000. Завдяки такому діапазону вдається оцінити, як різні DI-контейнери, такі як MS DI, Autofac, Ninject та DryIoc, поводяться за умов різної інтенсивності викликів методів та створення об’єктів.

Кожен сценарій у тестуванні охоплює кілька типових операцій, характерних для реальних програмних систем. Наприклад, операції CRUD є базовими для оцінки продуктивності репозиторіїв і контекстів бази даних. У той же час сценарії, що включають виконання складних бізнес-процесів, таких як обробка кількох взаємозалежних запитів, дозволяють оцінити швидкодію та масштабованість сервісів. Окрему увагу приділено сценаріям, пов’язаним із високими обчислювальними навантаженнями, що демонструють вплив конфігурацій життєвих циклів на швидкість обробки.

Формування конфігурацій тестування є ключовим завданням у проєктуванні. Для цього створено кілька базових конфігурацій, що дозволяють експериментувати з різними поєднаннями життєвих циклів. Наприклад, базова конфігурація передбачає використання Transient для сервісів без стану, Scoped для сервісів зі станом і репозиторіїв, а також Singleton для кешів. У альтернативних конфігураціях змінюється життєвий цикл сервісів, наприклад, переведення сервісів без стану в режим Singleton для оптимізації створення об’єктів або застосування Transient до всіх сервісів для перевірки навантаження на систему.

Важливою складовою тестування є оцінка продуктивності за допомогою набору метрик, які будуть отримані внаслідок запуску тестових сценаріїв під керуванням BenchmarkDotNet. У межах дослідження буде зібрано такі показники:

- mean (ms): середній час виконання тестованого методу (у мілісекундах). Це усереднене значення, що дає загальне уявлення про рівень продуктивності під навантаженням;
- error (ms): статистична похибка вимірювання, яка показує, наскільки значно можуть змінюватися значення середнього часу (Mean) у межах тестового діапазону;
- median (ms): медіанний час виконання. 50% замірів мають меншу або рівну цьому значенню тривалість, а решта 50% – більшу. Використовується для розуміння характерного «типового» часу виконання без впливу крайніх випадків;
- min (ms): мінімальний зафіксований час виконання операції. Дає змогу оцінити найсприятливіший сценарій виконання;
- max (ms): максимальний зафіксований час виконання. Дозволяє виявити пікові значення часу, коли система потенційно працює у найменш оптимальному режимі;
- p95 (ms): 95-й перцентиль. Означає, що 95% усіх зафіксованих вимірювань не перевищують цього часу. Це один із ключових показників, що демонструє, як часто виникають рідкісні, але доволі тривалі операції;

- p90 (ms): 90-й перцентиль. Аналогічно до P95, 90% вимірювань є меншими або рівними цьому значенню, а ще 10% можуть бути більшими;
- gen0, gen1, gen2: кількість спрацювань збирача сміття (Garbage Collector) для поколінь 0, 1 і 2. Цей показник відображає інтенсивність очистки пам’яті під час виконання тестів та може свідчити про рівень створення і видалення об’єктів;
- allocated (MB): загальний обсяг виділеної пам’яті у мегабайтах. Дає змогу оцінити, наскільки ресурсоємними були операції в межах конкретного тесту.

Завдяки цим метрикам можна виявити потенційні вузькі місця у роботі застосунку та визначити, наскільки добре різні DI-контейнери поводяться за різних поєднань життєвих циклів сервісів і різної інтенсивності викликів. Такий підхід створює основу для подальших експериментів у реальних умовах, де можна додатково розширити набір метрик або налаштувати сценарії тестування, аби глибше зрозуміти вплив конфігурацій DI та життєвих циклів сервісів на продуктивність у виробничих середовищах.

Щоб забезпечити наочність і краще розуміння архітектурного задуму, наступним кроком стало створення узагальненої концептуальної моделі, що демонструє взаємозв’язки між основними компонентами системи тестування продуктивності (див. рис. 4.5). Для цього було використано UML, зокрема діаграму класів, яка дозволяє відобразити структуру застосунку з акцентом на спадкування, реалізацію інтерфейсів та залежності між класами. У межах цього дослідження діаграма класів допомагає зрозуміти, яким чином організовані бенчмарки, адаптери DI-контейнерів, а також як ініціалізується і виконується запуск сценаріїв тестування.

Діаграма складається з двох основних логічних блоків: підсистеми Benchmarks та підсистеми Infrastructure_DIContainers. Підсистема бенчмарків містить абстрактний клас BenchmarkBase, від якого наслідуються спеціалізовані класи CrudBenchmarks, LoggingBenchmarks та CachingBenchmarks. Кожен з цих класів реалізує метод Tests, у межах якого виконується серія замірів із

використанням бібліотеки BenchmarkDotNet. Таким чином, кожен клас моделює окремий аспект продуктивності: операції з даними, логування та кешування відповідно.

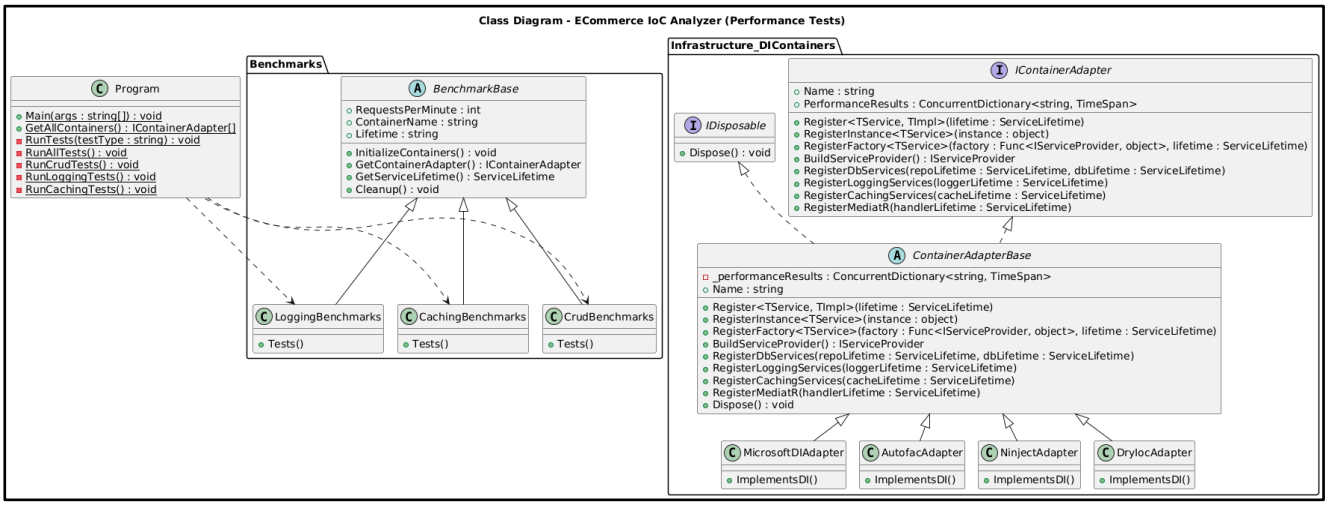

Рисунок 4.5 – UML-діаграма класів тестового середовища для оцінки DI-контейнерів (рисунок виконаний самостійно)

Підсистема інфраструктури DI-контейнерів включає інтерфейс IContainerAdapter, який визначає загальні вимоги до контейнера впровадження залежностей: методи реєстрації сервісів, побудови провайдера, конфігурації життєвих циклів тощо. Абстрактний клас ContainerAdapterBase реалізує базову логіку роботи з контейнерами, а також інтерфейс IDisposable для звільнення ресурсів. Від цього класу наслідуються конкретні реалізації адаптерів для кожного з досліджуваних контейнерів: MicrosoftDIAdapter, AutofacAdapter, NinjectAdapter та DryIocAdapter. Кожен з них реалізує власну логіку адаптації під специфіку відповідного контейнера, дотримуючись спільного контракту.

Варто також зазначити, що центральний клас Program ініціює тестування через метод Main, викликаючи відповідні функції запуску сценаріїв (RunCrudTests, RunLoggingTests, RunCachingTests). Ці методи оперують конкретними реалізаціями бенчмарків та передають до них адаптери контейнерів, отримані через метод GetAllContainers. Така структура дозволяє масштабувати систему, додаючи нові сценарії або адаптери без модифікації вже існуючих компонентів.

Таким чином, наведена UML-діаграма (див. рис. 4.4) слугує фундаментом для розуміння внутрішньої логіки експериментального середовища, де реалізуються й порівнюються різні DI-контейнери. Вона демонструє чіткий поділ відповідальностей між компонентами, що підвищує гнучкість архітектури й спрощує подальше розширення системи за рахунок додавання нових сценаріїв або контейнерів без порушення загальної структури.

4.5 Вимоги до бази даних та умов проведення експериментів

Розробка вимог до бази даних і тестового середовища є важливим етапом у створенні основи для тестування продуктивності DI-контейнерів. База даних має бути налаштована таким чином, щоб відповідати реалістичним умовам, характерним для програм корпоративного рівня. Запропонована модель бази даних передбачає наповнення кожної таблиці великим обсягом даних – щонайменше 100 000 записів. Такий обсяг вибрано для моделювання типових умов експлуатації застосунків, де обробка великих наборів даних є стандартною практикою.

Розмір бази даних у 100 000 записів дозволяє тестувати як базові CRUD-операції, так і складні вибірки даних із фільтрацією, сортуванням та агрегацією. Це допоможе оцінити ефективність роботи DI-контейнерів при виконанні запитів різної складності. Для більш інтенсивного тестування або імітації високонавантажених систем обсяг даних може бути збільшено, масштабування допоможе виявити поведінку контейнерів у критичних умовах.

Навантáження в експерименті реалізується за допомогою серійних запусків методів у фреймворку BenchmarkDotNet. У процесі тестування кожен сценарій виконується з різною кількістю ітерацій – 1000, 5000 та 10000 запусків. Такий підхід дозволяє створити контрольоване навантаження на систему, поступово збільшуючи кількість операцій і фіксуючи реакцію контейнера на зростання обсягу викликів. Завдяки цьому вдається виявити залежність продуктивності від масштабу навантаження та побачити, як DI-контейнери поводяться при роботі в умовах, наближених до реальних.

BenchmarkDotNet автоматично виконує серію прогрівів (warmup) перед основними замірами, що забезпечує стабільність результатів і зменшує вплив факторів, пов’язаних із першим запуском коду або JIT-компіляцією. Це дозволяє зосередитись на аналізі чистої продуктивності виконання операцій.

Окрім цього, в дослідженні використовуватиметься dotMemory – інструмент для аналізу використання пам’яті. Його застосування дозволить виявити потенційні проблеми, такі як витоки пам’яті чи надмірне споживання ресурсів. DotMemory буде використовуватися для визначення кількості створених об’єктів, їхнього життєвого циклу та обсягів пам’яті, які вони займають. Особливу увагу буде приділено виявленню ситуацій, коли Scoped-залежності використовуються в Singleton-сервісах, що може спричинити витоки пам’яті або некоректну поведінку системи [24].

Також буде використовуватися dotTrace для аналізу часу виконання операцій Dependency Injection-контейнерів. За допомогою dotTrace будуть виявлені "гарячі точки" у коді, що сповільнюють резолюцію залежностей, а також досліджено загальний час виконання операцій, таких як створення об'єктів або виконання багаторівневих залежностей [25]. Аналіз викликів методів і стеків дозволить визначити, які аспекти роботи DI-контейнера викликають найбільші затримки, і забезпечить рекомендації щодо їх оптимізації. Цей інструмент також допоможе порівняти швидкодію контейнерів, таких як MS DI, Autofac, Ninject та DryIoc, в умовах високого навантаження.

Незважаючи на те, що BenchmarkDotNet не надає прямої інформації про кількість створених об'єктів чи точний час резолюції залежностей, ці аспекти опосередковано фіксуються через такі метрики, як час виконання, кількість спрацювань GC та обсяг виділеної пам’яті. Їх аналіз дозволяє оцінити загальне навантаження на систему та ефективність використання ресурсів.

Загальна мета проєктування бази даних і тестового середовища полягає у створенні реалістичних, хоча й контрольованих умов для перевірки продуктивності DI-контейнерів. Проведені тести дозволять виявити оптимальні конфігурації для кожного контейнера, оцінити їхню стабільність і продуктивність

у різних умовах, а також зробити висновки щодо компромісів між складністю конфігурації, споживанням ресурсів та швидкодією.

Вибір діапазону від 1000 до 10000 ітерацій для кожного сценарію не є випадковим – такі значення забезпечують достатнє навантаження для виявлення особливостей поведінки DI-контейнерів без надмірного впливу зовнішніх факторів. У межах мікробенчмаркінгу цього достатньо, щоб проявилися закономірності у роботі збирача сміття, накопиченні пам’яті, ефективності резолюції залежностей і загальній швидкодії.

Крім того, саме в цьому діапазоні BenchmarkDotNet забезпечує баланс між тривалістю тесту і статистичною достовірністю результатів. Тестові серії в 1000–10000 запусків дозволяють виявити зміни в поведінці контейнера при зростанні навантаження, не затягуючи процес бенчмаркінгу до надмірної тривалості, що особливо важливо при порівнянні кількох конфігурацій.

Таким чином, вибраний підхід дозволяє імітувати характерні патерни навантаження, властиві реальним системам, при цьому залишаючись у межах керованого і відтворюваного середовища.

4.6 Обмеження існуючих підходів при роботі з DI-контейнерами

Незважаючи на розмаїття DI-контейнерів, існують певні обмеження, які впливають на їхню продуктивність, ефективність використання ресурсів та загальну зручність застосування. Ці недоліки можуть суттєво впливати на вибір контейнера для конкретного проєкту.

Одним із критичних аспектів є продуктивність резолюції залежностей, яка суттєво варіюється між різними контейнерами. Наприклад, Ninject, завдяки використанню рефлексії та динамічної генерації проксі, має порівняно повільну швидкість резолюції. У високонавантажених системах це може спричинити значні затримки, особливо коли одночасно обробляються великі графи залежностей. Натомість контейнери, такі як DryIoc, оптимізовані для швидкодії, повинні демонструвати кращі результати в таких умовах.

Ще одним важливим аспектом є споживання пам’яті. Архітектура деяких контейнерів передбачає використання складних механізмів кешування або зберігання метаданих про залежності, що збільшує обсяг використовуваних ресурсів. Це може стати критичним фактором у системах з обмеженою кількістю пам’яті, де оптимізація ресурсів є ключовою.

Складність конфігурації також може впливати на продуктивність і стабільність системи. Потужні та гнучкі контейнери, такі як Autofac, надають широкі можливості налаштування, проте вимагають від розробника високого рівня знань. Помилки під час налаштування можуть призвести до непередбачуваної поведінки системи або навіть до її збоїв. Тому розробникам, які працюють із такими контейнерами, необхідно приділяти особливу увагу тестуванню конфігурацій.

Ще одним обмеженням є відсутність єдиних стандартів для DI-контейнерів у .NET. Хоча вбудований контейнер MS DI пропонує базовий функціонал і встановлює деякий стандарт, інші контейнери часто мають несумісні API та підходи до конфігурації. Це може створювати додаткові труднощі під час інтеграції сторонніх бібліотек або міграції між контейнерами.

Використання різних життєвих циклів об'єктів також може впливати на продуктивність системи. Неправильне налаштування або використання цих циклів може призводити до неефективності. Наприклад, надмірне використання Transient-об'єктів може викликати підвищене навантаження на процесор через часте створення екземплярів. Неправильно налаштовані Singleton-об'єкти можуть спричиняти проблеми з потокобезпекою, якщо вони зберігають стан. У свою чергу, Scoped-об'єкти, використані поза межами визначеної області, можуть призводити до витоків пам’яті, якщо контекст не очищається коректно.

Таким чином, вибір DI-контейнера та його налаштування повинні враховувати особливості архітектури, продуктивність, зручність конфігурації та відповідність вимогам проєкту. Ретельний аналіз цих факторів допоможе мінімізувати ризики та підвищити ефективність розроблення програмного забезпечення.

4.7 Гіпотетичні шляхи покращення продуктивності

Для оптимізації роботи Dependency Injection (DI) контейнерів запропоновано низку підходів, спрямованих на підвищення їхньої продуктивності, ефективності використання ресурсів та адаптації до реальних умов. Ці вдосконалення базуються на аналізі ефективності DI-контейнерів у різних сценаріях, зокрема в умовах високонавантажених систем і мікросервісних архітектур.

Одним із ключових напрямів покращення є оптимізація життєвих циклів об'єктів, що реєструються в DI-контейнері. Важливо правильно розподілити ролі між сервісами з урахуванням їхніх вимог до стану та ізоляції:

– безстанові сервіси слід реєструвати як Transient або Singleton, залежно від потреб у доступності;

– сервіси зі станом рекомендується реєструвати як Scoped для забезпечення ізольованого контексту.

Для оцінки загальної ефективності DI-контейнерів було розроблено формулу, яка враховує комплекс метрик, що описують продуктивність системи, споживання ресурсів і швидкодію. Ефективність розраховується за формулою 4.1:

$$E = \frac{T_r \times P}{R_{med} + \alpha \times C_{alloc} + \beta \times GC_{total} + \gamma \times \Delta} \tag{4.1}$$

де E – ефективність роботи DI-контейнера,

T_r – пропускна здатність системи (запитів за секунду),

P – коефіцієнт стабільності, що враховує частку запитів, оброблених без затримок,

R_{med} – медіанний час резолюції (ms),

C_{alloc} – спожите пам'яті (MB),

GC_{total} – сумарна кількість зборів сміття ($Gen_0 + Gen_1 + Gen_2$),

Δ – показник латентності ($P_{95} - Median$),

α, β, γ – вагові коефіцієнти.

Ця формула дозволяє отримати універсальну метрику, що поєднує швидкодію та ефективність використання ресурсів. Наприклад, навіть якщо DI-контейнер забезпечує високий час відгуку, але при цьому витрачає значні обсяги пам’яті, його ефективність буде нижчою в порівнянні з більш оптимальними рішеннями.

На основі проведених теоретичних досліджень сформульовано загальні рекомендації щодо використання DI-контейнерів:

- для невеликих проєктів рекомендується використовувати MS DI через його простоту та інтеграцію в .NET Core;
- Autofac є оптимальним вибором для великих монолітних або багатошарових застосунків із складною логікою;
- Ninject доцільно застосовувати у проєктах, що вимагають аспектно-орієнтованого програмування (AOP) або гнучкої модульної конфігурації;
- DryIoc варто використовувати в умовах високонавантажених мікросервісних систем, де важлива мінімізація часу резолюції залежностей і економія пам’яті.

Пропоновані поліпшення підвищують гнучкість і ефективність застосунків, дозволяючи розробникам точніше налаштовувати DI-контейнери відповідно до потреб системи. Впровадження формули ефективності забезпечує нову метрику для комплексного оцінювання продуктивності, дозволяючи враховувати не лише час виконання операцій, але й стабільність і економічність використання ресурсів.

Запропоновані гіпотези та формула ефективності потребують практичної перевірки в реальних умовах. У подальшому дослідженні буде описано результати експериментів, які дозволять підтвердити чи спростувати ці теоретичні припущення.

5 ОПИС ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ

5.1 Розробка тестового застосунку

Для проведення експериментального дослідження було створено спеціалізований тестовий застосунок. Вибір тематики для цього застосунку припав на сферу eCommerce. Такий вибір зумовлений, перш за все, високою популярністю та затребуваністю цього напрямку в сучасній індустрії програмного забезпечення. Застосунки електронної комерції, як правило, характеризуються складною логікою, високим рівнем взаємодії з базами даних та різноманітністю функціональних вимог. Саме ці особливості роблять їх ідеальними кандидатами для перевірки ефективності використання різних Dependency Injection (DI) контейнерів у .NET-застосунках.

Тестовий застосунок складається з чотирьох фундаментальних модулів, які чітко розділяють відповідальності (див. рис. 5.1):

- Domain. Містить основні бізнес-сутності та правила. У цьому шарі визначено сутності, які імітують процеси електронної комерції, такі як продукти, виробники, типи продуктів, категорії, замовлення тощо;
- Application. Містить логіку застосунку, реалізовану за допомогою патерну CQRS. Команди (наприклад, CreateProductCommand) відповідають за зміну стану системи (додавання нових продуктів), а запити (наприклад, GetProductListQuery) – за отримання даних без змін стану;
- Infrastructure. Відповідає за взаємодію з базою даних та зовнішніми ресурсами. Для доступу до бази даних використовується Entity Framework Core із використанням SQL Server. Також у цьому шарі реалізовано загальний репозиторій, який забезпечує стандартизований доступ до даних та зменшує кількість повторюваного коду;
- API. Забезпечує зовнішній інтерфейс застосунку, побудований на ASP.NET Core.

Окрему увагу під час розробки тестового застосунку було приділено процесу наповнення бази даних тестовими даними. Першочерговою задачею було

створення не просто великого, але й максимально реалістичного набору даних, який би відповідав реальним умовам функціонування застосунків електронної комерції.

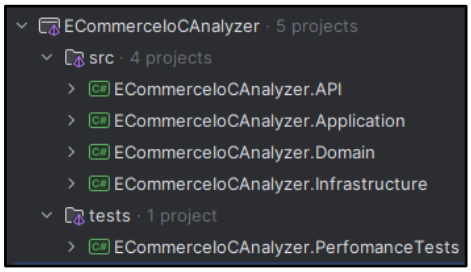

Рисунок 5.1 – Структура тестового застосунку (рисунок виконаний самостійно)

Після аналізу різних інструментів було обрано бібліотеку Bogus, яка зарекомендувала себе як один із найбільш ефективних інструментів для генерації фіктивних, але при цьому реалістичних даних. Bogus дозволяє створювати дані, які дуже близькі до тих, що використовуються в реальних системах, з урахуванням численних атрибутів, таких як імена, адреси, назви продуктів, описи товарів, ціни, характеристики виробників та інші елементи, що забезпечує якісну імітацію реальних даних.

Було згенеровано близько 200 000 записів, саме така велика кількість даних була необхідною для створення суттєвого навантаження на систему, щоб забезпечити виявлення реальних характеристик продуктивності DI-контейнерів. Це дозволило максимально наблизити умови експерименту до промислових реалій, де застосунки зазвичай працюють із великими обсягами інформації, що є критичним фактором для прийняття правильних архітектурних рішень.

Окрім кількості записів, значну увагу було приділено складності та різноманітності самих даних. Метою було не тільки створення великого набору даних, а й забезпечення його різноманітності, включаючи численні категорії, типи продуктів, варіанти їхньої доступності та різноманітні виробники. Це дозволило отримати глибше розуміння того, як різні DI-контейнери поводять себе в умовах, коли застосунок виконує складні фільтрації, пошуки, сортування та агреговані запити з множинними зв’язками між сутностями. Отже, отримані результати є більш повноцінними та значущими для реального застосування. Завдяки складній

структурі тестових даних експериментальні умови наблизилися до тих, що виникають у реалних системах.

Вибір бібліотек MediatR, Entity Framework Core та Bogus був зумовлений їхньою зручністю та широким застосуванням у .NET-розробці. MediatR спростив реалізацію патерна CQRS, зробивши код структурованим і зрозумілим. Entity Framework Core було використано завдяки його хорошій підтримці, продуктивності та інтеграції з екосистемою .NET. Бібліотека Bogus дозволила швидко згенерувати реалістичні тестові дані, що позитивно вплинуло на якість експерименту.

Таким чином, у процесі проєктування та реалізації тестового застосунку було враховано всі ключові аспекти, необхідні для створення стабільної, гнучкої та максимально наближеної до реальних умов системи. Ретельно спроєктована архітектура, налаштоване управління залежностями та варіативність у конфігурації життєвих циклів сервісів створили умови для проведення об’єктивного та точного аналізу продуктивності. Гнучкість у виборі контейнерів DI, ізольованість бізнес-логіки від інфраструктури та продумана організація доступу до даних дозволяють не лише ефективно проводити серію експериментів, а й гарантують повторюваність результатів у змінюваних умовах навантаження. Усе це робить розроблений застосунок надійною платформою для поглибленого дослідження ефективності Dependency Injection у сучасних .NET-системах.

5.2 Розробка тестової інфраструктури для проведення експериментів

Для ефективного порівняння впливу вибору різних Dependency Injection-контейнерів на продуктивність .NET-застосунків було створено спеціалізовану тестову інфраструктуру. Її основною метою було забезпечення гнучкості, повторюваності та достовірності результатів. З цією метою розроблено окремий проєкт – PerfomanceTests, призначений саме для дослідження продуктивності.

Під час створення інфраструктури було відібрано кілька ключових бібліотек з високою репутацією у .NET-спільноті, які гарантують точність і стабільність результатів. Основним інструментом вимірювання продуктивності став

BenchmarkDotNet – загальновизнаний стандарт для високоточних бенчмарків, що підтримує метрики середнього часу виконання, обсягу пам’яті та кількості спрацювань сміттєзбірника. Для логування використовувався Serilog, а для імітації кешування – Microsoft.Extensions.Caching.Memory.

Ключовим архітектурним рішенням, що підвищило гнучкість інфраструктури, стало впровадження інтерфейсу IContainerAdapter, який забезпечив повну абстракцію від конкретних реалізацій DI-контейнерів. Це дало змогу легко змінювати конфігурації або інтегрувати нові контейнери без змін у тестовій логіці. Для кожного контейнера (Microsoft DI, Autofac, DryIoc, Ninject) реалізовано власний адаптер. Наприклад, MicrosoftDIAdapter (див. рис. 5.2) гнучко реєструє сервіси з різними життєвими циклами (Transient, Scoped, Singleton), що дозволяє швидко перемикатися між конфігураціями та прискорює проведення експериментів.

```
public override IContainerAdapter Register<TService, TImplementation>(
    Microsoft.Extensions.DependencyInjection.ServiceLifetime lifetime)
{
    return MeasureOperation($"Register_{typeof(TService).Name}_{lifetime}", operation: () =>
    {
        switch (lifetime)
        {
            case Microsoft.Extensions.DependencyInjection.ServiceLifetime.Transient:
                _services.AddTransient<TService, TImplementation>();
                break;
            case Microsoft.Extensions.DependencyInjection.ServiceLifetime.Scoped:
                _services.AddScoped<TService, TImplementation>();
                break;
            case Microsoft.Extensions.DependencyInjection.ServiceLifetime.Singleton:
                _services.AddSingleton<TService, TImplementation>();
                break;
            default:
                throw new ArgumentOutOfRangeException(nameof(lifetime), lifetime, null);
        }

        return this;
    });
}
```

Рисунок 5.2 – Фрагмент адаптера Microsoft DI з підтримкою життєвих циклів сервісів (рисунок виконаний самостійно)

Наступним важливим компонентом тестової інфраструктури стали самі тести (Benchmarks). Для організації тестування було використано базовий клас

BenchmarkBase, який визначає загальну логіку роботи бенчмарків та конфігурації для тестування. Кожен сценарій тестування (логування, кешування, CRUD-операції) має власні класи-спадкоємці, які реалізують специфічну тестову логіку та різні сценарії навантаження.

Кожен окремий сценарій тестування (наприклад, логування, кешування, CRUD-операції) отримав власний клас-спадкоємець, що дозволяє максимально гнучко та просто розширювати функціональність і додавати нові тести в майбутньому. Наприклад, клас LoggingBenchmarks дозволяє легко перевірити продуктивність різних контейнерів під різними навантаженнями, змінюючи параметри за допомогою простих атрибутів. Такий підхід забезпечує надзвичайно просту модифікацію сценаріїв та значно скорочує час на підготовку і запуск експериментів.

Для забезпечення зручності запуску тестів було створено просту й зрозумілу точку входу, яка дозволяє швидко обирати тип тесту (CRUD, логування, кешування або всі одразу). Такий підхід спрощує проведення експериментів як у ручному режимі, так і в режимі автоматизованих сценаріїв безперервної інтеграції (CI/CD).

Таким чином, створена тестова інфраструктура повністю відповідає сучасним вимогам до проведення експериментальних досліджень продуктивності програмного забезпечення. Вона забезпечує не лише високу точність замірів, але й виняткову гнучкість та простоту в налаштуванні й розширенні. Завдяки ретельно продуманій архітектурі, універсальному підходу до роботи з контейнерами та використанню найкращих інструментів для збору та аналізу даних, ця інфраструктура стала потужним засобом для глибокого дослідження впливу вибору Dependency Injection-контейнерів на продуктивність .NET-застосунків.

5.3 Проведення експерименту

Для забезпечення максимальної достовірності й точності результатів, отриманих у ході експериментального дослідження, було проведено серію

комплексних тестів. Процес виконання експериментів був ретельно спланованим та мав системний характер, що вимагало значних витрат часу, зусиль та обчислювальних ресурсів. Основною метою цього етапу було отримати достатній обсяг даних, які дозволяють глибоко й об'єктивно оцінити вплив використання різних Dependency Injection-контейнерів на продуктивність .NET-застосунків.

Тестування здійснювалося на сучасному ноутбуці з сучасними компонентами, що забезпечують стабільність і точність отриманих результатів. Пристрій мав наступні характеристики: операційна система Windows 11, платформа .NET 9, процесор Intel Core i9-14900HX із 24 фізичними ядрами (32 логічними потоками) з базовою тактовою частотою 1.6 ГГц та максимальною частотою Turbo Boost до 5.8 ГГц, а також 32 ГБ оперативної пам'яті DDR5. Такі обчислювальні ресурси дозволили мінімізувати можливі спотворення результатів експерименту через нестачу ресурсів чи додаткові фактори, такі як фонова активність інших програм або сервісів.

Експериментальний процес мав послідовний та організований характер. Спочатку було обрано окремий тип тестування (наприклад, логування), після чого почергово запускалися тести для кожного DI-контейнера (Microsoft DI, Autofac, DryIoc, Ninject). Вибір такого послідовного підходу був обґрунтований необхідністю чітко контролювати хід експерименту та гарантувати чистоту отриманих даних. Проведення тестів окремо для кожного контейнера суттєво полегшувало процес моніторингу й аналізу результатів, знижувало ризик помилок та поліпшувало загальну прозорість експериментального процесу.

Для отримання кількісних та якісних характеристик продуктивності використовувався спеціалізований інструмент BenchmarkDotNet, що є визнаним стандартом серед .NET-розробників для проведення високоточних вимірювань продуктивності. Особливістю цього інструмента є автоматичне виконання етапу прогрівання (warmup) перед основними замірами, що забезпечує досягнення стабільності роботи програми. BenchmarkDotNet також автоматично визначає оптимальну кількість ітерацій для досягнення високої точності результатів, внаслідок чого деякі окремі тести могли займати значний час, іноді досягаючи

навіть восьми годин. Такий тривалий час виконання свідчить про високий рівень ретельності й точності проведених експериментів та глибину отриманих даних.

Під час кожного тестового запуску обиралися різні налаштування життєвих циклів сервісів (Transient, Scoped, Singleton), після чого запускалася тестова процедура й виконувалися автоматичні заміри продуктивності. Після завершення кожного етапу тестування результати зберігалися для подальшого детального порівняльного аналізу, що дозволяло чітко розмежовувати вплив різних факторів на загальну продуктивність застосунку.

Таким чином, проведений експеримент був комплексним, трудомістким та вимагав значних часових витрат. Його послідовна організація забезпечила високу повторюваність і надійність результатів, що є критично важливим для подальших рекомендацій щодо використання DI-контейнерів залежно від характеру задач і навантажень .NET-застосунків. В результаті вдалося отримати ґрунтовні дані, які створили основу для об'єктивних висновків та рекомендацій стосовно вибору оптимальних підходів до організації управління залежностями в реальних проєктах.

5.4 Аналіз отриманих результатів

Після завершення експериментальної фази було зібрано велику кількість даних, що відображають продуктивність різних DI-контейнерів у рамках практичних сценаріїв, характерних для .NET-застосунків. Для коректного аналізу ці результати були агреговані, нормалізовані та перетворені в єдину структуру за допомогою скриптів, що дозволило зосередитися саме на ключових показниках ефективності.

Серед таких показників особливу увагу було приділено середньому часу виконання (Mean execution time), який визначає швидкодію контейнера при резолюції залежностей. Ця метрика є критично важливою у високонавантажених системах, де навіть незначне збільшення часу виконання при багаторазових викликах може призвести до затримок, втрати відгуку або зниження продуктивності всього застосунку. Не менш важливим параметром є обсяг

виділеної пам’яті (Allocated memory), що демонструє, наскільки ощадливо контейнер працює з ресурсами. У випадках масового створення залежностей, паралельного виконання запитів або довготривалого функціонування, надмірне споживання пам’яті може спричинити деградацію продуктивності або навіть системні збої.

Для отримання більш узагальненої оцінки було також запроваджено інтегральну метрику ефективності, яка поєднує часові та ресурсні аспекти в єдине числове значення. Такий підхід дозволяє не лише оцінити окремі характеристики продуктивності, а й порівняти контейнери з точки зору їх загальної придатності до використання у продуктивному середовищі, де важлива як швидкодія, так і економність.

Аналіз базується на значному обсязі ітерацій, проведених для різних типів життєвих циклів об’єктів, що дозволяє не лише виявити середні показники, а й відстежити поведінку контейнерів при зростанні навантаження. Це має особливу цінність у контексті розробки enterprise-систем, які працюють в умовах неперервного росту навантаження та високих вимог до стабільності.

Загалом, отримані метрики не є абстрактними статистичними даними – вони виступають у ролі практичного інструменту, що дає змогу зробити обґрунтований вибір на користь того чи іншого DI-рішення залежно від архітектурних вимог системи: стабільності, ефективності споживання ресурсів або максимальної швидкодії при частих зверненнях.

5.4.1 Аналіз метрик продуктивності для сценарію логування

Сценарій логування було обрано першим для аналізу, оскільки він є не лише базовим, але й критично важливим для більшості .NET-застосунків. Саме логування забезпечує спостережуваність, підтримку стабільності, аналіз помилок і моніторинг роботи системи в реальному часі. Його особливість полягає у високій частоті викликів при мінімальній складності залежностей – логери використовуються майже у кожному сервісі, але самі по собі є легковагими

об’єктами. Це створює унікальне навантаження на DI-контейнер: з одного боку – мільйони звернень, з іншого – простота графа залежностей.

У ході експерименту були виміряні ключові показники продуктивності для чотирьох контейнерів (Microsoft DI, Autofac, DryIoc, Ninject) у трьох режимах життєвого циклу (Singleton, Scoped, Transient) та на трьох рівнях навантаження (1000, 5000, 10000 ітерацій). Це дозволило не лише оцінити середню продуктивність, а й простежити динаміку деградації або стабільності при масштабуванні навантаження. Візуалізація залежності обсягу виділеної пам’яті від кількості ітерацій для кожного з контейнерів подана на рисунку 5.3.

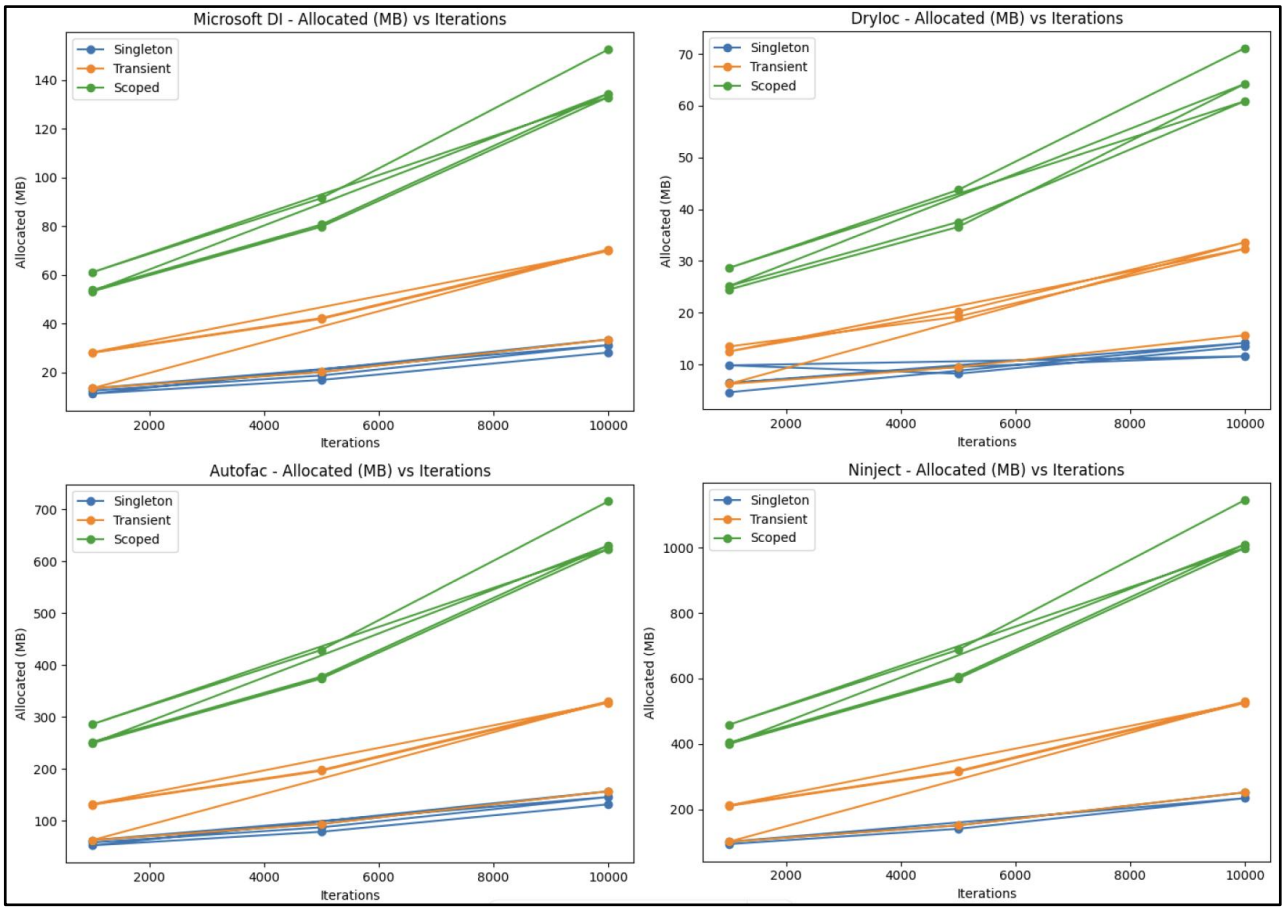

Рисунок 5.3 – Залежність об’єму виділеної пам’яті від кількості ітерацій для сценарію логування (рисунок виконаний самостійно)

Аналіз графіка демонструє цілком передбачувану закономірність: життєвий цикл Singleton стабільно забезпечує найменше споживання пам’яті. Це логічно, оскільки об’єкт створюється лише один раз і повторно використовується протягом усього життєвого циклу програми. Transient передбачає створення

нового екземпляра при кожному зверненні, що призводить до зростання обсягу пам’яті, але все ще в межах помірного споживання. Найвищі витрати демонструє Scoped – зважаючи на створення окремого об’єкта на кожну область видимості, цей режим є найменш доцільним для інфраструктурних сервісів, зокрема логерів.

Особливо негативно вирізняється контейнер Ninject, який навіть при незначному навантаженні демонструє непропорційно високі витрати пам’яті. Його поведінка є симптоматичною: велика частина внутрішніх механізмів побудована на рефлексії та runtime-генерації, що не тільки уповільнює роботу, а й провокує зростання алокацій. Autofac також виявляє тенденцію до зростання обсягу пам’яті зі збільшенням навантаження, хоч і менш виражено.

На цьому фоні DryIoc та Microsoft DI демонструють прогнозовану та економну поведінку. Вони мають стабільно низькі показники алокацій, що робить їх придатними для систем із частим логуванням або високою паралельністю. Серед них DryIoc виділяється ще нижчими витратами, що свідчить про глибоку оптимізацію механізмів створення залежностей навіть у найчастіших сценаріях.

Для більш зручного та детального вивчення поведінки контейнерів за показником використання пам’яті було додатково побудовано стовпчастий графік (рис. 5.4), який демонструє ті ж самі дані у більш зручному форматі, дозволяючи краще простежити відмінності між контейнерами за різних навантажень.

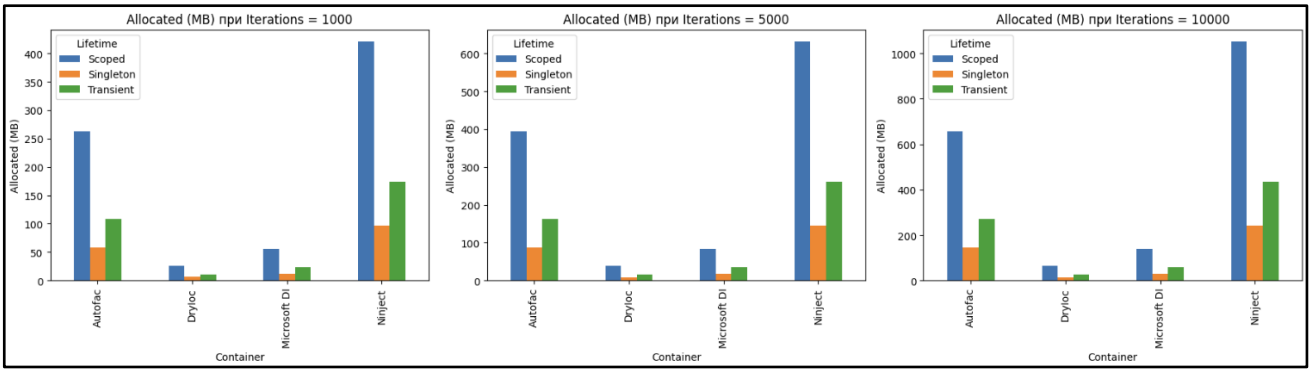

Рисунок 5.4 – Залежність об’єму виділеної пам’яті від кількості ітерацій для сценарію логування (рисунок виконаний самостійно)

Розглядаючи результати щодо використання пам'яті, о особливо помітно неефективність контейнера Ninject, який навіть на низьких навантаженнях

витрачає значно більше ресурсів, ніж решта контейнерів. Autofac також демонструє значне зростання витрат пам’яті зі збільшенням кількості ітерацій, що є наслідком більш складних внутрішніх механізмів цього контейнера.

DryIoc та Microsoft DI у цьому контексті виглядають значно ефективнішими. Вони демонструють помірні, стабільні й прогнозовані витрати пам’яті, що робить їх придатними для роботи навіть у сценаріях із високими вимогами до ресурсів. Однак навіть між цими двома лідерами DryIoc виявляється дещо економнішим рішенням, що додатково підтверджується іншими метриками, зокрема часовими характеристиками. Середній час виконання при різній кількості ітерацій для кожного контейнера та життєвого циклу наведено на рисунку 5.5.

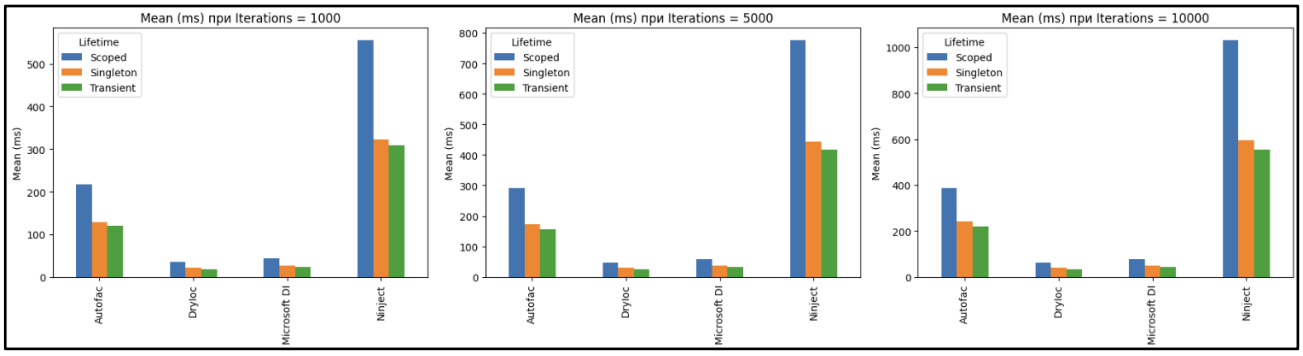

Рисунок 5.5 – Середній час виконаня при різній кількості ітерацій для сценарію логування (рисунок виконаний самостійно)

З графіка видно суттєві проблеми продуктивності контейнера Ninject, який характеризується високими витратами часу через активне використання рефлексії. Autofac демонструє середні результати, однак час виконання значно зростає зі збільшенням навантаження, що є наслідком його внутрішньої складності. Натомість DryIoc та Microsoft DI демонструють найкращі показники, маючи стабільно низький час виконання навіть за значних навантажень.

Для комплексної оцінки контейнерів застосовувалась метрика ефективності, яка враховує як часові, так і ресурсні показники, що дозволило зробити виважені висновки про загальну ефективність контейнерів (див. табл. 5.1).

Аналіз цієї таблиці демонструє очевидну перевагу DryIoc, який є найбільш ефективним серед усіх контейнерів, маючи середню ефективність 103.03 у режимі

Transient та 93.53 для Singleton. Microsoft DI посідає друге місце, демонструючи ефективність приблизно на 48-57% від DryIoc залежно від життєвого циклу. Це свідчить про те, що Microsoft DI є цілком конкурентоспроможним рішенням з огляду на його інтегрованість у стандартну .NET-інфраструктуру.

Таблиця 5.1 – Середня ефективність контейнерів залежно від життєвого циклу у сценарії логування (таблиця виконана самостійно)

Container	Efficiency	Relative (%)
Transient		
DryIoc	103.03	100
Microsoft DI	49.87	48.4
Autofac	2.07	2.01
Ninject	0.37	0.36
Scoped		
DryIoc	26.65	100
Microsoft DI	12.3	46.16
Autofac	0.52	1.94
Ninject	0.9	0.35
Singleton		
DryIoc	93.53	100
Microsoft DI	53.42	57.11
Autofac	2.34	2.5
Ninject	0.36	0.38

Значно нижчими є показники контейнерів Autofac та Ninject. Autofac має середню ефективність всього на рівні 2-3%, а Ninject взагалі менше 0.4% від DryIoc, що суттєво обмежує їхню доцільність використання в сценаріях з частим створенням і логуванням об’єктів. Для більш цілісного аналізу було розраховано й візуалізовано показник ефективності за виведеною формулою (див. рис. 5.6).

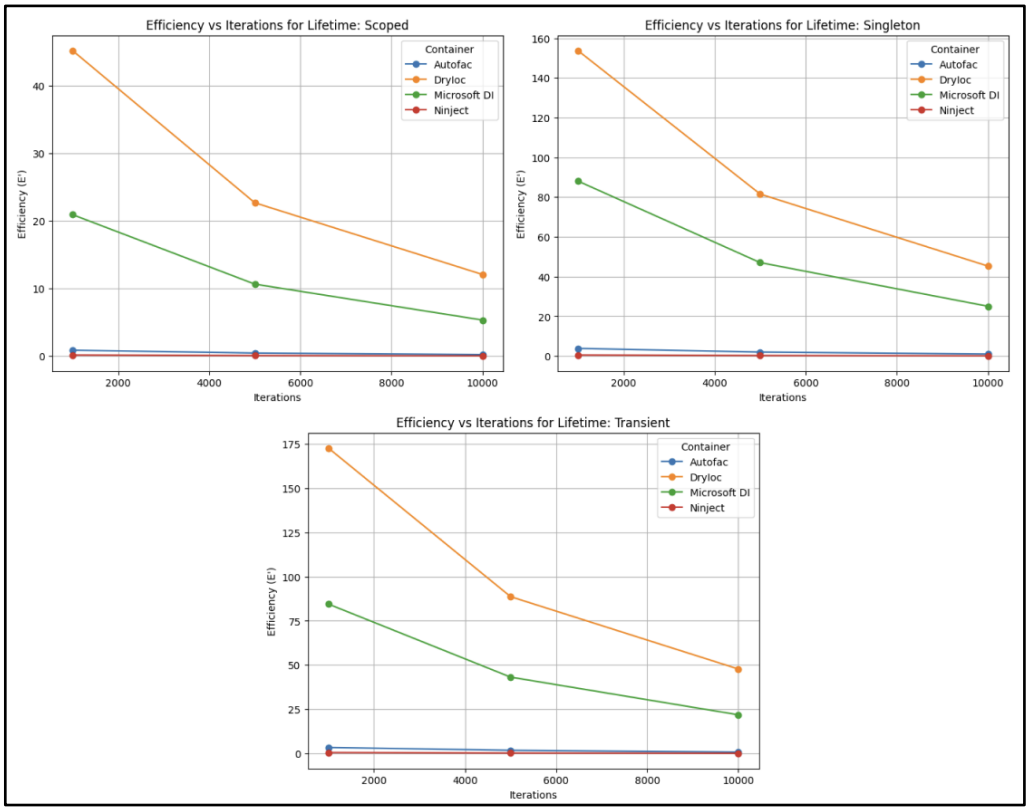

Рисунок 5.6 – Ефективність контейнерів залежно від кількості ітерацій та життєвого циклу для сценарію логування (рисунок виконаний самостійно)

Аналізуючи ці графіки, можна чітко простежити переваги DryIoc, який стабільно утримує лідерство за рівнем ефективності на всіх життєвих циклах і навантаженнях. Microsoft DI також демонструє дуже хороші показники, хоча і дещо поступається DryIoc, особливо при зростанні кількості ітерацій. Водночас Autofac і Ninject показують майже горизонтальні лінії з дуже низькою ефективністю, що ще раз вказує на вкрай неефективне використання ресурсів і неприйнятні втрати продуктивності навіть при середніх навантаженнях.

Таким чином, у сценарії логування можна рекомендувати як оптимальний варіант використання контейнерів DryIoc та Microsoft DI, причому DryIoc є безумовним лідером за показниками ефективності. Втім, враховуючи відносну новизну та не таку велику популярність цього контейнера на ринку, Microsoft DI залишається більш надійним та збалансованим рішенням для переважної більшості типових сценаріїв, особливо в умовах, коли потрібна інтегрованість зі стандартними рішеннями платформи .NET. Autofac може бути виправданий лише

в особливо складних сценаріях, логування не можна віднести до такого типу сценаріїв.

5.4.2 Аналіз метрик продуктивності для сценарію кешування

Сценарій кешування є ключовим елементом продуктивності у багатьох .NET-застосунках, особливо у високонавантажених вебсервісах, API, системах звітності та інтерфейсах з великою кількістю повторних запитів. Його основна мета – уникнути повторного виконання дорогих операцій, таких як звернення до бази даних або складних обчислень, шляхом збереження результатів у оперативній пам’яті. Саме тому ефективність роботи DI-контейнера у цьому контексті має безпосередній вплив на швидкість доступу до кешованих значень, стабільність системи та рівень використання ресурсів.

У межах цього дослідження було зібрано дані за трьома рівнями навантаження (1000, 5000, 10000 ітерацій) та по трьох типах життєвого циклу (Singleton, Scoped, Transient) для чотирьох контейнерів: Microsoft DI, Autofac, DryIoc та Ninject. Аналіз проводився за такими ключовими показниками, як середній час виконання та обсяг виділеної пам’яті.

На рисунку 5.7 зображено залежність середнього часу виконання (Mean execution time, ms) для різної кількості ітерацій у сценарії кешування.

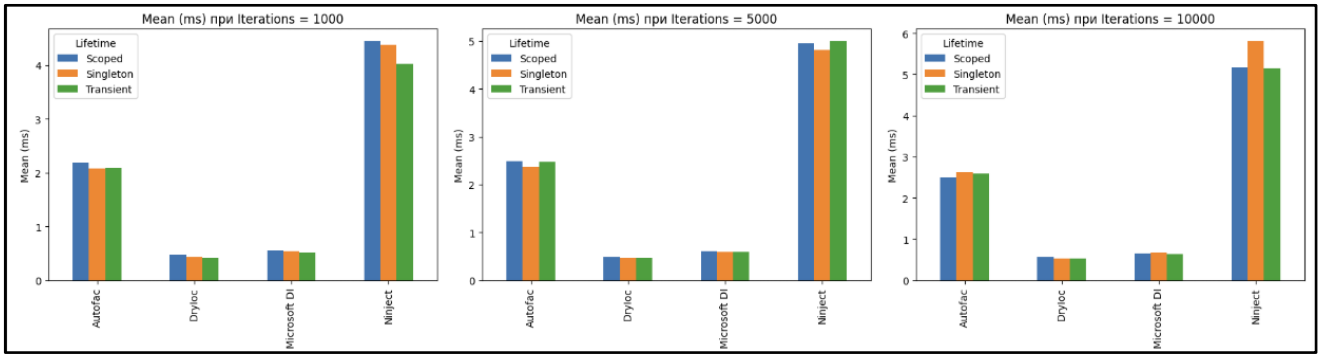

Рисунок 5.7 – Середній час виконаня при різній кількості ітерацій для сценарію кешування (рисунок виконаний самостійно)

Як показує графік, DryIoc та Microsoft DI утримують лідерство за швидкістю, демонструючи найменші значення часу у всіх режимах життєвого

циклу. Особливо стабільним виявився DryIoc, показники якого майже не змінюються навіть при 10000 ітераціях. Це свідчить про його здатність ефективно працювати у сценаріях із повторним доступом до залежностей, що є типовим для кешування. Microsoft DI теж демонструє хороші результати, дещо поступаючись DryIoc, але зберігаючи стабільність у часі. Autofac та Ninject, навпаки, відстають – особливо Ninject, який в усіх режимах показав найгіршу швидкодію. Це в черговий раз вказує на надмірні внутрішні витрати при побудові графів залежностей та використанні рефлексії.

Оцінка обсягу виділеної пам’яті для цього ж сценарію наведена окремо (див. рис. 5.8). З представлених результатів видно, що Ninject знову значно поступається іншим контейнерам, демонструючи високі та швидко зростаючі витрати пам’яті навіть на невеликих навантаженнях. Контейнер Autofac використовує помірну кількість пам’яті, але також демонструє певну тенденцію до зростання при збільшенні кількості ітерацій. Натомість DryIoc та Microsoft DI відзначаються стабільністю та передбачуваністю споживання ресурсів, що робить їх найкращими кандидатами для сценаріїв з високим навантаженням, таких як кешування.

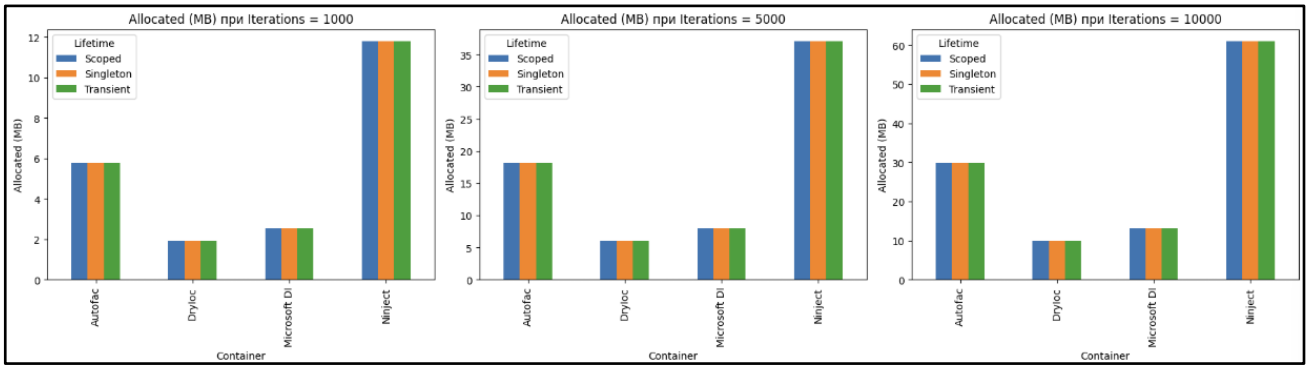

Рисунок 5.8 – Залежність об’єму виділеної пам’яті від кількості ітерацій для сценарію кешування (рисунок виконаний самостійно)

Щоб отримати узагальнену картину ефективності, була обчислена спеціальна метрика, яка враховує як час виконання, так і обсяг виділеної пам’яті (див. табл 5.2). Аналізуючи таблицю, можна констатувати значну перевагу DryIoc за всіма життєвими циклами. Microsoft DI впевнено займає друге місце,

демонструючи результати приблизно 60-65% від показників DryIoc. Це свідчить про високу конкурентоспроможність Microsoft DI, особливо враховуючи його повну інтегрованість у екосистему .NET, відсутність зовнішніх залежностей та стабільність перевіреного часом рішення.

Контейнери Autofac та Ninject мають суттєво нижчі результати – у межах лише 1-6% від ефективності DryIoc. Це обумовлено більш ресурсомісткими механізмами створення та управління об'єктами, що призводить до значних витрат як пам'яті, так і часу виконання, особливо у сценаріях з частим повторним використанням об'єктів, яким є кешування.

Таблиця 5.2 – Середня ефективність контейнерів залежно від життєвого циклу у сценарії кешування (таблиця виконана самостійно)

Container	Efficiency	Relative (%)
Transient		
DryIoc	60927.83	100
Microsoft DI	37754.53	61.97
Autofac	3588.02	5.89
Ninject	908.6	1.49
Scoped		
DryIoc	54763.01	100
Microsoft DI	35439.64	64.71
Autofac	3475.27	6.35
Ninject	835.67	1.53
Singleton		
DryIoc	59643.13	100
Microsoft DI	36270.08	60.81
Autofac	3589.98	6.02
Ninject	841.87	1.41

Варто також відзначити, що значення метрики ефективності у сценарії кешування значно перевищують аналогічні показники у сценарії логування. Це пов'язано зі специфікою самого процесу кешування, який передбачає мінімальні затрати ресурсів на повторне створення об’єктів завдяки повторному використанню результатів, що накопичуються у кеші. Відповідно, продуктивність контейнерів у цьому випадку визначається переважно ефективністю внутрішніх механізмів самого контейнера та його оптимізацією роботи з кешем. Динаміка зміни ефективності в залежності від кількості ітерацій відображена на рисунку 5.9.

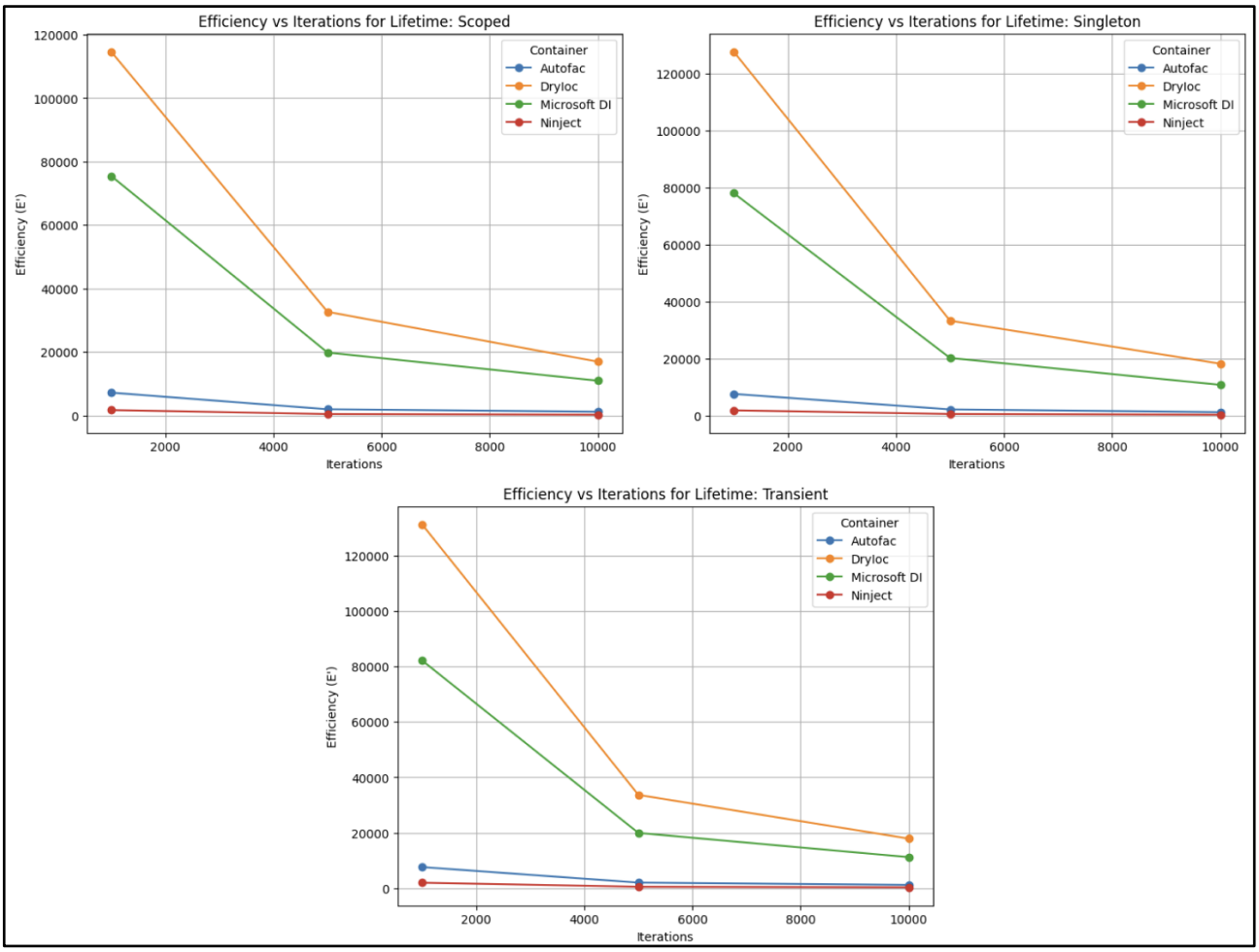

Рисунок 5.9 – Ефективність контейнерів залежно від кількості ітерацій та життєвого циклу для сценарію кешування (рисунок виконаний самостійно)

Аналіз побудованих графіків ефективності демонструє цікаву й водночас закономірну особливість: криві для різних типів життєвих циклів (Scoped, Transient, Singleton) в межах одного контейнера майже не відрізняються між

собою. На перший погляд це може здатися нетиповим – зазвичай різні режими життєвого циклу призводять до суттєвих змін у поведінці контейнера. Проте у випадку кешування така схожість є цілком очікуваною і логічною.

Справа в тому, що сам сценарій кешування по своїй природі передбачає значно меншу динаміку створення нових об’єктів. Основна частина навантаження припадає не на резолюцію нових залежностей, як у випадку CRUD або логування, а на повторне використання вже створених екземплярів. Це знижує залежність продуктивності від життєвого циклу об'єкта – незалежно від того, чи об’єкт створюється як Transient, чи реєструється як Singleton, у більшості випадків він буде повторно використовуватись із кешу. Відповідно, ключовим чинником, який формує загальну ефективність контейнера у цьому сценарії, є не стільки логіка створення, скільки ефективність самого механізму кешування, кеш-lookup, внутрішні таблиці посилань, а також економія на внутрішніх перевірках.

Це також пояснює, чому DryIoc та Microsoft DI виявляються найбільш ефективними: обидва контейнери мають добре оптимізовані шляхи доступу до кешованих залежностей, використовують мінімальну кількість рефлексії й зберігають результати резолюції у легковагих внутрішніх структурах. Вони не лише швидко повертають потрібну залежність, а й роблять це з мінімальними витратами пам’яті, що особливо помітно на великих ітераціях. Саме завдяки цим властивостям ефективність залишається високою і майже не залежить від типу життєвого циклу.

На противагу цьому, Ninject і Autofac демонструють менш ефективну поведінку, оскільки кожен запит проходить крізь більш складну систему побудови графів залежностей і не використовує кешування так агресивно або ефективно. У підсумку – незалежно від життєвого циклу, кожна резолюція має приблизно однакову вагу й час, що не дозволяє досягти помітного приросту ефективності, навіть якщо залежності мали б бути повторно використані.

Таким чином, загальна схожість результатів між різними життєвими циклами не є ознакою відсутності впливу, а радше – індикатором оптимізованої логіки повторного використання залежностей, що характерна для сучасних DI-

контейнерів. Вона також підкреслює важливість обирати не просто контейнер, який “швидко створює об’єкти”, а той, який грамотно керує їх повторним використанням, що й становить суть ефективного кешування.

У контексті цього сценарію вибір контейнера – це вибір між внутрішньою оптимізацією доступу до вже збережених об’єктів. DryIoc у цьому плані демонструє беззаперечне лідерство. Microsoft DI, хоч і поступається за абсолютними показниками, залишається надійним вибором з хорошим балансом між швидкодією, простотою й інтеграцією. У той час як Ninject та Autofac, через глибші витрати на кожен цикл резолюції, виглядають менш придатними до задач, де кешування використовується інтенсивно й часто.

5.4.3 Аналіз метрик продуктивності для сценарію створення

Одним з найчастіших сценаріїв, з якими стикаються .NET-застосунки, є операції створення сутностей у базі даних. На відміну від простіших задач, таких як логування або кешування, сценарій створення сутностей містить цілий комплекс дій: від резолюції залежностей і створення об’єктів до запуску транзакцій баз даних, перевірок валідності та роботи ORM-інструментів. Висока частота таких операцій у реальних проєктах вимагає максимальної ефективності, стабільності та прогнозованості роботи DI-контейнерів, щоб забезпечити комфортний користувацький досвід навіть при високому навантаженні.

Після проведення серії експериментів результати були агреговані та ретельно проаналізовані за допомогою Python, аналогічно до попередніх розділів. Як і раніше, для аналізу було виділено кілька ключових показників, а саме: об'єм виділеної пам'яті, час виконання операцій та метрика ефективності.

Після серії експериментів було отримано дані щодо середнього часу виконання операцій при різній кількості ітерацій для кожного контейнера та типу життєвого циклу (див. рис. 5.10). Найбільше занепокоєння викликає поведінка Ninject та Autofac, особливо у режимі Singleton, де спостерігається різке зростання споживання пам’яті. Це наслідок збереження об’єктів у пам’яті протягом усього часу життя програми. У продакшн-середовищі подібна поведінка може мати

серйозні наслідки – від поступового зниження швидкодії до аварійних збоїв при довготривалому використанні.

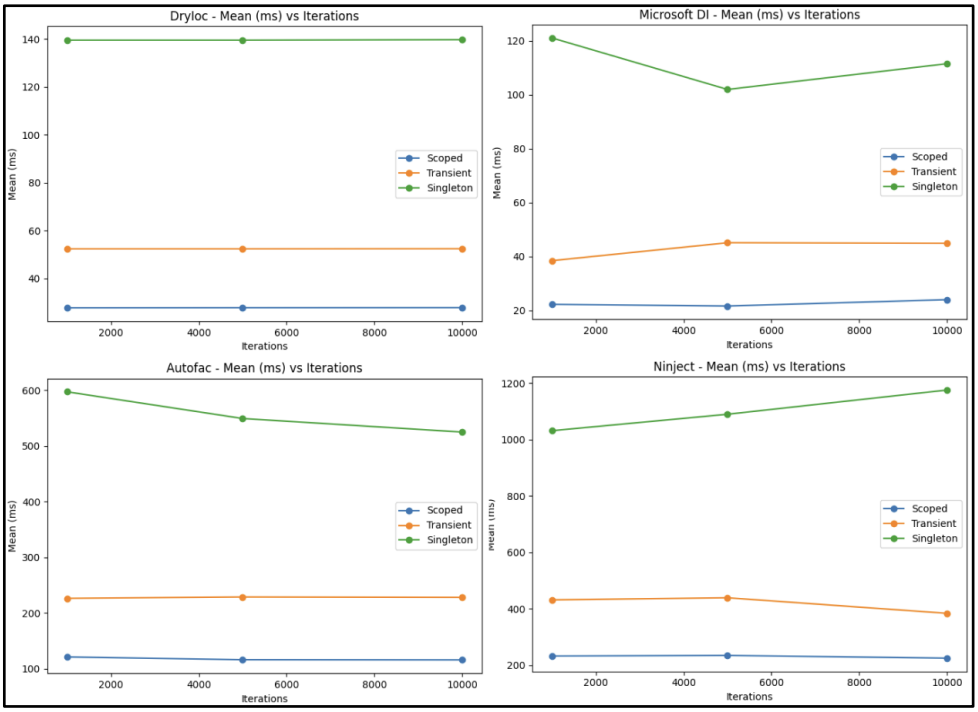

Рисунок 5.10 – Середній час виконаня при різній кількості ітерацій для сценарію створення (рисунок виконаний самостійно)

Натомість DryIoc, як і очікувалося, демонструє високу продуктивність і залишається на позиції одного з найефективніших рішень. Однак справжнім сюрпризом став Microsoft DI, який у цьому сценарії зумів обійти навіть DryIoc за швидкістю створення об’єктів, особливо в режимах Scoped і Transient. Це ставить під сумнів поширене уявлення, що Microsoft DI є базовим, а отже – менш оптимізованим. Навпаки, видно, що стандартний контейнер .NET має добре пропрацьовану архітектуру і підходить для практичних задач.

Стовпчаста візуалізація результатів дозволяє чітко порівняти обсяг використаної пам’яті між контейнерами (див. рис. 5.11). Тут легко помітити перевагу DryIoc та Microsoft DI, які демонструють значно нижчі витрати пам’яті, стабільну поведінку та меншу залежність від типу життєвого циклу. У контексті CRUD-сценаріїв, де в межах одного запиту може створюватися велика кількість об’єктів, така економність є важливою перевагою. Вона забезпечує можливість

масштабування систем без зростання ризику витоку пам’яті або деградації продуктивності.

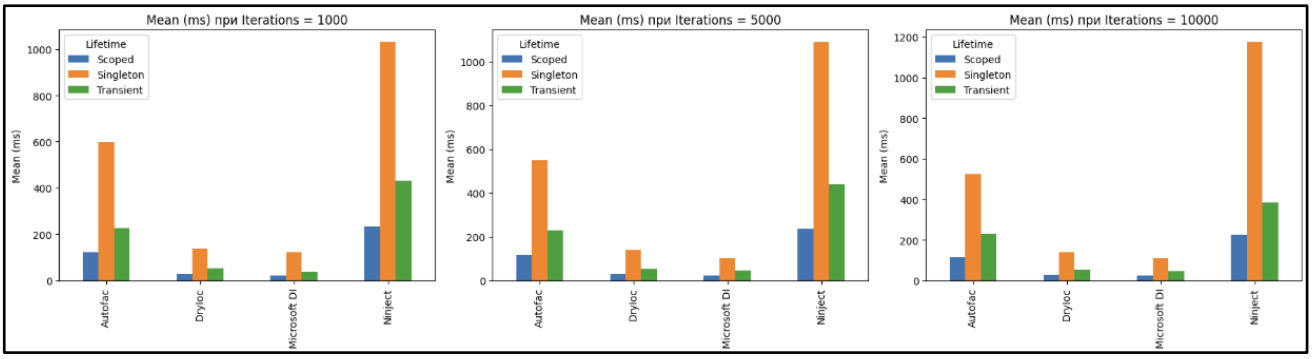

Рисунок 5.11 – Середній час виконаня при різній кількості ітерацій для сценарію створення (рисунок виконаний самостійно)

Додатковий графік розподілу пам’яті залежно від кількості ітерацій (див. рис. 5.12) демонструє лінійну залежність обсягу виділеної пам’яті (Allocated memory, MB) від навантаження для кожного з контейнерів. Тут також яскраво проявляються обмеження Ninject та Autofac, де агресивне використання рефлексії й динамічної побудови графів залежностей створює суттєве навантаження на пам’ять. Це підкреслює, наскільки важливою є внутрішня архітектура контейнера для поведінки системи під навантаженням.

DryIoc і Microsoft DI, навпаки, демонструють передбачувану динаміку навіть при великій кількості ітерацій, що дозволяє системам масштабуватися без втрати стабільності. Для високонавантажених застосунків, де кожна мілісекунда й мегабайт мають значення, така поведінка критично важлива для підтримки продуктивності, зменшення затримок і збереження бізнес-результатів.

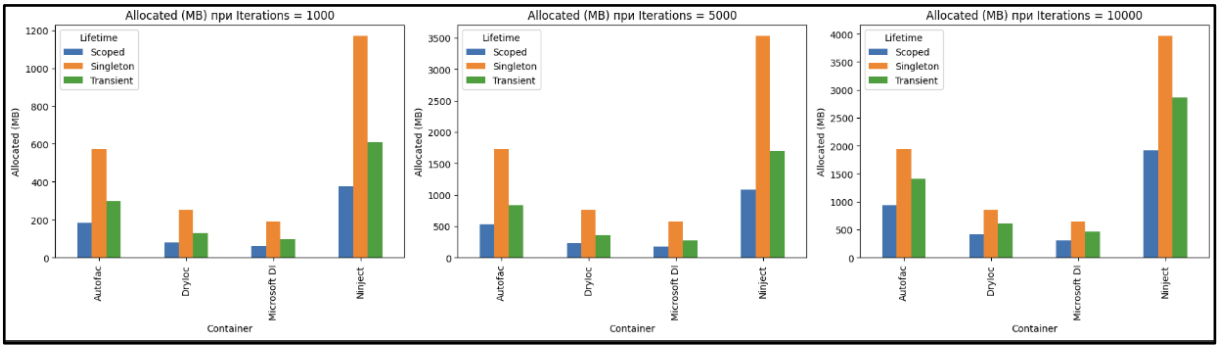

Рисунок 5.12 – Залежність об’єму виділеної пам’яті від кількості ітерацій для сценарію створення (рисунок виконаний самостійно)

Щоб отримати узагальнену картину ефективності, була обчислена спеціальна метрика, яка враховує як час виконання, так і обсяг виділеної пам’яті (див. табл. 5.3).

Таблиця 5.3 – Середня ефективність контейнерів залежно від життєвого циклу у сценарії створення (таблиця виконана самостійно)

Container	Efficiency	Relative (%)
Transient		
DryIoc	11.65	60.5
Microsoft DI	19.26	100
Autofac	0.86	4.49
Ninject	0.21	1.11
Scoped		
DryIoc	35.36	64.31
Microsoft DI	54.99	100
Autofac	2.45	4.47
Ninject	0.68	1.24
Singleton		
DryIoc	1.73	65.19
Microsoft DI	2.66	100
Autofac	0.12	4.87
Ninject	0.03	1.32

Особливий інтерес становлять результати, що поєднують обидва чинники – швидкодію та використання пам’яті. Тут Microsoft DI досягає 100% відносної ефективності у всіх режимах, тоді як DryIoc трохи поступається, але зберігає стабільно високі показники. У Scoped-режимі DryIoc навіть випереджає Microsoft DI в абсолютному значенні, що пояснюється ретельно оптимізованим управлінням об’єктами в межах одного запиту – критично важливою властивістю для веб-застосунків.

Контейнери Autofac і Ninject демонструють суттєво нижчі значення, залишаючись у межах 1–5%, що свідчить про їхню обмежену придатність до інтенсивних сценаріїв створення сутностей, особливо коли важливі швидкість реакції та ефективне використання ресурсів.

Щоб простежити зміну ефективності з урахуванням навантаження, було побудовано графік залежності цієї метрики від кількості ітерацій для кожного з життєвих циклів (див. рис. 5.13). Така візуалізація дозволяє краще оцінити динаміку та стабільність поведінки кожного контейнера в часі.

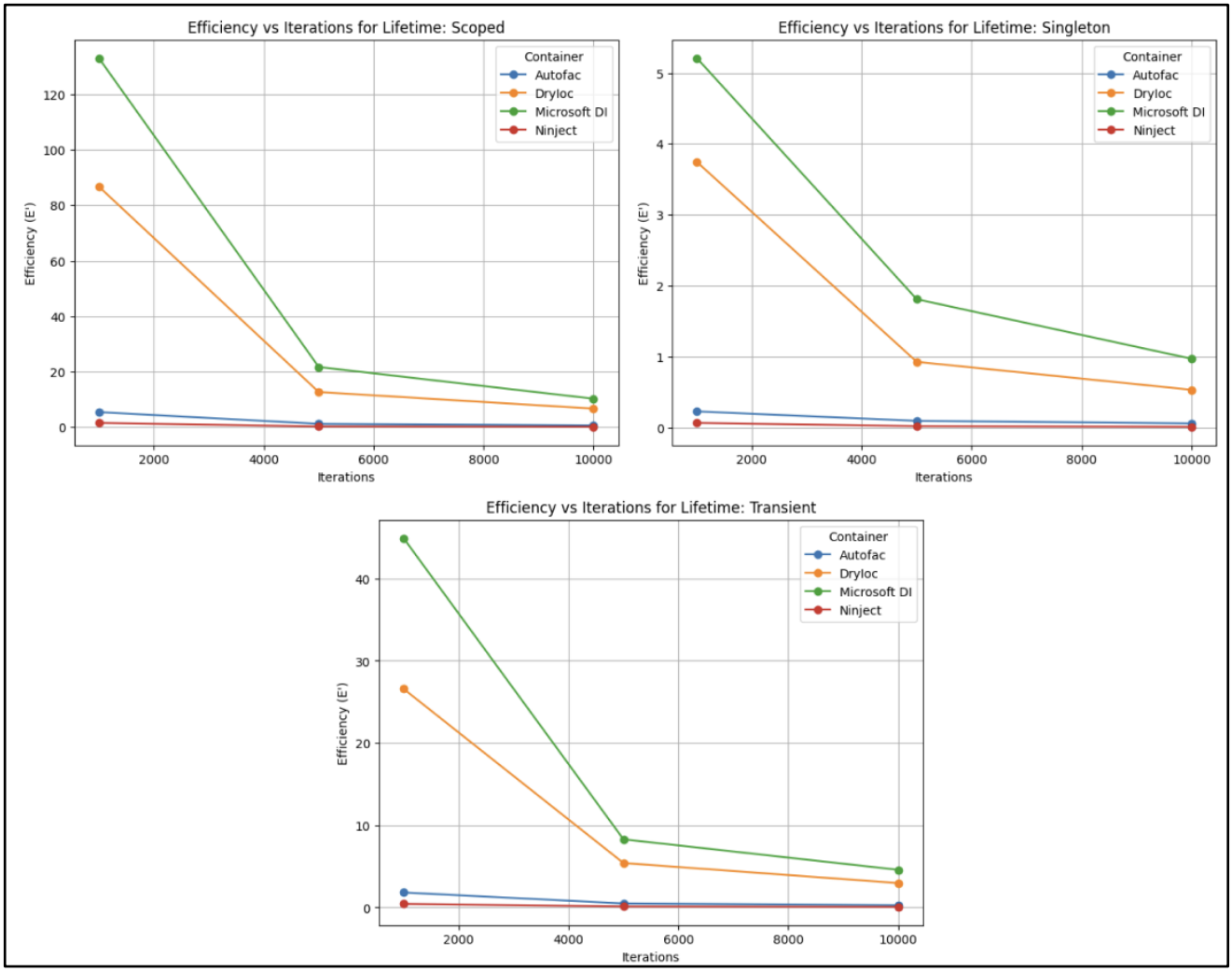

Рисунок 5.13 – Ефективність контейнерів залежно від кількості ітерацій та життєвого циклу для сценарію створення (рисунок виконаний самостійно)

З першого погляду видно, що DryIoc однозначно лідирує при низькому навантаженні – до 1000 ітерацій він демонструє максимальне значення ефективності, що перевищує 45 умовних одиниць. Це підтверджує репутацію

DryIoc як контейнера, оптимізованого для швидкого створення легковагих об'єктів. Тим не менш, із зростанням кількості ітерацій його ефективність стрімко падає – до 8 одиниць при 10000 ітераціях. Це може свідчити про те, що DryIoc використовує внутрішні кеші або структури, які ефективно працюють на коротких дистанціях, але потребують більше ресурсів при довготривалому використанні. Втім, навіть із падінням ефективності він залишається серед лідерів.

Microsoft DI, навпаки, демонструє не стільки максимальні значення, скільки стабільність ефективності. Його крива йде майже горизонтально – починаючи приблизно з 27 і спускаючись до 7.9, тобто зміна становить лише близько 70%, що є відносно невеликим зниженням для такого навантаження. Ця передбачуваність – великий плюс у реальному продакшн-середовищі. Це означає, що поведінка контейнера буде однаково надійною як при невеликому, так і при масштабованому навантаженні. Саме така характеристика і робить Microsoft DI зручним вибором для enterprise-застосунків, де пріоритетом є стабільність та інтеграція зі стеком .NET, а не максимальна продуктивність у синтетичних тестах.

Autofac, попри свою популярність у минулі роки, демонструє вкрай обмежену ефективність – на рівні 2–3 одиниць навіть при мінімальному навантаженні, з поступовим спадом до майже нульових значень при 10 000 ітераціях. Це вказує на наявність суттєвих внутрішніх витрат, зокрема через активне використання рефлексії, побудову графу залежностей у runtime, а також складну конфігурацію. Хоча гнучкість – сильна сторона Autofac, вона, як видно, має свою ціну – втрату продуктивності.

Найбільш вражаючим є провал Ninject. У всьому діапазоні ітерацій контейнер демонструє ефективність, близьку до нуля. Навіть при 1000 ітераціях значення ефективності ледве перевищує 1, а при 10000 – практично нуль. Це свідчить про критично неефективну реалізацію механізмів створення об’єктів у Transient-режимі. Найімовірніше, причина – це надмірна залежність від повільних механізмів рефлексії, повільна побудова графів залежностей та відсутність агресивного кешування. Така поведінка робить Ninject непридатним для будь-яких highload-сценаріїв, де створення об’єктів відбувається масово і часто.

Ще один важливий аспект, який видно з графіка, тенденція до зближення DryIoc та Microsoft DI при зростанні навантаження. Це свідчить про те, що хоча DryIoc демонструє перевагу у сценаріях з помірним навантаженням, Microsoft DI показує кращу масштабованість. У випадках, коли система очікує стабільне навантаження 24/7, краще мати контейнер, який не демонструє різких спадів ефективності.

Загалом цей графік ідеально демонструє вибір, перед яким стоїть архітектор при розробці системи: максимальна продуктивність (DryIoc) чи прогнозована стабільність і простота інтеграції (Microsoft DI). У той же час він чітко вказує на обмежену придатність Ninject і Autofac у системах із високими вимогами до продуктивності.

5.4.4 Аналіз метрик продуктивності для сценарію читання

Одним із найбільш типовий сценаріїв у будь-якому .NET-застосунку є операції читання – тобто отримання даних з бази або іншого джерела. На відміну від створення чи оновлення, читання не змінює стан системи, але часто виконується значно частіше, особливо в інформаційних панелях, звітах, аналітичних системах або веб-інтерфейсах, які регулярно відображають поточні дані для користувача. З цієї причини продуктивність сценаріїв читання є критично важливою для забезпечення швидкої реакції інтерфейсу та загального відчуття "легкості" системи.

Незважаючи на уявну простоту, сценарій читання все ще передбачає залучення DI-контейнера: необхідно створити екземпляри сервісів, які реалізують запити, інтерпретують результати, використовують кеш, трансформують DTO-об’єкти або взаємодіють із репозиторіями. У деяких випадках ці запити можуть виконуватись паралельно або бути частиною більш складних залежностей, тому читання може створювати значне навантаження на контейнер залежностей, особливо в високонавантажених системах.

Більш того, читання часто є першим сценарієм – саме з нього починається перше враження користувача від системи. Якщо інтерфейс гальмує при відкритті

сторінки або завантаженні звіту, це одразу впливає на сприйняття продуктивності. Саме тому важливо не лише оптимізувати логіку запитів, а й забезпечити ефективну роботу контейнера під навантаженням, зокрема під час побудови залежностей, що беруть участь у цих запитах (див. рис. 5.14).

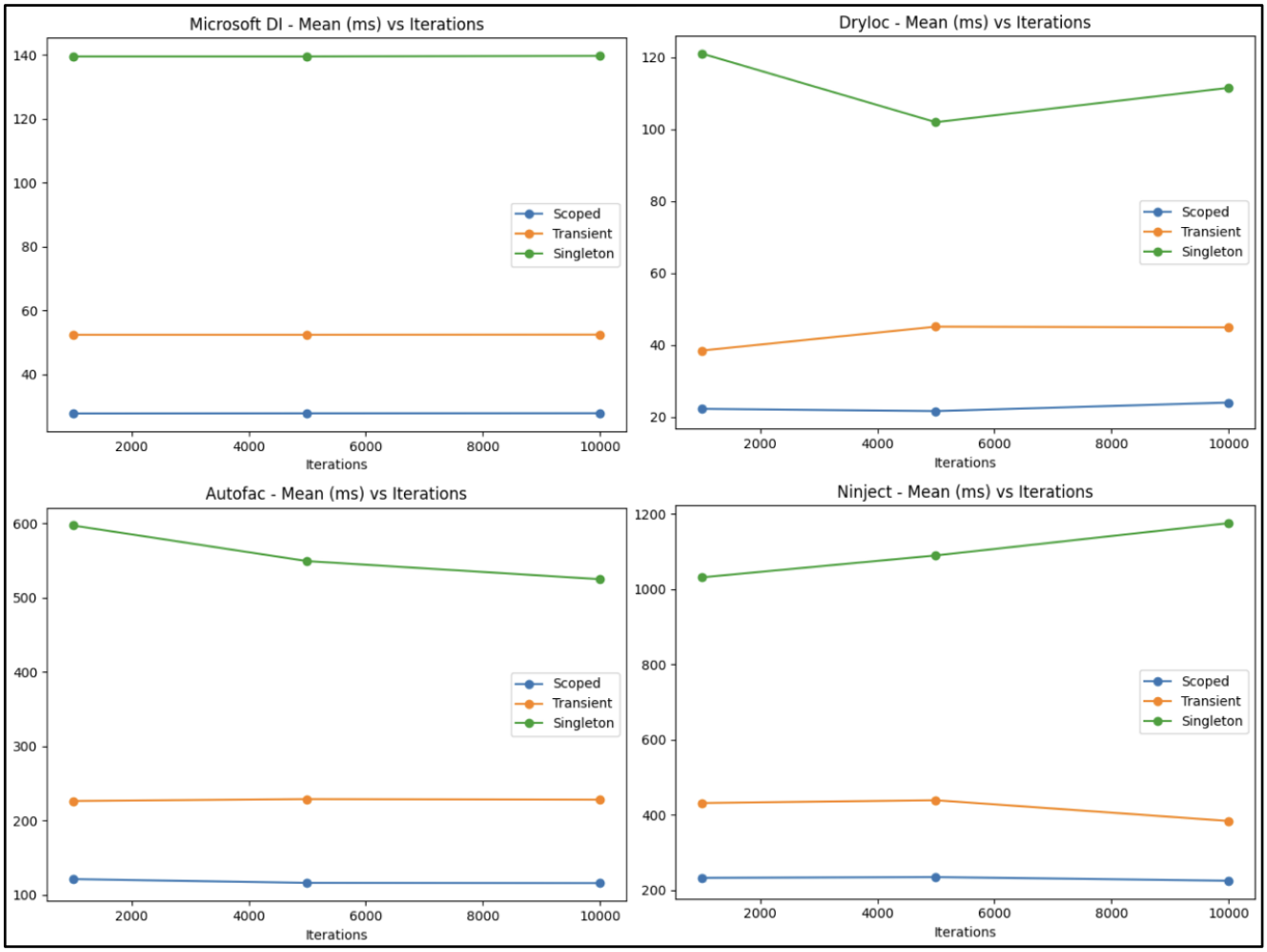

Рисунок 5.14 – Середній час виконаня при різній кількості ітерацій для сценарію створення (рисунок виконаний самостійно)

Аналізуючи представлені графіки, можна помітити, що всі контейнери демонструють відносно стабільну продуктивність у сценарії читання. Це відрізняється від сценарію створення об'єктів, де навантаження на контейнер зазвичай суттєво вище через часті виклики конструкторів та залежностей. У випадку читання, контейнери здебільшого працюють з уже створеними об’єктами або виконують виклики, які не потребують складної ініціалізації. Втім, навіть у цьому сценарії спостерігається помітна різниця в часі реакції між контейнерами –

а це має прямий вплив на сприйняття швидкодії системи користувачем. Варто також зауважити, що сценарії читання можуть включати складну бізнес-логіку, наприклад, фільтрацію або обробку вкладених структур.

Найбільше увагу привертає DryIoc, який у цьому сценарії знову обійшов Microsoft DI, незважаючи на те, що в попередньому сценарії створення об’єктів саме стандартний контейнер .NET показав себе найкраще. Така зміна лідера не випадкова – DryIoc оптимізований для швидкої резолюції залежностей і мінімізації внутрішніх витрат під час кожного звернення, що особливо добре проявляється в читальних операціях. Усі три типи життєвого циклу (Scoped, Transient, Singleton) у DryIoc тримаються на стабільно низькому рівні – значення не перевищують 120 мс навіть при 10000 ітераціях. Особливо цікавим є зниження часу у Singleton режимі при збільшенні ітерацій – це може свідчити про ефективне повторне використання кешованих посилань на об’єкти або агресивну оптимізацію внутрішніх структур.

Microsoft DI, натомість, демонструє повну стабільність, але й суттєво вищі значення часу у Singleton-режимі (понад 140 мс на кожну операцію). Така поведінка може бути пов’язана з менш агресивною оптимізацією резолюції, що дозволяє забезпечити передбачуваність і надійність, але за рахунок меншої швидкодії. З цієї точки зору Microsoft DI залишається дуже привабливим для корпоративних рішень, де важливі стабільність, підтримка та інтеграція з іншим .NET-стеком, навіть якщо ціною є втрата частини продуктивності.

Зовсім іншу ситуацію спостерігаємо з Autofac і Ninject. Їхні результати виглядають значно гірше – середній час виконання операцій коливається у межах 200–600 мс для Autofac та 400–1100 мс для Ninject. Особливо тривожним є зростання часу виконання у Singleton-режимі при збільшенні кількості ітерацій для Ninject, що свідчить про те, що навіть прості запити з часом починають виконуватись повільніше. Це може бути наслідком неефективного кешування, використання рефлексії або небажаного витоку пам’яті. В реальних умовах подібна деградація є критичною, адже сценарій читання виконується постійно й масово.

Ці спостереження дозволяють зробити кілька важливих висновків. По-перше, DryIoc підтвердив свою перевагу у сценаріях, де важлива швидкість реакції, зокрема для читання. Його агресивна оптимізація внутрішніх механізмів резолюції дає помітний ефект, особливо при масштабних навантаженнях. По-друге, Microsoft DI зберігає стабільність, і хоч поступається за абсолютною швидкістю, він дає впевненість у поведінці контейнера незалежно від ітерацій. І нарешті, Autofac та Ninject вкотре демонструють невтішні результати, підтверджуючи, що в умовах сучасних високонавантажених систем їхнє використання повинно бути дуже обґрунтованим і, скоріше, обмеженим спеціалізованими задачами.

Таким чином, DryIoc, попри те що поступився Microsoft DI у сценарії створення, повертає лідерство у читанні – і це добре узгоджується з його архітектурою, орієнтованою на швидке розв'язання легковагих залежностей. Усе це підтверджується й у візуальному порівнянні середнього часу виконання для різних рівнів навантаження та життєвих циклів (див. рис. 5.15), яке дозволяє побачити, як змінюється поведінка контейнерів у типових точках масштабу.

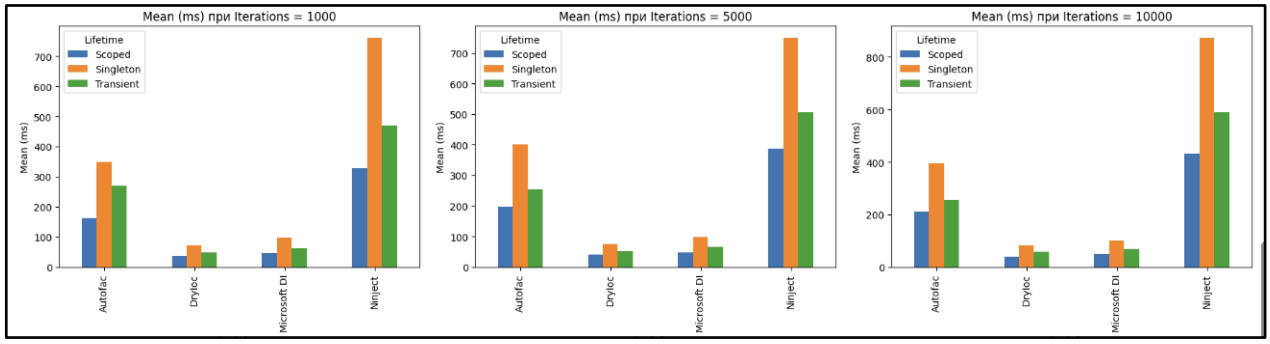

Рисунок 5.15 – Середній час виконаня при різній кількості ітерацій для сценарію створення (рисунок виконаний самостійно)

Однак час виконання – лише один бік продуктивності. У сценаріях читання, які часто запускаються паралельно й у великій кількості, не менш важливим фактором є споживання пам'яті. Навіть незначні витрати на одну ітерацію можуть у підсумку створити суттєве навантаження на систему – і саме це найкраще

проявляється при порівнянні обсягів виділеної пам’яті залежно від кількості ітерацій (див. рис. 5.16).

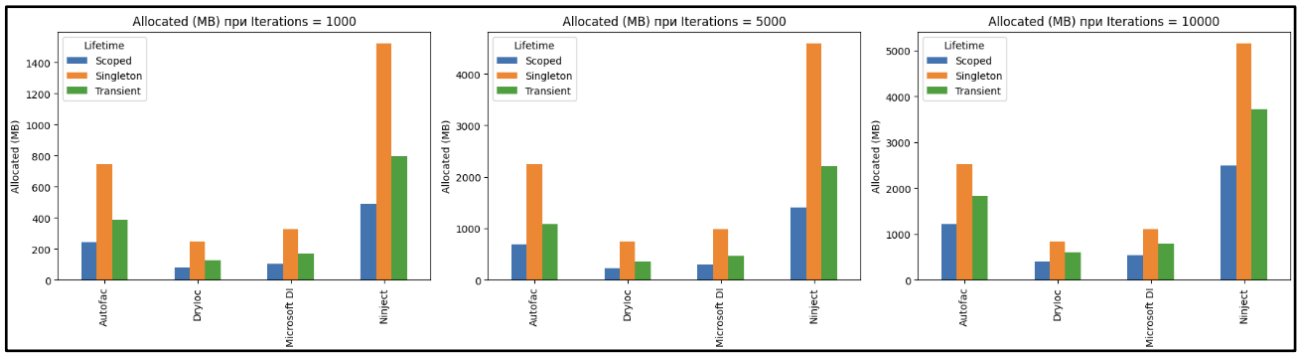

Рисунок 5.16 – Залежність об’єму виділеної пам’яті від кількості ітерацій для сценарію створення (рисунок виконаний самостійно)

Вже при 1000 ітераціях бачимо чітку ієрархію: DryIoc знову демонструє найнижчі значення пам’яті для всіх режимів, ледь перевищуючи 100–300 МБ залежно від циклу. Це підтверджує, що архітектура DryIoc не тільки орієнтована на швидкість, а й має високий рівень контролю над алокацією ресурсів. Його легкість у Scoped і Transient режимах особливо вражає: у цих режимах він показує менше половини споживання пам’яті, ніж Microsoft DI, і на порядок менше ніж у Ninject.

Microsoft DI, своєю чергою, займає середню позицію – він стабільний, не переходить критичні межі навіть при 10 000 ітераціях, але й не пропонує таких мінімальних витрат, як DryIoc. Наприклад, при 5000 ітераціях у Singleton режимі він використовує близько 900 МБ – це суттєво більше за DryIoc, але значно менше ніж у Autofac і тим більше – Ninject.

Найбільш негативно виділяються Autofac і Ninject, які демонструють найвищі показники пам’яті, причому з чіткою тенденцією до зростання. Особливо це стосується Singleton режиму, де, наприклад, у Ninject при 10000 ітераціях обсяг виділеної пам’яті перевищує 5000 МБ. Такий рівень витрат може бути неприпустимим для продакшн-систем, де кожен мегабайт має значення – особливо на сервері з обмеженим обсягом оперативної пам’яті чи при запуску в хмарному середовищі, де ресурси оплачуються погодинно.

Цікаво, що в Autofac спостерігається найбільший дисбаланс між різними типами життєвого циклу – у Singleton пам’ять витрачається у рази більше, ніж у Scoped. Це може свідчити про проблеми в кешуванні або агресивну побудову графу залежностей з подальшим збереженням об’єктів у пам’яті, що створює накопичення.

Цей графік лише підсилює попередні висновки: ефективність DI-контейнера визначається не тільки швидкістю, а й здатністю ощадливо працювати з ресурсами – особливо в умовах високої паралельності або довготривалого використання компонентів. Враховуючи ці аспекти, доцільно звернутись до узагальненої метрики, яка поєднує два ключові фактори: швидкодію та обсяг спожитої пам’яті. На її основі було сформовано порівняльну таблицю ефективності для всіх контейнерів у сценарії читання залежно від типу життєвого циклу (див. табл. 5.4).

Таблиця 5.4 – Середня ефективність контейнерів залежно від життєвого циклу у сценарії читання (таблиця виконана самостійно)

Container	Efficiency	Relative (%)
Transient		
DryIoc	19.73	100
Microsoft DI	12.22	61.93
Autofac	0.94	4.78
Ninject	0.23	1.2
Scoped		
DryIoc	37.4	100
Microsoft DI	24.01	64.21
Autofac	1.92	5.14
Ninject	0.44	1.2
Singleton		
DryIoc	8.09	100

Кінець таблиці 5.4

Container	Efficiency	Relative (%)
Microsoft DI	4.71	58.27
Autofac	0.39	4.83
Ninject	0.09	1.14

З представлених даних чітко видно, що DryIoc утримує впевнене лідерство в усіх режимах життєвого циклу, незалежно від типу навантаження. Його показники ефективності значно перевищують конкурентів – від майже 20 у Transient до 37.4 у Scoped, що свідчить про чудову оптимізацію як у короткоживучих запитах, так і в більш стабільних сесіях.

Другу позицію стабільно займає Microsoft DI, який хоч і поступається DryIoc на 35–40%, однак залишається єдиним із конкурентів, здатним забезпечити хоч якусь помітну ефективність. Це робить його безпечним і надійним вибором для більшості продакшн-сценаріїв, особливо якщо важлива інтеграція з платформою .NET та передбачувана поведінка.

Натомість Autofac і Ninject виглядають критично слабкими – їхні показники не перевищують 5% відносної ефективності, незалежно від життєвого циклу. Навіть у Scoped режимі, де очікується відносна стабільність, Autofac дає лише 1.92, а Ninject – 0.44, що є фактично непридатними значеннями для сучасних застосунків, орієнтованих на високу доступність і швидку реакцію.

Особливо показовим є розрив між DryIoc і рештою контейнерів у Singleton режимі. Якщо DryIoc тримає показник 8.09, то Ninject демонструє лише 0.09, що менше навіть за 1% відносної ефективності. Це ще раз підтверджує, що кешування або повторне використання залежностей у таких контейнерах реалізовано неефективно або навіть шкідливо при високому навантаженні.

Таким чином, інтегральний показник ефективності ще раз закріплює загальні висновки цього розділу: DryIoc – беззаперечний лідер, Microsoft DI – стабільний і практичний варіант, а Autofac та Ninject – варто розглядати лише у разі наявності критичних причин, що виправдовують втрату ефективності. У

контексті читання, де частота викликів висока, а час реакції критичний, ці відмінності мають суттєвий вплив на загальну продуктивність системи.

Щоб доповнити оцінку ефективності ще одним ракурсом, доцільно звернутись до графіків динаміки метрики ефективності в залежності від кількості ітерацій для кожного життєвого циклу окремо. На відміну від попередньої таблиці, яка відображала середні значення, ці графіки дозволяють оцінити, як саме поводяться контейнери при поступовому зростанні навантаження – чи здатні вони зберігати продуктивність, чи демонструють різку деградацію (див. рис. 5.17).

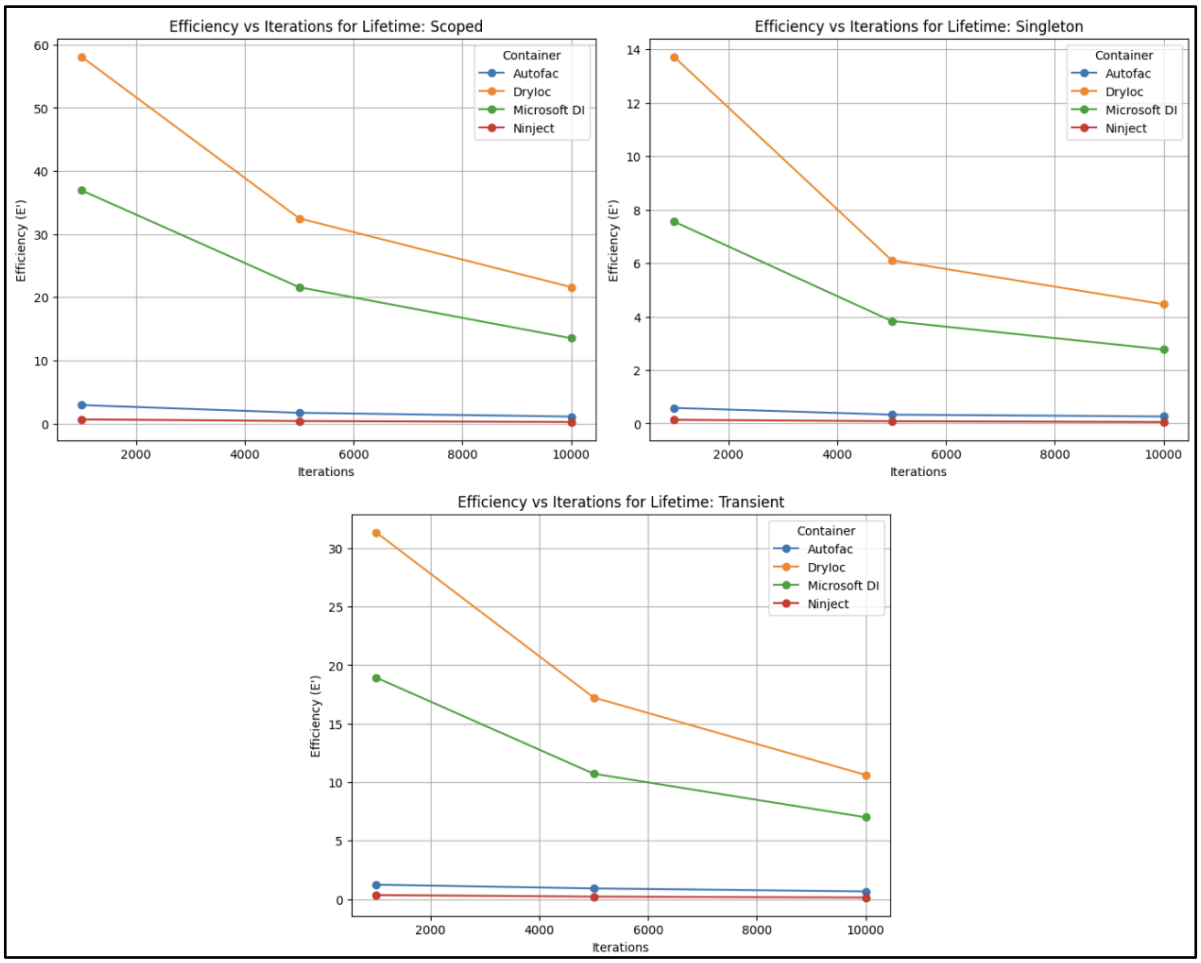

Рисунок 5.17 – Ефективність контейнерів залежно від кількості ітерацій та життєвого циклу для сценарію читання (рисунок виконаний самостійно)

На всіх трьох графіках помітна схожа тенденція: ефективність поступово знижується зі збільшенням кількості ітерацій, однак темп цієї деградації є різним. DryIoc починає з найвищих значень у кожному життєвому циклі і поступово

зменшується, зберігаючи відрив. Microsoft DI стартує нижче, але також демонструє передбачувану і більш плавну криву зниження. Для обох контейнерів характерне те, що попри зростання навантаження, ефективність знижується поступово, а не стрибкоподібно – це важлива властивість для стабільного масштабування у реальних системах.

Особливо цікаво виглядає порівняння Scoped і Transient режимів: DryIoc у Scoped зберігає ефективність навіть при 10000 ітераціях вище, ніж Microsoft DI у початкових точках, що лише підкреслює високий рівень оптимізації. У Singleton обидва контейнери демонструють очікуване зниження, але залишаються у прийнятних межах. Це дозволяє розглядати їх як придатні для довготривалих життєвих циклів.

Сценарій читання підтвердив, що вибір DI-контейнера безпосередньо впливає не лише на продуктивність у межах однієї операції, а й на поведінку системи при масштабуванні. DryIoc виявився найбільш збалансованим за всіма параметрами – час, пам’ять, ефективність та стабільність при навантаженні. Microsoft DI, хоч і менш продуктивний у пікових значеннях, показав себе як стабільне і передбачуване рішення для широкого спектра задач. Таким чином, при проєктуванні системи, орієнтованої на велику кількість читальних запитів, варто враховувати не лише синтетичні показники, а й те, як система поводиться у динаміці, що наочно демонструють саме ці графіки.

5.5 Висновки експериментального дослідження та рекомендації

На основі проведеного експериментального дослідження було сформовано комплексні висновки щодо ефективності різних DI-контейнерів у типових для .NET-застосунків сценаріях. Результати показали, що продуктивність контейнерів залежить від конкретних умов експлуатації, типу життєвого циклу об’єктів, а також інтенсивності навантаження.

У ході експерименту контейнер DryIoc продемонстрував стабільно високі результати практично в усіх сценаріях: логування, кешування та читання. Однак у сценарії створення сутностей він дещо поступився Microsoft DI, який несподівано

проявив вищу швидкодію та стабільність при масштабуванні навантаження. Це свідчить про високу якість стандартного рішення Microsoft, яке повністю інтегроване у .NET-екосистему, забезпечуючи передбачуваність та надійність у роботі.

Особливої уваги заслуговує контейнер DryIoc, який, попри нижчий загальний рівень використання порівняно з більш зрілими рішеннями, такими як Autofac, демонструє стрімке зростання популярності. За останні роки DryIoc активно набирає довіру спільноти, що свідчить про його потенціал як сучасного та ефективного рішення. DryIoc ще перебуває на етапі активного впровадження у проєкти, проте вже зараз може розглядатися як технологічно зрілий інструмент із фокусом на продуктивність, мінімізацію споживання ресурсів та гнучкість у конфігурації. Його подальше визнання значною мірою залежить від розширення спільноти, зростання кількості прикладів використання та розбудови довіри серед розробників.

Водночас важливо враховувати, що DryIoc не є офіційним продуктом Microsoft і поставляється у вигляді зовнішнього NuGet-пакету. Саме це може створювати певні ризики для проєктів, орієнтованих на максимальну стабільність і підтримку з боку платформи. Проте у випадках, коли критично важливими є саме швидкість резолюції залежностей та ефективне використання ресурсів, використання DryIoc може бути компромісним рішенням, яке дозволяє досягти вищих показників продуктивності, особливо у високонавантажених компонентах системи.

Втім, для більшості типових .NET-проєктів, де ключовими є стабільність, прогнозованість та підтримка виробника, стандартний Microsoft DI залишається найбільш раціональним вибором. Його результати стабільно високі у більшості сценаріїв, і він не вимагає додаткових зовнішніх залежностей, що значно спрощує процес розробки та підтримки застосунку.

Щодо контейнерів Autofac та Ninject, то їх використання рекомендується лише з обмеженнями. Контейнер Ninject продемонстрував критично низькі результати за всіма показниками й може вважатися застарілим рішенням, яке не

відповідає сучасним вимогам продуктивності та ресурсоефективності. З огляду на це, доцільно проводити міграцію існуючих проєктів з Ninject на більш сучасні рішення, такі як Microsoft DI або DryIoc.

Autofac має специфічні переваги у складних сценаріях, що потребують глибокої конфігурації або спеціальних функцій резолюції залежностей. Однак його суттєва деградація продуктивності при зростанні навантаження не дозволяє рекомендувати його для повного використання у високонавантажених системах. Найкращим підходом буде використання Autofac лише в окремих специфічних модулях чи компонентах, а не для організації всього додатка.

Таким чином, загальні рекомендації, отримані в результаті дослідження, дозволяють розробникам здійснити обґрунтований вибір DI-контейнера відповідно до конкретних вимог проєкту, з урахуванням як технологічних, так і бізнес-факторів.

Проведене дослідження підтвердило, що вибір DI-контейнера у .NET-застосунках має безпосередній і суттєвий вплив на продуктивність системи. Теоретичний аналіз дав змогу сформулювати базові критерії оцінки ефективності впровадження залежностей, зокрема через вибір життєвого циклу (Transient, Scoped, Singleton) і механізмів управління об’єктами. У той час як теоретичні моделі виявили вплив DI-конфігурацій на масштабованість, стабільність і витрати ресурсів, практичне експериментальне дослідження дало змогу кількісно підтвердити ці гіпотези та виявити реальні переваги та обмеження конкретних рішень.

Вихідний код розробленого застосунку доступний для завантаження з [23]. Основні положення роботи були представлені у тезах доповіді [24].

ВИСНОВКИ

У результаті проведеного дослідження було підтверджено, що вибір Dependency Injection (DI) контейнера та конфігурація життєвих циклів об’єктів мають безпосередній вплив на продуктивність .NET-застосунків. Комплексний підхід до аналізу охоплював не лише теоретичне обґрунтування архітектурних рішень, а й широкомасштабне експериментальне тестування, що дозволило отримати об’єктивні, кількісно підтверджені дані щодо поведінки контейнерів у реальних сценаріях – логування, кешування, створення та читання об'єктів.

DryIoc продемонстрував найвищі показники ефективності у більшості сценаріїв завдяки агресивній оптимізації, ощадливому використанню пам’яті та високій швидкодії. Його переваги особливо помітні в задачах із високою повторюваністю запитів – таких як логування, кешування та читання, де важливе ефективне повторне використання залежностей. Водночас Microsoft DI несподівано випередив DryIoc у сценарії створення сутностей, продемонструвавши кращу швидкодію за умов зростання навантаження. Це свідчить про високу якість інтегрованого рішення, яке, попри свою базову реалізацію, є здатним забезпечити стабільну продуктивність у продакшн-середовищі.

Натомість контейнери Autofac і особливо Ninject показали значно нижчі результати за всіма ключовими метриками. Ninject виявився критично неефективним навіть у найпростіших сценаріях, що унеможливлює його використання в сучасних продуктивних системах. Autofac, незважаючи на гнучкість конфігурації, показав нестабільну поведінку під навантаженням і не зміг забезпечити належну ресурсну ефективність, що знижує його доцільність для широкого використання.

Окремий аналіз конфігурацій життєвих циклів дозволив встановити залежність між типом задачі та оптимальним режимом створення об’єктів. У сценаріях логування та кешування найефективнішим виявився Singleton, що мінімізує створення зайвих екземплярів та знижує навантаження на систему. Для створення об'єктів доцільно використовувати Scoped або Transient, які

забезпечують ізольованість та мінімізують побічні ефекти. У читальних операціях найкраще себе проявив Scoped як оптимальний компроміс між стабільністю, ефективністю та можливістю повторного використання залежностей.

Загальні рекомендації, сформовані на основі дослідження, свідчать про доцільність використання DryIoc у високонавантажених, продуктивно-чутливих компонентах системи, де критичними є швидкодія й ощадливе використання ресурсів. Для більшості типових .NET-застосунків найкращим вибором є Microsoft DI, який забезпечує стабільність, простоту підтримки й повну інтеграцію з екосистемою платформи. Застосування Autofac варто розглядати лише в спеціалізованих випадках, де потрібна складна конфігурація залежностей або модульність. Контейнер Ninject, зважаючи на застарілу архітектуру та вкрай низьку ефективність, не рекомендується до використання.

Отримані результати мають значну практичну цінність для архітекторів і розробників .NET-застосунків, оскільки дозволяють обґрунтовано обирати DI-контейнери та стратегії конфігурації життєвих циклів залежностей відповідно до характеру проєкту та його навантажувального профілю. Вони також підтверджують, що навіть стандартні засоби, за умови правильного налаштування, можуть забезпечити високий рівень продуктивності й масштабованості.

Результати роботи були апробовані на наукових конференціях: «Інформаційні технології та автоматизація – 2024» (див. додаток Д) та «1 Міжнародна науково-практична конференція «СУЧАСНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ МІТ@AIS-2025»» (див. додаток Е).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Dependency Injection Pattern with Examples. Medium. 2022. URL: https://medium.com/@mohamed.hashish42/dependency-injection-pattern-with-examples-659cc57da03d (дата звернення: 20.04.2025).

2. Metrics Applicable for Evaluating Software at The Design Stage, Gruzdo I., Kyrychenko I., Tereshchenko G., Shanidze N. Computational Linguistics and Intelligent Systems (COLINS 2021), Kharkiv, Ukraine, 22-23 April, 2021. pp. 916-936. URL: http://ceur-ws.org/Vol-2870 (дата звернення: 20.04.2025).

3. Investigation of Architecture and Technology Stack for e-Archive System, Falatiuk H., Shirokopetleva M., Dudar Z. IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), Kyiv, Ukraine, 8-11 October, 2019. doi: 10.1109/picst47496.2019.9061407 (дата звернення: 20.04.2025).

4. Holistic Adaptive Optimization Techniques for Distributed Data Streaming Systems, Vysotska V., Kyrychenko I., Demchuk V., Gruzdo I. CEUR Workshop Proceedings, Vol. 3668, 2024, pp. 120–132. doi: 10.31110/COLINS/2024-2/009 (дата звернення: 20.04.2025).

5. Optimized Indexing Method in a Hybrid Image Storage Model for Efficient Storage and Access in Big Data Environments, Kyrychenko I., Tereshchenko G., Smelyakov K. IEEE International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET 2024), Lviv-Slavske, Ukraine, 20–24 February, 2024, pp. 1–4. doi: 10.1109/TCSET64720.2024.10755763 (дата звернення: 02.06.2025).

6. Inversion of Control Containers and the Dependency Injection pattern. martinfowler.com. 2004. URL: https://martinfowler.com/articles/injection.html (дата звернення: 20.04.2025).

7. Examining the Influence of Dependency Injection on Software Maintainability, Seemann M., Deursen S. Manning. 2019. 552 p.

8. Dependency Injection in .NET Core 2.0: Make use of constructors, parameters, setters, and interface injection to write reusable and loosely coupled code, Posados M. Packt Publishing, 2017. 436 p.

9. Examining the Influence of Dependency Injection on Software Maintainability, Sun C. -W., Liao C. -F. IEEE/ACIS 8th International Conference on Big Data, Cloud Computing, and Data Science (BCD), Hochimin City, Vietnam, 14-16 December, 2023. doi: 10.1109/BCD57833.2023.10466290 (дата звернення: 21.04.2025).

10. Measuring Impact of Dependency Injection on Software Maintainability, Sun P., Kim D. -K., Ming H., Lu, L. Department of Computer Science and Engineering, Oakland University, Rochester, MI 48309, USA, 14-18 September, 2022. doi: 10.3390/computers11090141 (дата звернення: 21.04.2025).

11. .NET dependency injection. Microsoft. 2024. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection (дата звернення: 23.04.2025).

12. BenchmarkDotNet NuGet. GitHub. 2024. URL: https://github.com/dotnet/BenchmarkDotNet (Дата звернення: 26.04.2025).

13. Service lifetimes. Microsoft. 2024. URL: https://learn.microsoft.com/uk-ua/dotnet/core/extensions/dependency-injection#service-lifetimes (дата звернення: 23.04.2025).

14. Вплив життєвих циклів об'єктів у Dependency Injection на продуктивність .NET-застосунків, Коробов І., Інформаційні технології та автоматизація: матеріали XVII Міжнародної науково-практичної конференції. Одеса, Україна, 31 жовтня – 1 листопада 2024 року. ОНТУ, ВНТУ, 2024. – с. 463-465.

15. NuGet Trends. 2025. URL: https://nugettrends.com/packages?ids=DryIoc&ids=Autofac&ids=Ninject&months=72 (дата звернення 30.04.2025).

16. MS DI Namespace. Microsoft. 2024. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection (дата звернення: 26.04.2025).

17. Autofac Documentation. 2024. URL: https://autofac.org (дата звернення: 26.04.2025).

18. Ninject Documentation. 2024. URL: http://www.ninject.org (дата звернення: 26.04.2025).

19. A step-by-step guide to using Ninject for dependency injection in C#. Medium. 2012. URL: https://medium.com/the-liberators/a-step-by-step-guide-to-using-ninject-for-dependency-injection-in-c-68a125bd7fa4 (дата звернення: 26.04.2025).

20. DryIoc Documentation. GitHub. 2024. URL: https://github.com/dadhi/DryIoc/blob/master/docs/DryIoc.Docs/Home.md (дата звернення: 26.04.2025).

21. Vertical Slice Architecture. Milan Jovanovic Tech. 2023. URL: https://www.milanjovanovic.tech/blog/vertical-slice-architecture (дата звернення: 27.04.2025).

22. CQRS Pattern With MediatR. Milan Jovanovic Tech. 2023. URL: https://www.milanjovanovic.tech/blog/cqrs-pattern-with-mediatr (дата звернення: 27.04.2025).

23. Github – Архівація. URL: https://github.com/ivan-kkk/master-archive (дата звернення 09.06.2025).

24. Methodology for Selecting the Most Efficient DI Container to Maximize .NET Application Performance, Korobov I., Nazarov O., Modern Information Technologies and Artificial Intelligence Systems: Proceedings of the 1st International Scientific and Practical Conference (MIT@AIS-2025). Part 1. Kharkiv – Yaremche, Ukraine, May 19-22, 2025. NURE, 2025. – pp. 142-145.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

2. Metrics Applicable for Evaluating Software at The Design Stage, Gruzdo I., Kyrychenko I., Tereshchenko G., Shanidze N. Computational Linguistics and Intelligent Systems (COLINS 2021), Kharkiv, Ukraine, 22-23 April, 2021. pp. 916-936. URL: http://ceur-ws.org/Vol-2870 (дата звернення: 20.04.2025).

3. Investigation of Architecture and Technology Stack for e-Archive System, Falatiuk H., Shirokopetleva M., Dudar Z. IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), Kyiv, Ukraine, 8-11 October, 2019. doi: 10.1109/picst47496.2019.9061407 (дата звернення: 20.04.2025).

4. Holistic Adaptive Optimization Techniques for Distributed Data Streaming Systems, Vysotska V., Kyrychenko I., Demchuk V., Gruzdo I. CEUR Workshop Proceedings, Vol. 3668, 2024, pp. 120–132. doi: 10.31110/COLINS/2024-2/009 (дата звернення: 20.04.2025).

5. Optimized Indexing Method in a Hybrid Image Storage Model for Efficient Storage and Access in Big Data Environments, Kyrychenko I., Tereshchenko G., Smelyakov K. IEEE International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET 2024), Lviv-Slavske, Ukraine, 20–24 February, 2024, pp. 1–4. doi: 10.1109/TCSET64720.2024.10755763 (дата звернення: 02.06.2025).