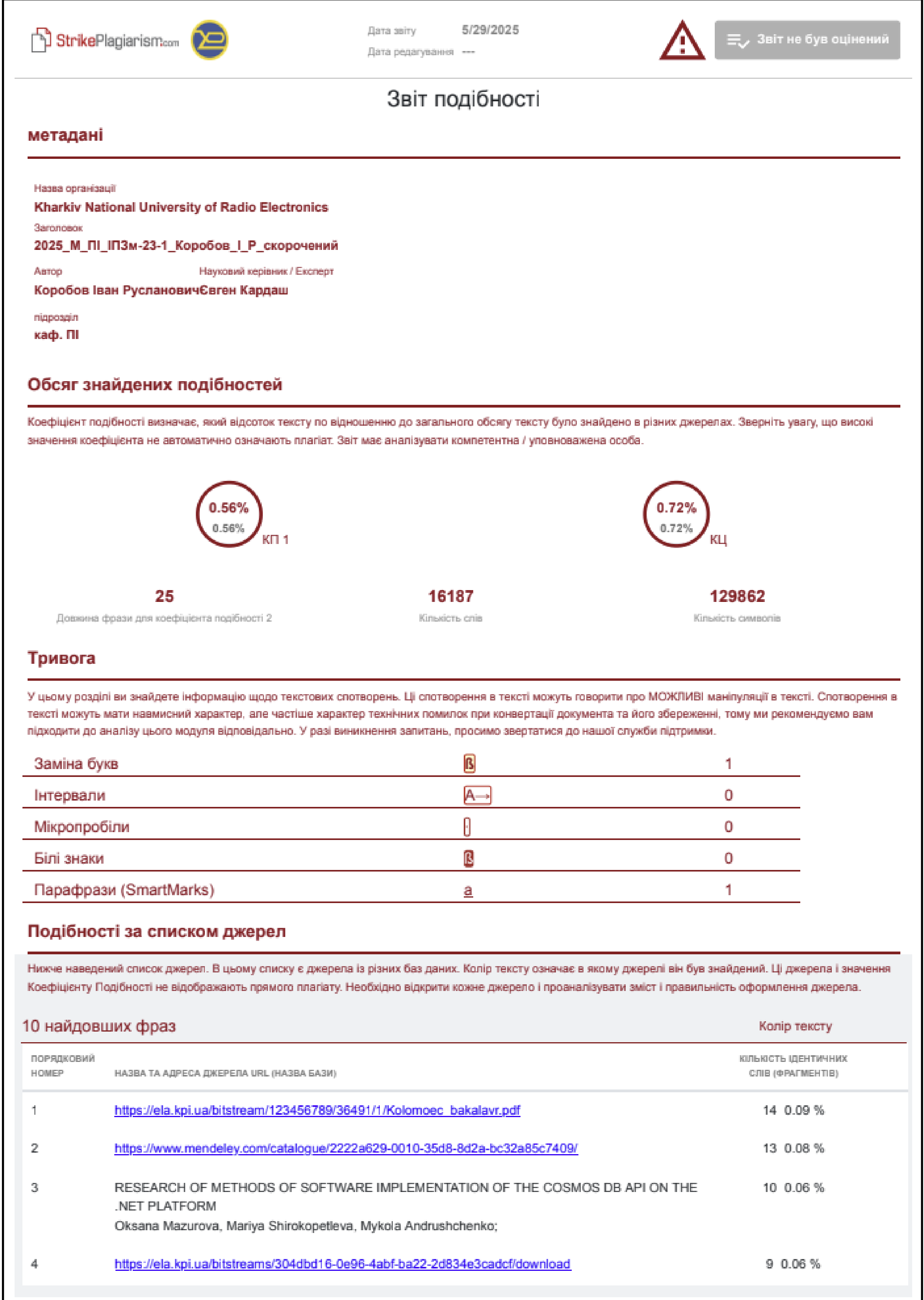
ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ



StrikePlagiarism.com

Дата звіту 5/29/2025
Дата редагування ---

Звіт не був оцінений

Звіт подібності

метадані

Назва організації
Kharkiv National University of Radio Electronics
Заголовок
2025_М_ПІ_ІПЗм-23-1_Коробов_І_Р_скорочений
Автор | Науковий керівник / Експерт
Коробов Іван РуслановичЄвген Кардаш
підрозділ
каф. ПІ

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.56% 0.56% КП 1

0.72% 0.72% КЦ

25 — Довжина фрази для коефіцієнта подібності 2

16187 — Кількість слів

129862 — Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	1
Інтервали	A→	0
Мікропробіли		0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	1

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://ela.kpi.ua/bitstream/123456789/36491/1/Kolomoec_bakalavr.pdf	14 0.09 %
2	https://www.mendeley.com/catalogue/2222a629-0010-35d8-8d2a-bc32a85c7409/	13 0.08 %
3	RESEARCH OF METHODS OF SOFTWARE IMPLEMENTATION OF THE COSMOS DB API ON THE .NET PLATFORM Oksana Mazurova, Mariya Shirokopetleva, Mykola Andrushchenko;	10 0.06 %
4	https://ela.kpi.ua/bitstreams/304dbd16-0e96-4abf-ba22-2d834e3cadcf/download	9 0.06 %

Рисунок А.1 - Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ

ДОДАТОК Б

Код адаптера контейнера для Microsoft DI

```
    public interface IContainerAdapter : IDisposable
    {
        string Name { get; }

        IContainerAdapter Register<TService,
TImplementation>(Microsoft.Extensions.DependencyInjection.ServiceLifetime
lifetime)
            where TService : class
            where TImplementation : class, TService;

        IContainerAdapter RegisterInstance<TService>(TService instance)
            where TService : class;

        IContainerAdapter
RegisterFactory<TService>(Func<IServiceProvider, TService> factory,
Microsoft.Extensions.DependencyInjection.ServiceLifetime lifetime)
            where TService : class;

        IServiceProvider BuildServiceProvider();

        IContainerAdapter
RegisterDbServices(Microsoft.Extensions.DependencyInjection.ServiceLifetime
repositoryLifetime,
Microsoft.Extensions.DependencyInjection.ServiceLifetime
dbContextLifetime);

        IContainerAdapter
RegisterLoggingServices(Microsoft.Extensions.DependencyInjection.ServiceLif
etime loggerLifetime);

        IContainerAdapter
RegisterCachingServices(Microsoft.Extensions.DependencyInjection.ServiceLif
etime cacheLifetime);

        IContainerAdapter
RegisterMediatR(Microsoft.Extensions.DependencyInjection.ServiceLifetime
handlerLifetime);

        ConcurrentDictionary<string, TimeSpan> PerformanceResults { get;
}
    }

public abstract class ContainerAdapterBase : IContainerAdapter
{
    protected readonly ConcurrentDictionary<string, TimeSpan>
_performanceResults = new();

    public abstract string Name { get; }

    public abstract IContainerAdapter Register<TService,
TImplementation>(Microsoft.Extensions.DependencyInjection.ServiceLifetime
lifetime)
        where TService : class
```

```
        where TImplementation : class, TService;

    public abstract IContainerAdapter RegisterInstance<TService>(TService
instance)
        where TService : class;

    public abstract IContainerAdapter
RegisterFactory<TService>(Func<IServiceProvider, TService> factory,
Microsoft.Extensions.DependencyInjection.ServiceLifetime lifetime)
        where TService : class;

    public abstract IServiceProvider BuildServiceProvider();

    protected T MeasureOperation<T>(string operationName, Func<T>
operation)
    {
        var stopwatch = Stopwatch.StartNew();
        var result = operation();
        stopwatch.Stop();

        _performanceResults.AddOrUpdate(
            operationName,
            stopwatch.Elapsed,
            (_, existingValue) => existingValue + stopwatch.Elapsed);

        return result;
    }

    public ConcurrentDictionary<string, TimeSpan> PerformanceResults =>
_performanceResults;

    public abstract IContainerAdapter
RegisterDbServices(Microsoft.Extensions.DependencyInjection.ServiceLifetime
repositoryLifetime,
Microsoft.Extensions.DependencyInjection.ServiceLifetime
dbContextLifetime);

    public abstract IContainerAdapter RegisterLoggingServices(
        Microsoft.Extensions.DependencyInjection.ServiceLifetime
loggerLifetime);

    public abstract IContainerAdapter RegisterCachingServices(
        Microsoft.Extensions.DependencyInjection.ServiceLifetime
cacheLifetime);

    public abstract IContainerAdapter
RegisterMediatR(Microsoft.Extensions.DependencyInjection.ServiceLifetime
handlerLifetime);

    public virtual void Dispose()
    {
        GC.SuppressFinalize(this);
    }
}
using ECommerceIoCAnalyzer.Infrastructure.Abstractions;
using ECommerceIoCAnalyzer.Infrastructure.EF;
using ECommerceIoCAnalyzer.PerfomanceTests.Infrastructure.Caching;
using Microsoft.EntityFrameworkCore;
```

```
using Microsoft.Extensions.DependencyInjection.Extensions;
using Serilog.Extensions.Logging;
using ILogger = Serilog.ILogger;

namespace ECommerceIoCAnalyzer.PerfomanceTests.Infrastructure.DIContainers;

public class MicrosoftDIAdapter : ContainerAdapterBase
{
    private readonly IServiceCollection _services;
    private IServiceProvider? _serviceProvider;

    public MicrosoftDIAdapter()
    {
        _services = new ServiceCollection();
    }

    public override string Name => "Microsoft DI";

    public override IContainerAdapter Register<TService, TImplementation>(
        Microsoft.Extensions.DependencyInjection.ServiceLifetime lifetime)
    {
        return
MeasureOperation($"Register_{typeof(TService).Name}_{lifetime}", () =>
        {
            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;
        });
    }

    public override IContainerAdapter RegisterInstance<TService>(TService
instance)
    {
        return
MeasureOperation($"RegisterInstance_{typeof(TService).Name}", () =>
        {
            _services.AddSingleton(instance);
            return this;
        });
    }
```

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

            return this;
        });
    }

    public override IServiceProvider BuildServiceProvider()
    {
        return MeasureOperation("BuildServiceProvider", () =>
        {
            _serviceProvider = _services.BuildServiceProvider();
            return _serviceProvider;
        });
    }

    public override IContainerAdapter RegisterDbServices(
        Microsoft.Extensions.DependencyInjection.ServiceLifetime
repositoryLifetime,
        Microsoft.Extensions.DependencyInjection.ServiceLifetime
dbContextLifetime)
    {
        return
MeasureOperation($"RegisterDbServices_{repositoryLifetime}_{dbContextLifeti
me}", () =>
        {
            _services.RemoveAll<DbContext>();

            _services.AddDbContext<AppDbContext>(options =>
                options.UseSqlServer(connectionString), dbContextLifetime);

            _services.AddScoped<DbContext>(sp =>
sp.GetRequiredService<AppDbContext>());
```

```
            _services.TryAdd(
                new ServiceDescriptor(typeof(IRepository<>),
typeof(GenericRepository<>), repositoryLifetime));

            return this;
        });
    }

    public override IContainerAdapter RegisterLoggingServices(
        Microsoft.Extensions.DependencyInjection.ServiceLifetime
loggerLifetime)
    {
        return
MeasureOperation($"RegisterLoggingServices_{loggerLifetime}", () =>
        {
            var serilogLogger = new LoggerConfiguration()
                .MinimumLevel.Information()
                .WriteTo.Console()
                .WriteTo.File("logs/log-.txt", rollingInterval:
RollingInterval.Day)
                .CreateLogger();

            _services.AddSingleton<ILogger>(serilogLogger);

            _services.AddSingleton<ILoggerFactory>(services =>
                new
SerilogLoggerFactory(services.GetRequiredService<ILogger>()));

            _services.Add(new ServiceDescriptor(typeof(ILogger<>),
typeof(Logger<>), loggerLifetime));

            return this;
        });
    }

    public override IContainerAdapter RegisterCachingServices(
        Microsoft.Extensions.DependencyInjection.ServiceLifetime
cacheLifetime)
    {
        return MeasureOperation($"RegisterCachingServices_{cacheLifetime}",
() =>
        {
            _services.AddMemoryCache();

            _services.Add(new ServiceDescriptor(typeof(ICacheService),
typeof(MemoryCacheService), cacheLifetime));

            return this;
        });
    }

    public override IContainerAdapter RegisterMediatR(
        Microsoft.Extensions.DependencyInjection.ServiceLifetime
handlerLifetime)
    {
        return MeasureOperation($"RegisterMediatR_{handlerLifetime}", () =>
        {
```

```
            var applicationAssembly =
typeof(ECommerceIoCAnalyzer.Application.Features.Orders.Commands.CreateOrde
r.CreateOrderCommandHandler).Assembly;

            _services.AddMediatR(cfg =>
            {
                cfg.RegisterServicesFromAssembly(applicationAssembly);

                switch (handlerLifetime)
                {
                    case
Microsoft.Extensions.DependencyInjection.ServiceLifetime.Singleton:
                        cfg.Lifetime =
Microsoft.Extensions.DependencyInjection.ServiceLifetime.Singleton;
                        break;
                    case
Microsoft.Extensions.DependencyInjection.ServiceLifetime.Scoped:
                        cfg.Lifetime =
Microsoft.Extensions.DependencyInjection.ServiceLifetime.Scoped;
                        break;
                    case
Microsoft.Extensions.DependencyInjection.ServiceLifetime.Transient:
                    default:
                        cfg.Lifetime =
Microsoft.Extensions.DependencyInjection.ServiceLifetime.Transient;
                        break;
                }
            });

            return this;
        });
    }

    public override void Dispose()
    {
        if (_serviceProvider is IDisposable disposable)
        {
            disposable.Dispose();
        }

        base.Dispose();
    }
}
```

ДОДАТОК В

Код розрахунку метрики ефективності для контейнерів на основі виміряних значень

```
import pandas as pd
from io import StringIO
import matplotlib.pyplot as plt

# Placeholder for benchmark results.
# Paste full benchmark dataset here as a tab-separated string in the
raw_data variable below.
# Expected columns: Lifetime, Method, Iterations, Mean (ms), Median (ms),
Min (ms), Max (ms),
# P95 (ms), P90 (ms), Gen0, Gen1, Allocated (MB)
raw_data = ""  # <-- Paste the benchmark dataset here

# Example format (optional, for reference only - not used in processing)
example_data = """Lifetime Method     Iterations Mean (ms)  Median (ms)
     Min (ms)   Max (ms)   P95 (ms)   P90 (ms)   Gen0  Gen1  Allocated (MB)
DryIoc
Scoped     GetProducts      1000  44.82 44.7  43.2  46.4  46    45.5  370.47
     85.32 105.08
"""

# Parse raw data and assign container names as a new column
lines = raw_data.strip().splitlines()
header = lines[0].split("\t") + ["Container"]
data_rows = []
current_container = None

for line in lines[1:]:
    line = line.strip()
    if not line:
        continue
    cells = line.split("\t")
    if len(cells) == 1:
        current_container = cells[0].strip()
    elif len(cells) == len(header) - 1:
        cells.append(current_container or "")
        data_rows.append("\t".join(cells))

# Construct clean data and read into DataFrame
processed_data = "\n".join(["\t".join(header)] + data_rows)
df = pd.read_csv(StringIO(processed_data), sep="\t")

# Convert all numeric columns except for categorical ones
categorical_cols = ["Lifetime", "Method", "Container"]
numeric_cols = df.columns.difference(categorical_cols)
df[numeric_cols] = df[numeric_cols].apply(pd.to_numeric, errors="coerce")

# Optional: filter by a specific container
selected_container = None
if selected_container:
    df = df[df["Container"] == selected_container]
```

```
# Group data by container, lifetime, and iteration count
grouped = df.groupby(["Container", "Lifetime", "Iterations"],
as_index=False).mean(numeric_only=True)

# Efficiency formula coefficients
ALPHA = 0.1
BETA = 0.01
GAMMA = 0.5
P_FACTOR = 100  # Stability multiplier

def calculate_efficiency(row):
    """
    Calculates efficiency score E' using the formula:

        E' = (Tr * P) / (Median + α * Allocated + β * GC_total + γ * Delta)

    where:
        - Tr = 1000 / Mean (throughput)
        - GC_total = Gen0 + Gen1 (+ Gen2 if available)
        - Delta = P95 - Median (latency variability)
    """
    mean_time = row["Mean (ms)"]
    if mean_time <= 0:
        return None

    throughput = 1000 / mean_time
    median = row["Median (ms)"]
    allocated = row["Allocated (MB)"]
    gen0 = row["Gen0"]
    gen1 = row["Gen1"]
    gen2 = row.get("Gen2", 0)

    gc_total = gen0 + gen1 + (gen2 if pd.notna(gen2) else 0)
    delta = row["P95 (ms)"] - median if pd.notna(row.get("P95 (ms)")) else
0

    denominator = median + ALPHA * allocated + BETA * gc_total + GAMMA *
delta
    return (throughput * P_FACTOR) / denominator if denominator != 0 else
None

# Apply efficiency calculation
grouped["Efficiency"] = grouped.apply(calculate_efficiency, axis=1)

# Print grouped metrics with efficiency
print("Grouped metrics with calculated efficiency (E'):")
print(grouped)

# Plot efficiency over iterations per container, for each lifetime
for lifetime in grouped["Lifetime"].unique():
    subset = grouped[grouped["Lifetime"] == lifetime]
    plt.figure(figsize=(8, 6))
    for container, container_data in subset.groupby("Container"):
        container_data = container_data.sort_values("Iterations")
        plt.plot(container_data["Iterations"],
container_data["Efficiency"], marker="o", label=container)
```

```
    plt.title(f"Efficiency vs. Iterations – Lifetime: {lifetime}")
    plt.xlabel("Iterations")
    plt.ylabel("Efficiency (E')")
    plt.legend(title="IoC Container")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# === Average efficiency computation per Container and Lifetime ===
avg_efficiency = grouped.groupby(["Container", "Lifetime"],
as_index=False)["Efficiency"].mean()

# Sort by lifetime and descending efficiency
avg_efficiency = avg_efficiency.sort_values(by=["Lifetime", "Efficiency"],
ascending=[True, False])

# Compute relative efficiency (%) within each lifetime group
avg_efficiency["Relative (%)"] =
avg_efficiency.groupby("Lifetime")["Efficiency"] \
    .transform(lambda x: (x / x.max() * 100).round(2))

# Output results
print("\nAverage Efficiency by Container and Lifetime:")
print(avg_efficiency.to_string(index=False))
```

ДОДАТОК Г

Презентаційний матеріал до кваліфікаційної роботи

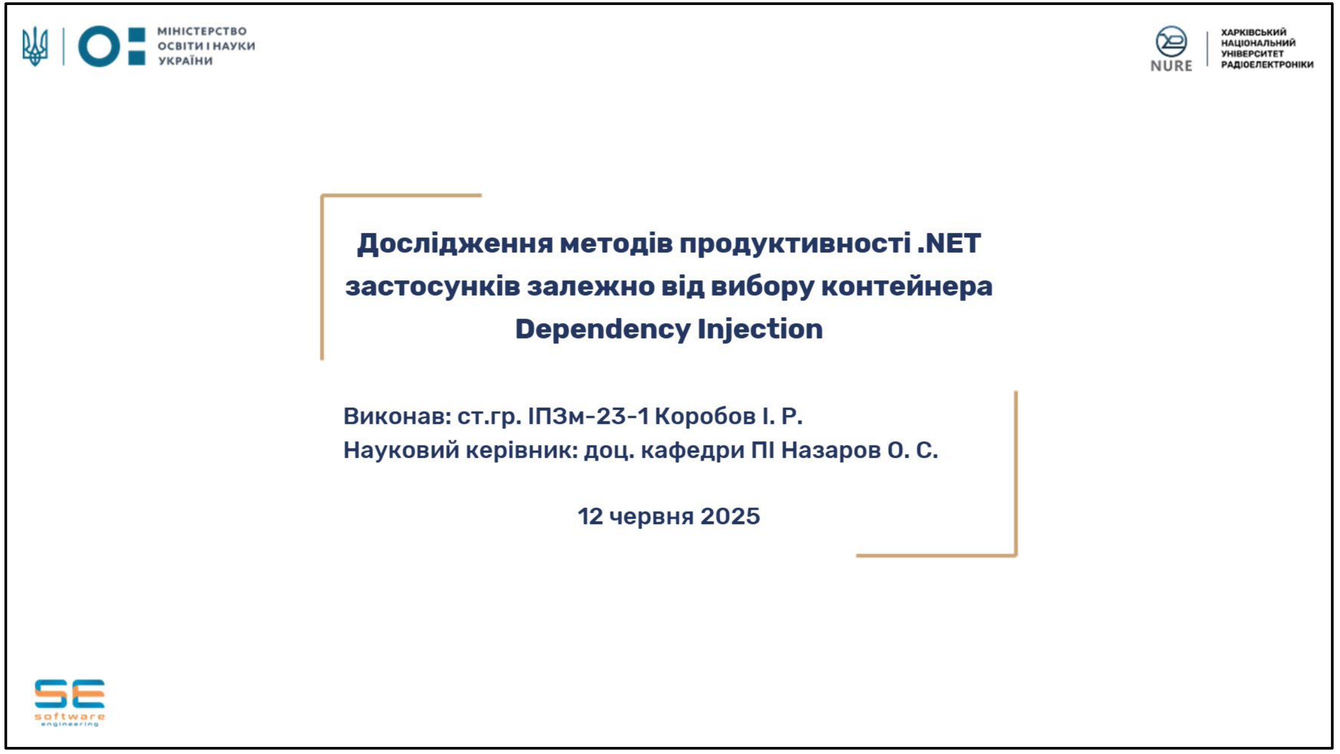


Рисунок Г.1 – Титульний аркуш (рисунок виконаний самостійно)

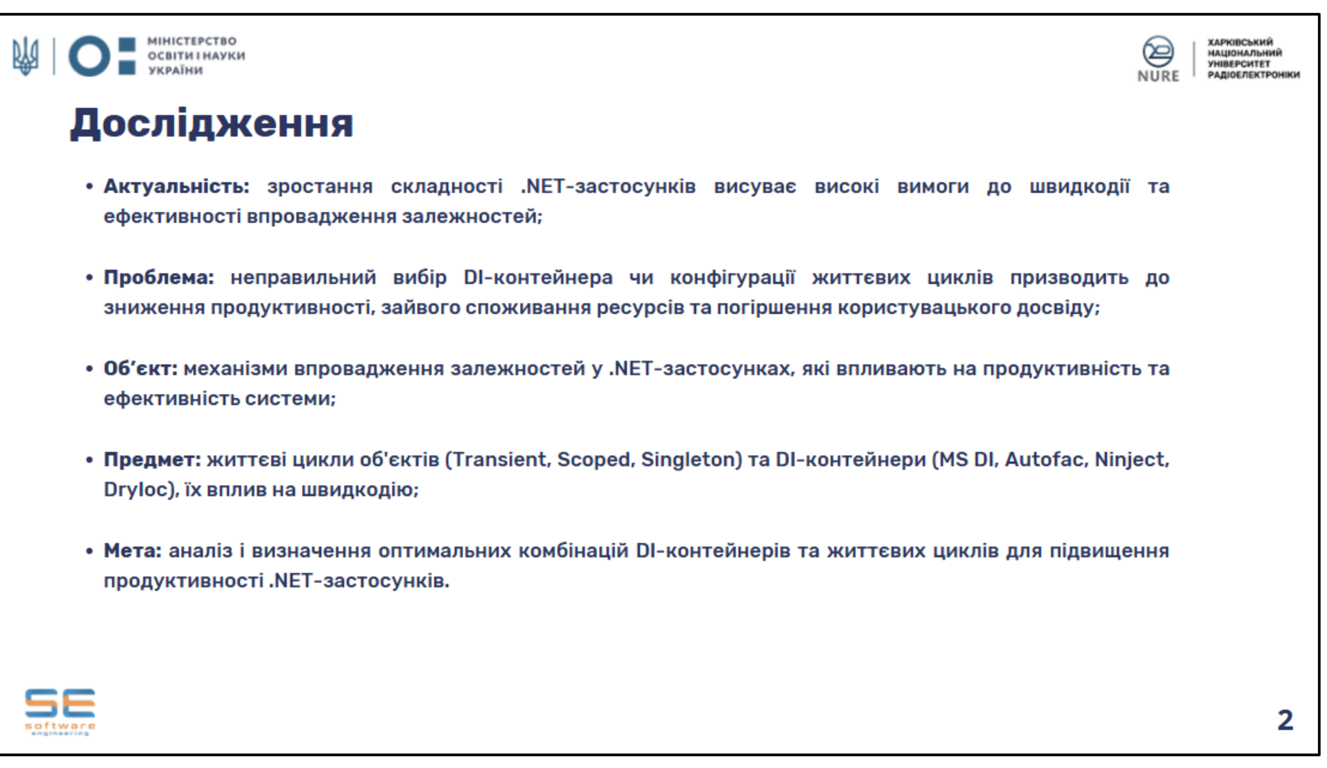


Рисунок Г.2 – Аркуш «Дослідження» (рисунок виконаний самостійно)

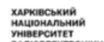


Рисунок Г.3 – Аркуш «Огляд літератури» (рисунок виконаний самостійно)

Рисунок Г.4 – Аркуш «Постановка задачі» (рисунок виконаний самостійно)

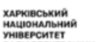


Рисунок Г.5 – Аркуш «Методологія» (рисунок виконаний самостійно)

Рисунок Г.6 – Аркуш «Проведення експерименту» (рисунок виконаний самостійно)

Рисунок Г.7 – Аркуш «Опис програмного забезпечення» (рисунок виконаний самостійно)

Рисунок Г.8 – Аркуш «Високорівневий огляд архітектури тестового проєкту» (рисунок виконаний самостійно)

Рисунок Г.9 – Аркуш «UML-діаграма класів інфраструктури для тестування» (рисунок виконаний самостійно)

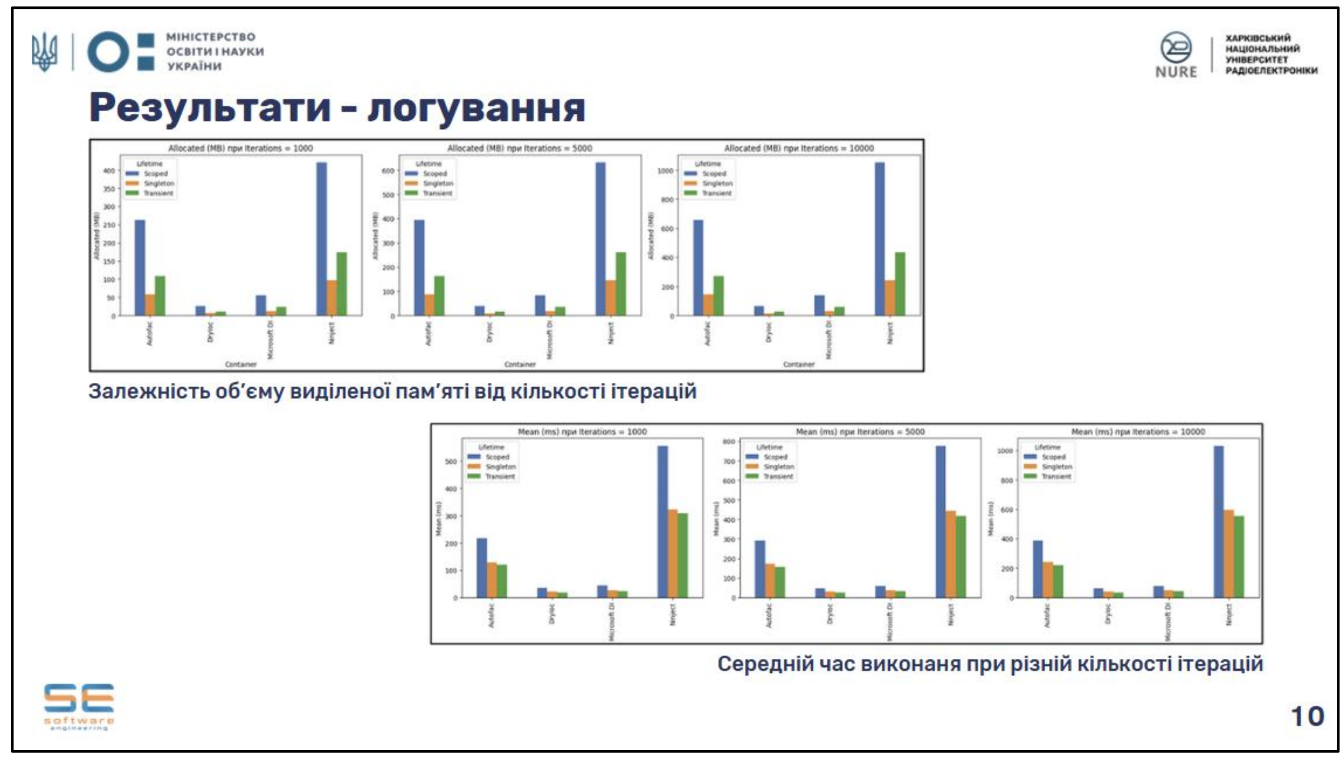


Рисунок Г.10 – Аркуш «Результати – логування» (рисунок виконаний самостійно)

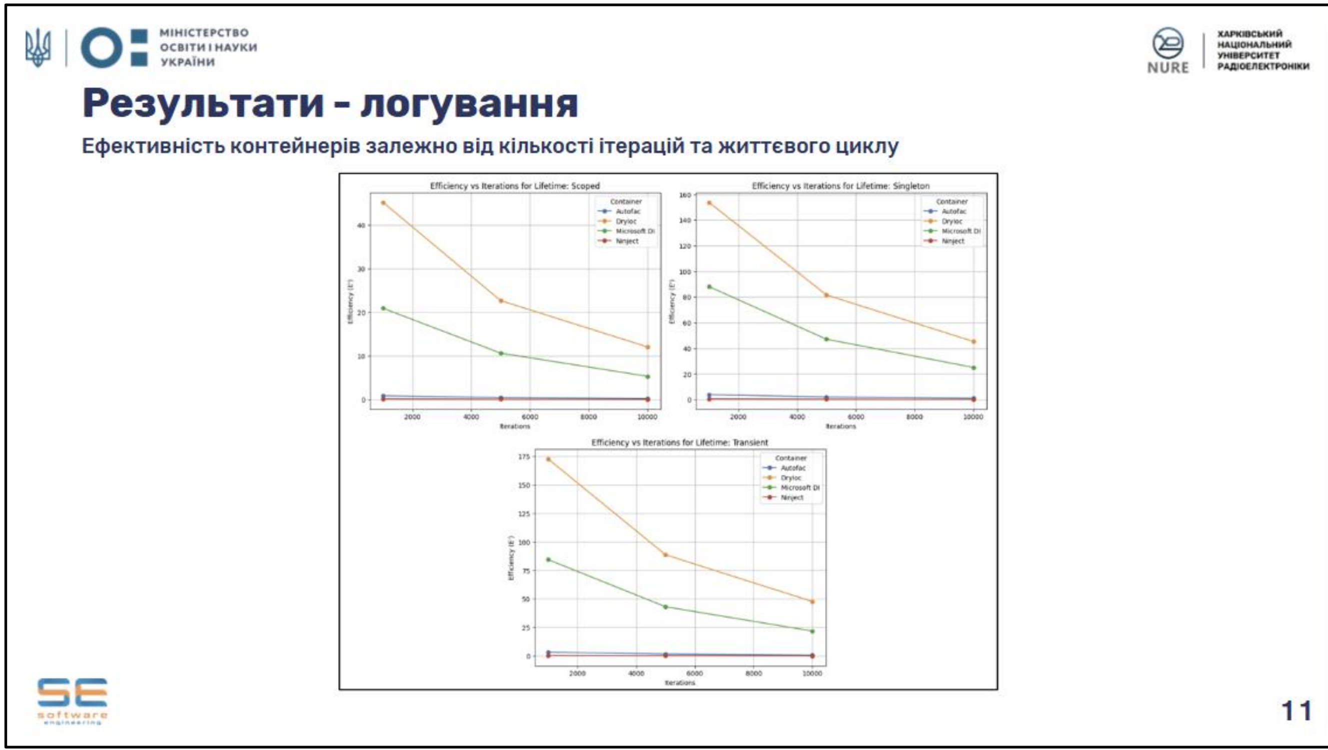


Рисунок Г.11 – Аркуш «Результати – логування» (рисунок виконаний самостійно)

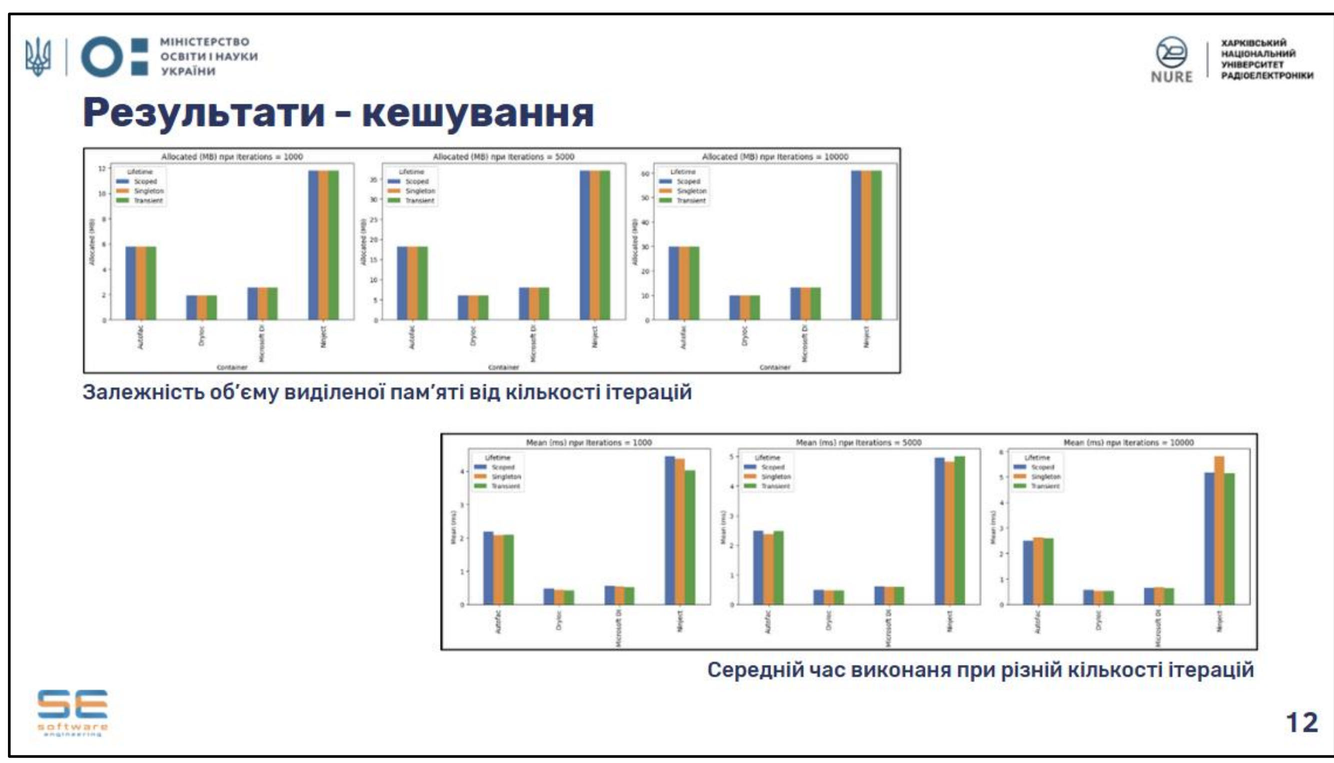


Рисунок Г.12 – Аркуш «Результати – кешування» (рисунок виконаний самостійно)

Рисунок Г.13 – Аркуш «Результати – кешування» (рисунок виконаний самостійно)

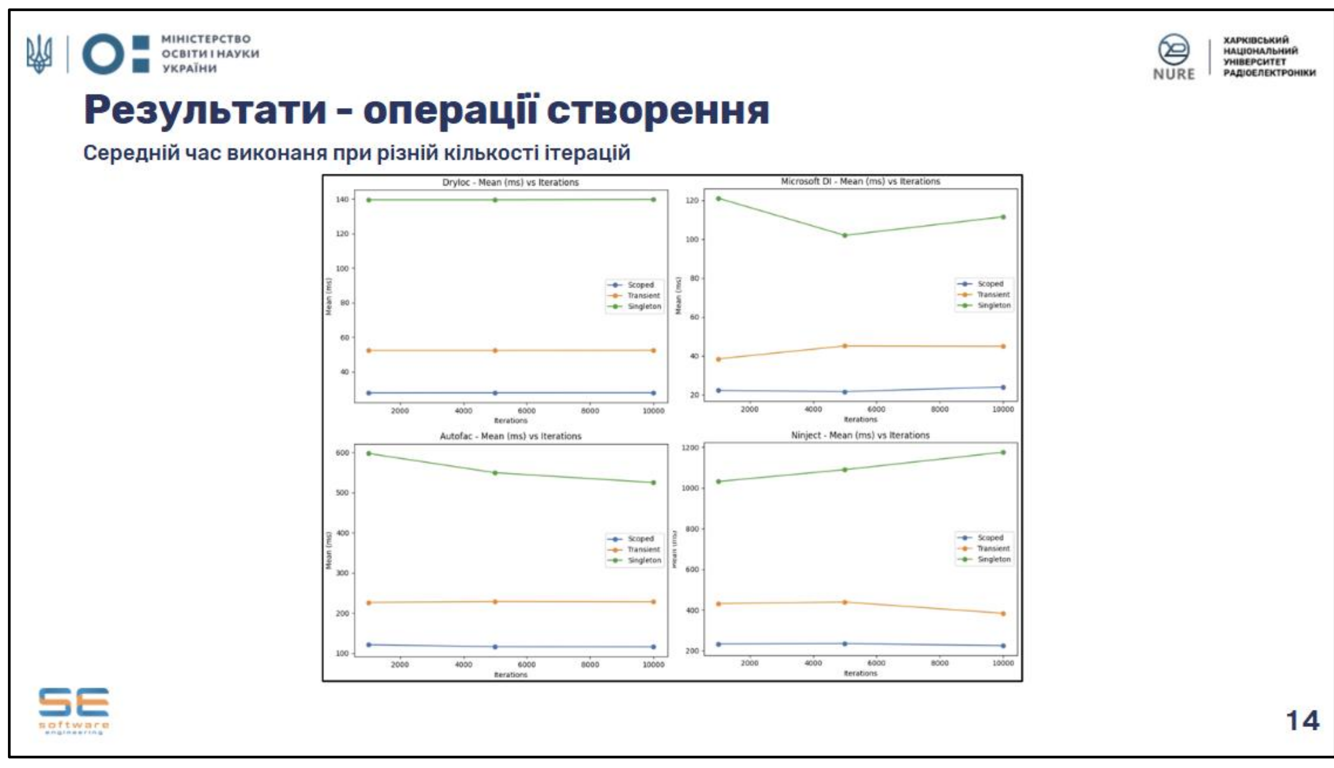


Рисунок Г.14 – Аркуш «Результати – операції створення» (рисунок виконаний самостійно)

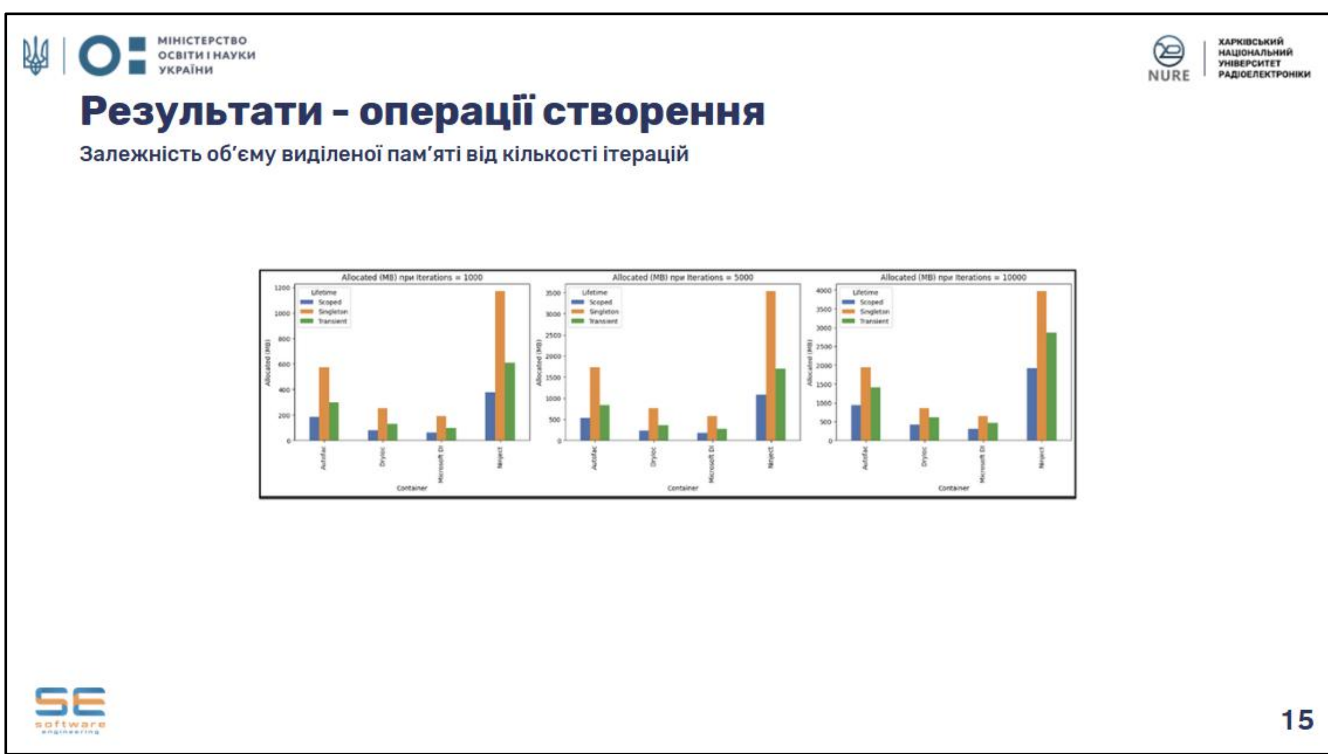


Рисунок Г.15 – Аркуш «Результати – операції створення» (рисунок виконаний самостійно)

Рисунок Г.16 – Аркуш «Результати – операції створення» (рисунок виконаний самостійно)

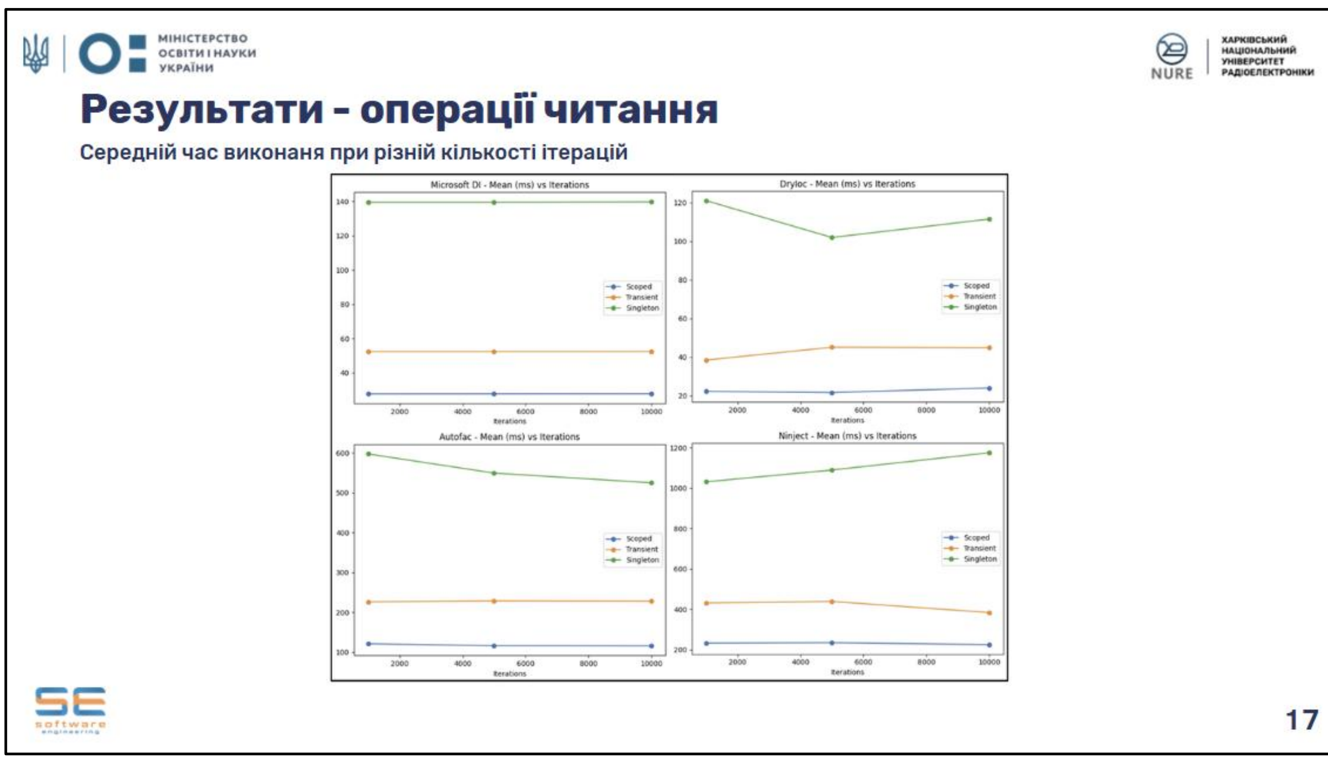


Рисунок Г.17 – Аркуш «Результати – операції читання» (рисунок виконаний самостійно)

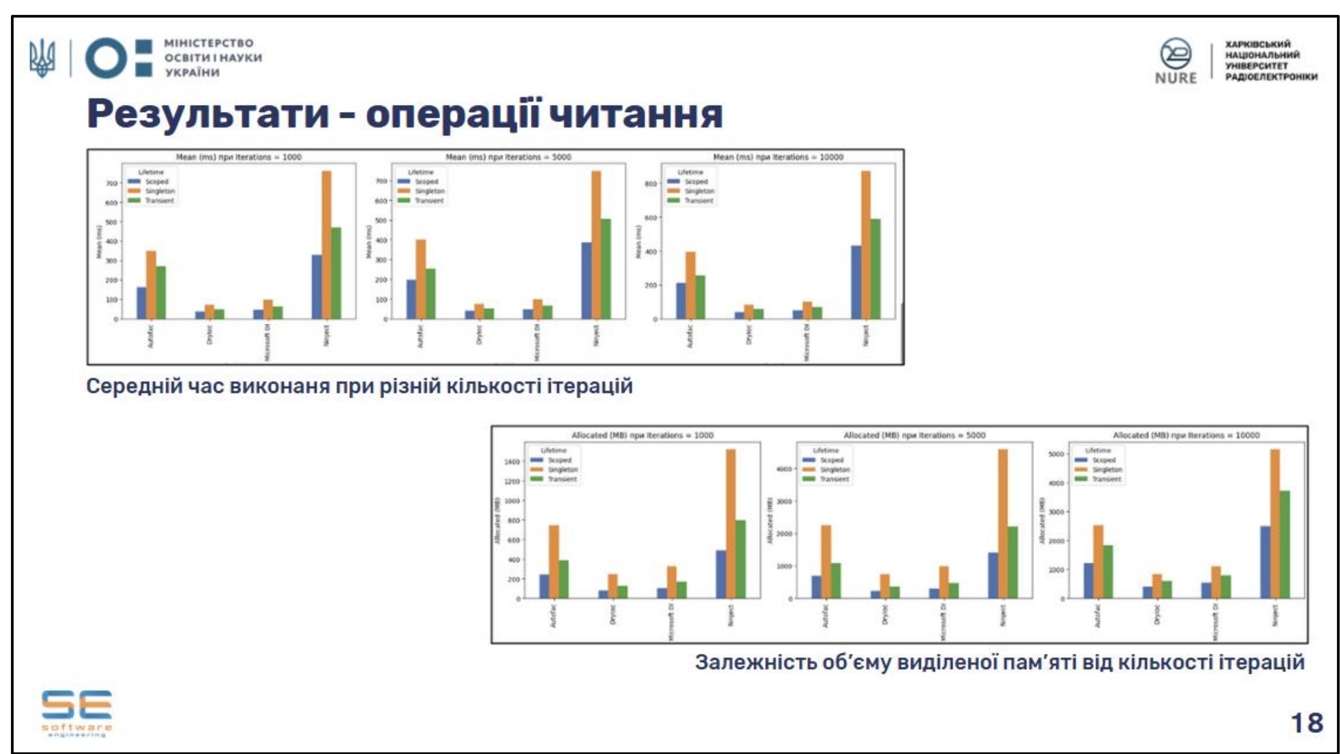


Рисунок Г.18 – Аркуш «Результати – операції читання» (рисунок виконаний самостійно)

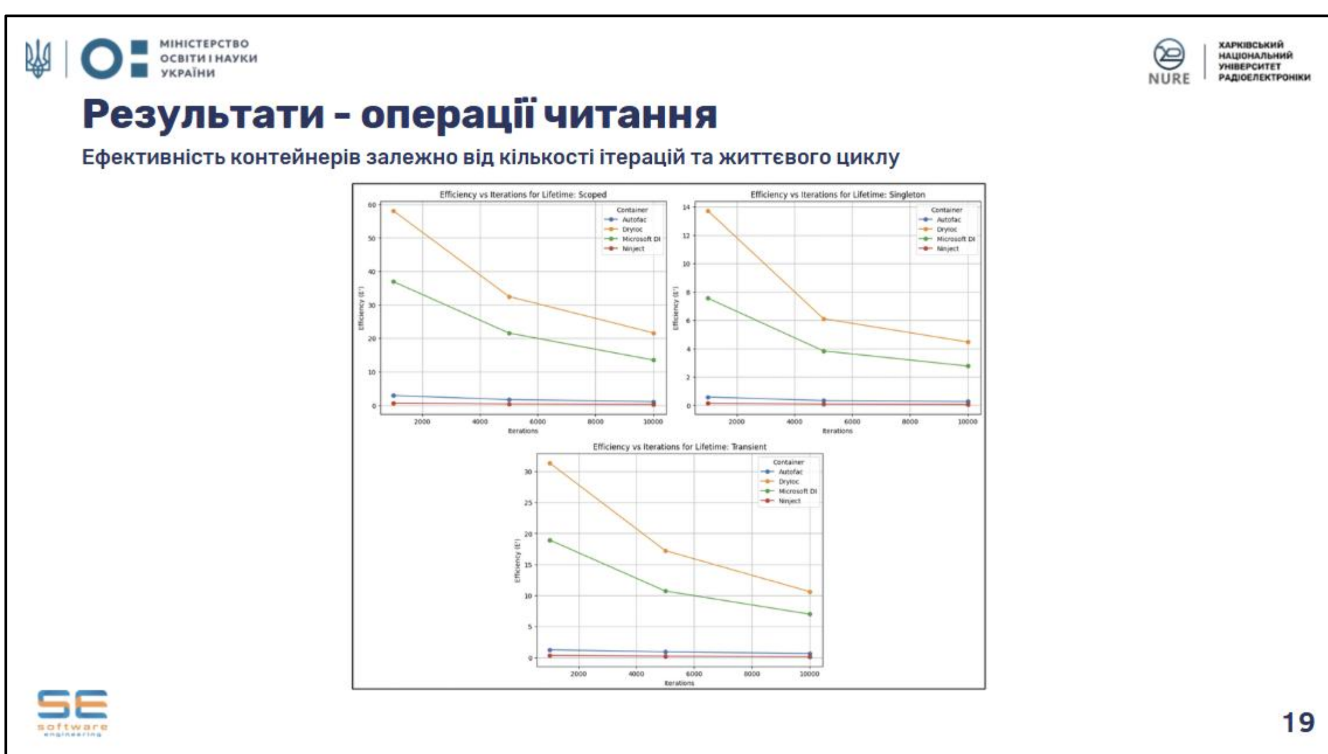


Рисунок Г.19 – Аркуш «Результати – операції читання» (рисунок виконаний самостійно)

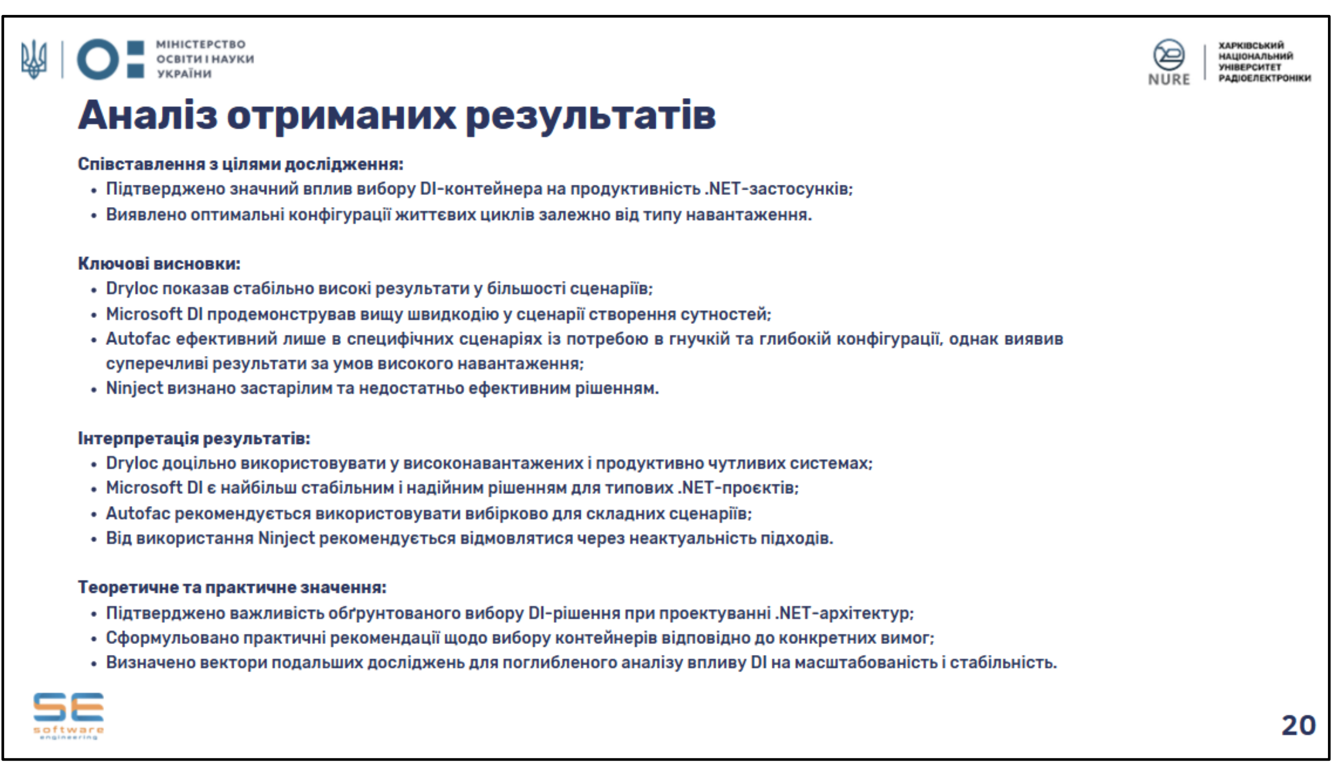


Рисунок Г.20 – Аркуш «Аналіз отриманих результатів» (рисунок виконаний самостійно)

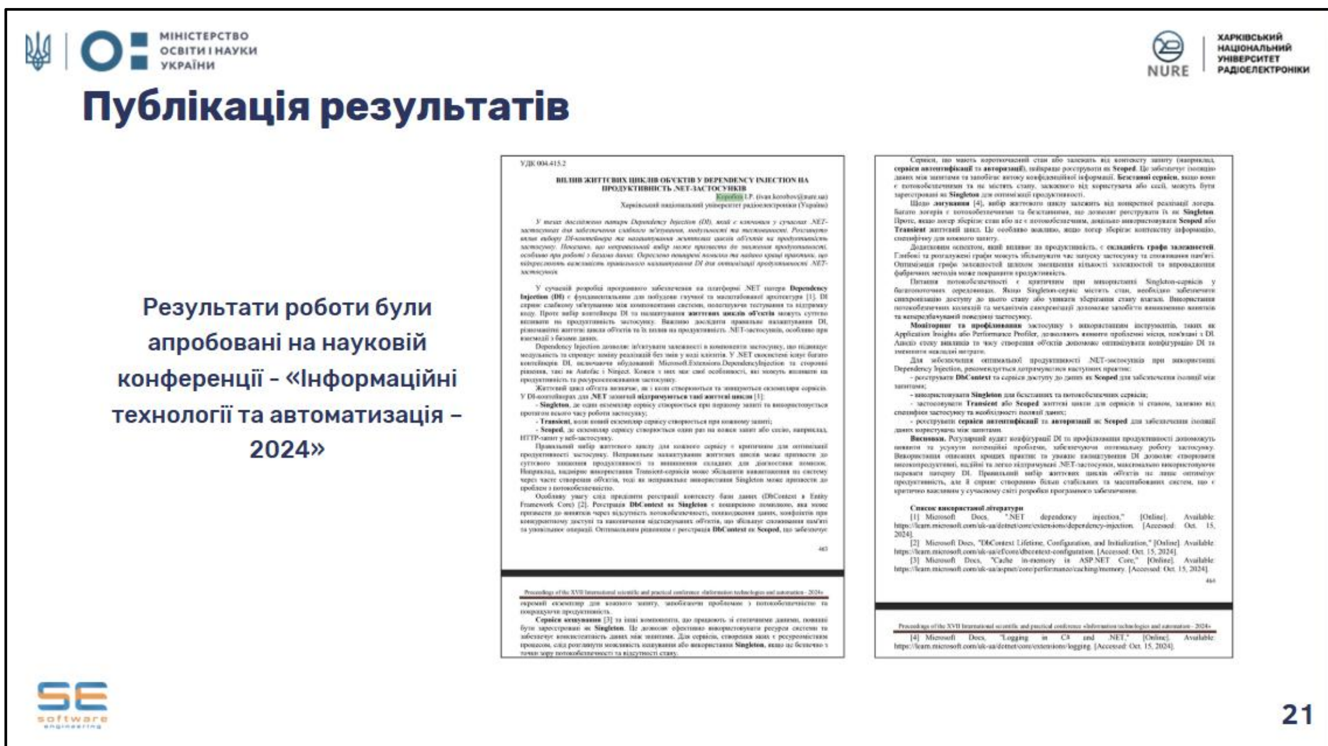


Рисунок Г.21 – Аркуш «Публікація результатів» (рисунок виконаний самостійно)

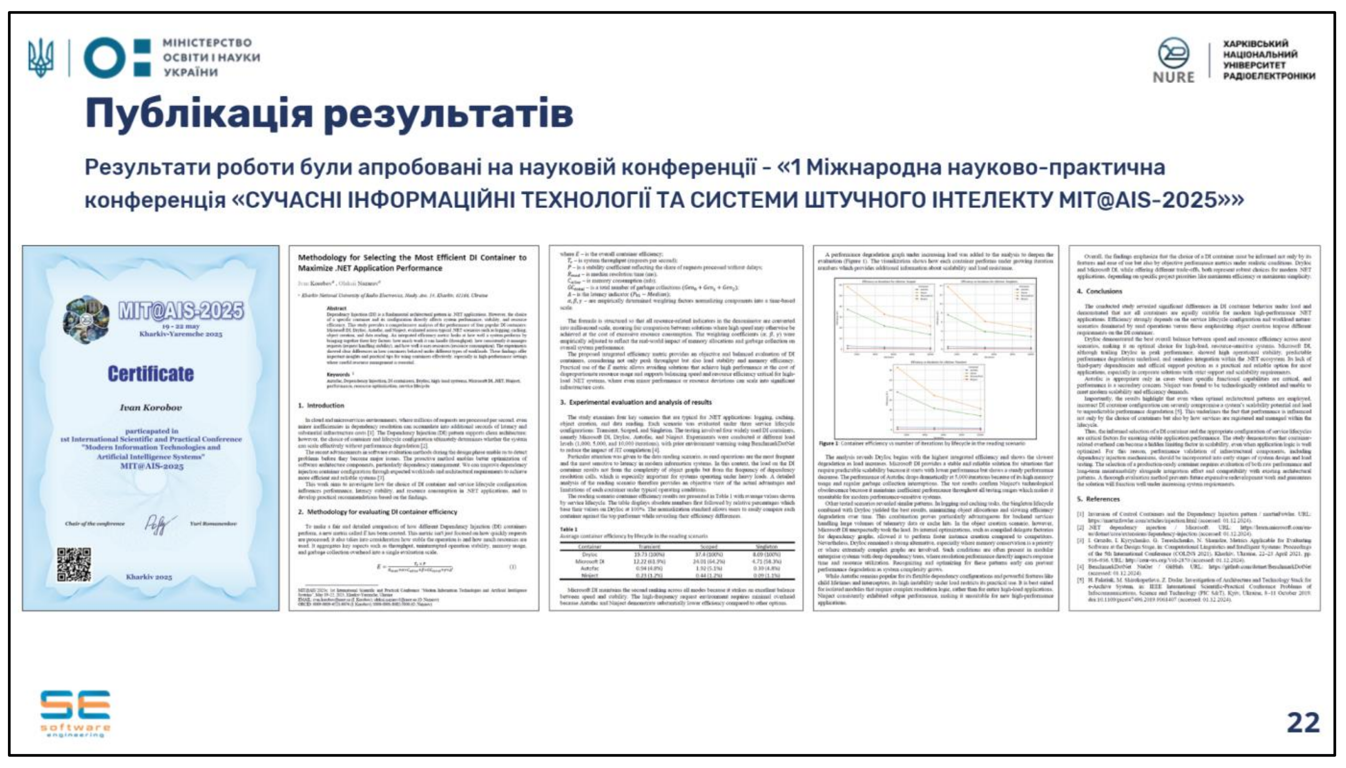
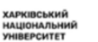


Рисунок Г.22 – Аркуш «Публікація результатів» (рисунок виконаний самостійно)



Висновки

Вплив вибору DI-контейнерів на продуктивність:

- Dryloc забезпечує високу швидкодію та ощадливе використання ресурсів у більшості сценаріїв;
- Microsoft DI - інтегроване рішення без сторонніх залежностей, що забезпечує конкурентну швидкодію;
- Autofac актуальний у контексті складних сценаріїв, однак менш ефективний під високим навантаженням;
- Ninject демонструє застарілий підхід та невисоку ефективність.

Рекомендоване використання життєвих циклів:

- Singleton оптимальний для сервісів із високою повторюваністю (логування, кешування);
- Scoped найкраще підходить для читальних операцій, забезпечуючи баланс між стабільністю та повторним використанням;
- Transient для швидких і короткоживучих об'єктів без зовнішніх залежностей.

Оцінка компромісів:

- Використання Dryloc може нести ризики через відсутність офіційної підтримки Microsoft, проте забезпечує найкращу швидкодію;
- Microsoft DI простіший у підтримці та інтеграції, але менш гнучкий у налаштуванні складних сценаріїв.

Подальші напрямки досліджень:

- Аналіз конфігурацій DI у контексті мікросервісних архітектур;
- Практичне дослідження Dryloc в умовах високонавантажених систем;
- Вивчення можливостей подальшої оптимізації життєвих циклів об'єктів у .NET-застосунках.

23

Рисунок Г.23 – Аркуш «Висновки» (рисунок виконаний самостійно)

Дякую за увагу!

Рисунок Г.24 – Аркуш «Дякую за увагу!» (рисунок виконаний самостійно)

ДОДАТОК Д

Апробація результатів роботи на конференції «Інформаційні технології та автоматизація – 2024»

УДК 004.415.2

ВПЛИВ ЖИТТЄВИХ ЦИКЛІВ ОБ'ЄКТІВ У DEPENDENCY INJECTION НА ПРОДУКТИВНІСТЬ .NET-ЗАСТОСУНКІВ

Коробов І.Р. (ivan.korobov@nure.ua)
Харківський національний університет радіоелектроніки (Україна)

У тезах досліджено патерн Dependency Injection (DI), який є ключовим у сучасних .NET-застосунках для забезпечення слабкого зв'язування, модульності та тестованості. Розглянуто вплив вибору DI-контейнера та налаштування життєвих циклів об'єктів на продуктивність застосунку. Показано, що неправильний вибір може призвести до зниження продуктивності, особливо при роботі з базами даних. Окреслено поширені помилки та надано кращі практики, що підкреслюють важливість правильного налаштування DI для оптимізації продуктивності .NET-застосунків.

У сучасній розробці програмного забезпечення на платформі .NET патерн **Dependency Injection (DI)** є фундаментальним для побудови гнучкої та масштабованої архітектури [1]. DI сприяє слабкому зв'язуванню між компонентами системи, полегшуючи тестування та підтримку коду. Проте вибір контейнера DI та налаштування **життєвих циклів об'єктів** можуть суттєво впливати на продуктивність застосунку. Важливо дослідити правильне налаштування DI, різноманітні життєві цикли об'єктів та їх вплив на продуктивність .NET-застосунків, особливо при взаємодії з базами даних.

Dependency Injection дозволяє ін'єктувати залежності в компоненти застосунку, що підвищує модульність та спрощує заміну реалізацій без змін у коді клієнтів. У .NET екосистемі існує багато контейнерів DI, включаючи вбудований Microsoft.Extensions.DependencyInjection та сторонні рішення, такі як Autofac і Ninject. Кожен з них має свої особливості, які можуть впливати на продуктивність та ресурсоспоживання застосунку.

Життєвий цикл об'єкта визначає, як і коли створюються та знищуються екземпляри сервісів. У DI-контейнерах для **.NET** зазвичай **підтримуються такі життєві цикли** [1]:

- **Singleton**, де один екземпляр сервісу створюється при першому запиті та використовується протягом всього часу роботи застосунку;
- **Transient**, коли новий екземпляр сервісу створюється при кожному запиті;
- **Scoped**, де екземпляр сервісу створюється один раз на кожен запит або сесію, наприклад, HTTP-запит у веб-застосунку.

Правильний вибір життєвого циклу для кожного сервісу є критичним для оптимізації продуктивності застосунку. Неправильне налаштування життєвих циклів може призвести до суттєвого зниження продуктивності та виникнення складних для діагностики помилок. Наприклад, надмірне використання Transient-сервісів може збільшити навантаження на систему через часте створення об'єктів, тоді як неправильне використання Singleton може призвести до проблем з потокобезпечністю.

Особливу увагу слід приділити реєстрації контексту бази даних (DbContext в Entity Framework Core) [2]. Реєстрація **DbContext** як **Singleton** є поширеною помилкою, яка може призвести до винятків через відсутність потокобезпечності, пошкодження даних, конфліктів при конкурентному доступі та накопичення відстежуваних об'єктів, що збільшує споживання пам'яті та уповільнює операції. Оптимальним рішенням є реєстрація **DbContext** як **Scoped**, що забезпечує

Proceedings of the XVII International scientific and practical conference «Information technologies and automation– 2024»

окремий екземпляр для кожного запиту, запобігаючи проблемам з потокобезпечністю та покращуючи продуктивність.

Сервіси кешування [3] та інші компоненти, що працюють зі статичними даними, повинні бути зареєстровані як **Singleton**. Це дозволяє ефективно використовувати ресурси системи та забезпечує консистентність даних між запитами. Для сервісів, створення яких є ресурсомістким процесом, слід розглянути можливість кешування або використання **Singleton**, якщо це безпечно з точки зору потокобезпечності та відсутності стану.

Рисунок Д.1 – Перша сторінка статті (рисунок виконаний самостійно)

Сервіси, що мають короткочасний стан або залежать від контексту запиту (наприклад, **сервіси автентифікації** та **авторизації**), найкраще реєструвати як **Scoped**. Це забезпечує ізоляцію даних між запитами та запобігає витоку конфіденційної інформації. **Безстанні сервіси**, якщо вони є потокобезпечними та не містять стану, залежного від користувача або сесії, можуть бути зареєстровані як **Singleton** для оптимізації продуктивності.

Щодо **логування** [4], вибір життєвого циклу залежить від конкретної реалізації логера. Багато логерів є потокобезпечними та безстанними, що дозволяє реєструвати їх як **Singleton**. Проте, якщо логер зберігає стан або не є потокобезпечним, доцільно використовувати **Scoped** або **Transient** життєвий цикл. Це особливо важливо, якщо логер зберігає контекстну інформацію, специфічну для кожного запиту.

Додатковим аспектом, який впливає на продуктивність, є **складність графа залежностей**. Глибокі та розгалужені графи можуть збільшувати час запуску застосунку та споживання пам'яті. Оптимізація графа залежностей шляхом зменшення кількості залежностей та впровадження фабричних методів може покращити продуктивність.

Питання потокобезпечності є критичним при використанні Singleton-сервісів у багатопоточних середовищах. Якщо Singleton-сервіс містить стан, необхідно забезпечити синхронізацію доступу до цього стану або уникати зберігання стану взагалі. Використання потокобезпечних колекцій та механізмів синхронізації допоможе запобігти виникненню винятків та непередбачуваній поведінці застосунку.

Моніторинг та профілювання застосунку з використанням інструментів, таких як Application Insights або Performance Profiler, дозволяють виявити проблемні місця, пов'язані з DI. Аналіз стеку викликів та часу створення об'єктів допоможе оптимізувати конфігурацію DI та зменшити накладні витрати.

Для забезпечення оптимальної продуктивності .NET-застосунків при використанні Dependency Injection, рекомендується дотримуватися наступних практик:

- реєструвати **DbContext** та сервіси доступу до даних як **Scoped** для забезпечення ізоляції між запитами;
- використовувати **Singleton** для безстанних та потокобезпечних сервісів;
- застосовувати **Transient** або **Scoped** життєві цикли для сервісів зі станом, залежно від специфіки застосунку та необхідності ізоляції даних;
- реєструвати **сервіси автентифікації** та **авторизації** як **Scoped** для забезпечення ізоляції даних користувача між запитами.

Висновки. Регулярний аудит конфігурації DI та профілювання продуктивності допоможуть виявити та усунути потенційні проблеми, забезпечуючи оптимальну роботу застосунку. Використання описаних кращих практик та уважне налаштування DI дозволяє створювати високопродуктивні, надійні та легко підтримувані .NET-застосунки, максимально використовуючи переваги патерну DI. Правильний вибір життєвих циклів об'єктів не лише оптимізує продуктивність, але й сприяє створенню більш стабільних та масштабованих систем, що є критично важливим у сучасному світі розробки програмного забезпечення.

Список використаної літератури

[1] Microsoft Docs, ".NET dependency injection," [Online]. Available: https://learn.microsoft.com/uk-ua/dotnet/core/extensions/dependency-injection. [Accessed: Oct. 15, 2024].

[2] Microsoft Docs, "DbContext Lifetime, Configuration, and Initialization," [Online]. Available: https://learn.microsoft.com/uk-ua/ef/core/dbcontext-configuration. [Accessed: Oct. 15, 2024].

[3] Microsoft Docs, "Cache in-memory in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/uk-ua/aspnet/core/performance/caching/memory. [Accessed: Oct. 15, 2024].

[4] Microsoft Docs, "Logging in C# and .NET," [Online]. Available: https://learn.microsoft.com/uk-ua/dotnet/core/extensions/logging. [Accessed: Oct. 15, 2024].

Рисунок Д.2 – Друга сторінка статті (рисунок виконаний самостійно)

ДОДАТОК Е

Апробація результатів роботи на конференції «1 Міжнародна науково-практична конференція «СУЧАСНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ MIT@AIS-2025»»

Methodology for Selecting the Most Efficient DI Container to Maximize .NET Application Performance

Ivan Korobov[a] , Oleksii Nazarov[a]

[a] *Kharkiv National University of Radio Electronics, Nauky Ave. 14, Kharkiv, 61166, Ukraine*

Abstract

Dependency Injection (DI) is a fundamental architectural pattern in .NET applications. However, the choice of a specific container and its configuration directly affects system performance, stability, and resource efficiency. This study provides a comprehensive analysis of the performance of four popular DI containers: Microsoft DI, DryIoc, Autofac, and Ninject, evaluated across typical .NET scenarios such as logging, caching, object creation, and data reading. An integrated efficiency metric looks at how well a system performs by bringing together three key factors: how much work it can handle (throughput), how consistently it manages requests (request handling stability), and how well it uses resources (resource consumption). The experiments showed clear differences in how containers behaved under different types of workloads. These findings offer important insights and practical tips for using containers effectively, especially in high-performance settings where careful resource management is essential.

Keywords [1]

Autofac, Dependency Injection, DI containers, DryIoc, high-load systems, Microsoft DI, .NET, Ninject, performance, resource optimization, service lifecycle

1. Introduction

In cloud and microservices environments, where millions of requests are processed per second, even minor inefficiencies in dependency resolution can accumulate into additional seconds of latency and substantial infrastructure costs [1]. The Dependency Injection (DI) pattern supports clean architecture; however, the choice of container and lifecycle configuration ultimately determines whether the system can scale effectively without performance degradation [2].

The recent advancements in software evaluation methods during the design phase enable us to detect problems before they become major issues. The proactive method enables better optimization of software architecture components, particularly dependency management. We can improve dependency injection container configuration through expected workloads and architectural requirements to achieve more efficient and reliable systems [3].

This work aims to investigate how the choice of DI container and service lifecycle configuration influences performance, latency stability, and resource consumption in .NET applications, and to develop practical recommendations based on the findings.

2. Methodology for evaluating DI container efficiency

To make a fair and detailed comparison of how different Dependency Injection (DI) containers perform, a new metric called E has been created. This metric isn't just focused on how quickly requests are processed; it also takes into consideration how stable the operation is and how much resources are used. It aggregates key aspects such as throughput, uninterrupted operation stability, memory usage, and garbage collection overhead into a single evaluation scale.

$$E = \frac{T_r \times P}{R_{med} + \alpha \times C_{alloc} + \beta \times GC_{total} + \gamma \times \Delta'} \quad (1)$$

MIT@AIS'2025s: 1st International Scientific and Practical Conference "Modern Information Technologies and Artificial Intelligence Systems", May 19–22, 2025, Kharkiv-Yaremche, Ukraine
EMAIL: ivan.korobov@nure.ua (I. Korobov); oleksii.nazarov1@nure.ua (O. Nazarov)
ORCID: 0009-0009-4521-8074 (I. Korobov); 0000-0001-8682-5000 (O. Nazarov)

Рисунок Е.1 – Перша сторінка статті (рисунок виконаний самостійно)

where E – is the overall container efficiency;

T_r – is system throughput (requests per second);

P – is a stability coefficient reflecting the share of requests processed without delays;

R_{med} – is median resolution time (ms);

C_{alloc} – is memory consumption (mb);

GC_{total} – is a total number of garbage collections ($Gen_0 + Gen_1 + Gen_2$);

Δ – is the latency indicator ($P_{95} - Median$);

α, β, γ – are empirically determined weighting factors normalizing components into a time-based scale.

The formula is structured so that all resource-related indicators in the denominator are converted into millisecond scale, ensuring fair comparison between solutions where high speed may otherwise be achieved at the cost of excessive resource consumption. The weighting coefficients (α, β, γ) were empirically adjusted to reflect the real-world impact of memory allocations and garbage collection on overall system performance.

The proposed integrated efficiency metric provides an objective and balanced evaluation of DI containers, considering not only peak throughput but also load stability and memory efficiency. Practical use of the E metric allows avoiding solutions that achieve high performance at the cost of disproportionate resource usage and supports balancing speed and resource efficiency critical for high-load .NET systems, where even minor performance or resource deviations can scale into significant infrastructure costs.

3. Experimental evaluation and analysis of results

The study examines four key scenarios that are typical for .NET applications: logging, caching, object creation, and data reading. Each scenario was evaluated under three service lifecycle configurations: Transient, Scoped, and Singleton. The testing involved four widely used DI containers, namely Microsoft DI, DryIoc, Autofac, and Ninject. Experiments were conducted at different load levels (1,000, 5,000, and 10,000 iterations), with prior environment warming using BenchmarkDotNet to reduce the impact of JIT compilation [4].

Particular attention was given to the data reading scenario, as read operations are the most frequent and the most sensitive to latency in modern information systems. In this context, the load on the DI container results not from the complexity of object graphs but from the frequency of dependency resolution calls, which is especially important for systems operating under heavy loads. A detailed analysis of the reading scenario therefore provides an objective view of the actual advantages and limitations of each container under typical operating conditions.

The reading scenario container efficiency results are presented in Table 1 with average values shown by service lifecycle. The table displays absolute numbers first followed by relative percentages which base their values on DryIoc at 100%. The normalization standard allows users to easily compare each container against the top performer while revealing their efficiency differences.

Table 1

Average container efficiency by lifecycle in the reading scenario

Container	Transient	Scoped	Singleton
DryIoc	19.73 (100%)	37.4 (100%)	8.09 (100%)
Microsoft DI	12.22 (61.9%)	24.01 (64.2%)	4.71 (58.3%)
Autofac	0.94 (4.8%)	1.92 (5.1%)	0.39 (4.8%)
Ninject	0.23 (1.2%)	0.44 (1.2%)	0.09 (1.1%)

Microsoft DI maintains the second ranking across all modes because it strikes an excellent balance between speed and stability. The high-frequency request environment requires minimal overhead because Autofac and Ninject demonstrate substantially lower efficiency compared to other options.

Рисунок Е.2 – Друга сторінка статті (рисунок виконаний самостійно)

A performance degradation graph under increasing load was added to the analysis to deepen the evaluation (Figure 1). The visualization shows how each container performs under growing iteration numbers which provides additional information about scalability and load resistance.

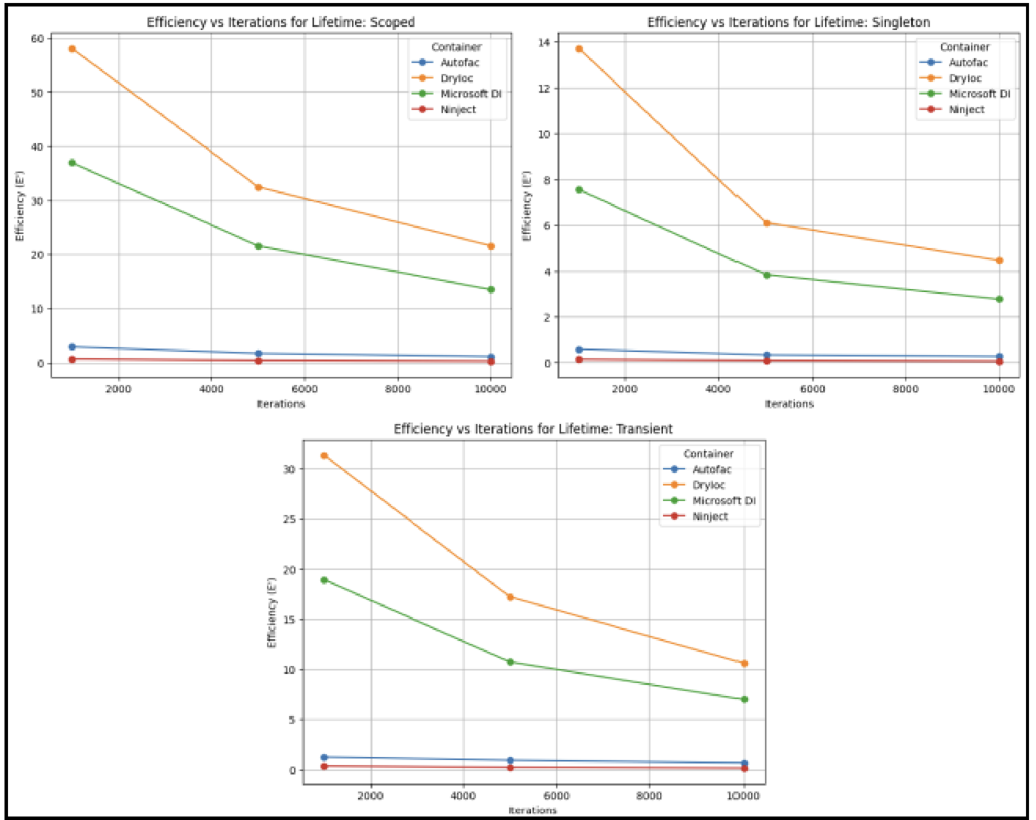


Figure 1: Container efficiency vs number of iterations by lifecycle in the reading scenario

The analysis reveals DryIoc begins with the highest integrated efficiency and shows the slowest degradation as load increases. Microsoft DI provides a stable and reliable solution for situations that require predictable scalability because it starts with lower performance but shows a steady performance decrease. The performance of Autofac drops dramatically at 5,000 iterations because of its high memory usage and regular garbage collection interruptions. The test results confirm Ninject's technological obsolescence because it maintains inefficient performance throughout all testing ranges which makes it unsuitable for modern performance-sensitive systems.

Other tested scenarios revealed similar patterns. In logging and caching tasks, the Singleton lifecycle combined with DryIoc yielded the best results, minimizing object allocations and slowing efficiency degradation over time. This combination proves particularly advantageous for backend services handling large volumes of telemetry data or cache hits. In the object creation scenario, however, Microsoft DI unexpectedly took the lead. Its internal optimizations, such as compiled delegate factories for dependency graphs, allowed it to perform faster instance creation compared to competitors. Nevertheless, DryIoc remained a strong alternative, especially where memory conservation is a priority or where extremely complex graphs are involved. Such conditions are often present in modular enterprise systems with deep dependency trees, where resolution performance directly impacts response time and resource utilization. Recognizing and optimizing for these patterns early can prevent performance degradation as system complexity grows.

While Autofac remains popular for its flexible dependency configurations and powerful features like child lifetimes and interceptors, its high instability under load restricts its practical use. It is best suited for isolated modules that require complex resolution logic, rather than for entire high-load applications. Ninject consistently exhibited subpar performance, making it unsuitable for new high-performance applications.

Рисунок Е.3 – Третя сторінка статті (рисунок виконаний самостійно)

Overall, the findings emphasize that the choice of a DI container must be informed not only by its features and ease of use but also by objective performance metrics under realistic conditions. DryIoc and Microsoft DI, while offering different trade-offs, both represent robust choices for modern .NET applications, depending on specific project priorities like maximum efficiency or maximum simplicity.

4. Conclusions

The conducted study revealed significant differences in DI container behavior under load and demonstrated that not all containers are equally suitable for modern high-performance .NET applications. Efficiency strongly depends on the service lifecycle configuration and workload nature: scenarios dominated by read operations versus those emphasizing object creation impose different requirements on the DI container.

DryIoc demonstrated the best overall balance between speed and resource efficiency across most scenarios, making it an optimal choice for high-load, resource-sensitive systems. Microsoft DI, although trailing DryIoc in peak performance, showed high operational stability, predictable performance degradation underload, and seamless integration within the .NET ecosystem. Its lack of third-party dependencies and official support position as a practical and reliable option for most applications, especially in corporate solutions with strict support and scalability requirements.

Autofac is appropriate only in cases where specific functional capabilities are critical, and performance is a secondary concern. Ninject was found to be technologically outdated and unable to meet modern scalability and efficiency demands.

Importantly, the results highlight that even when optimal architectural patterns are employed, incorrect DI container configuration can severely compromise a system's scalability potential and lead to unpredictable performance degradation [5]. This underlines the fact that performance is influenced not only by the choice of containers but also by how services are registered and managed within the lifecycle.

Thus, the informed selection of a DI container and the appropriate configuration of service lifecycles are critical factors for ensuring stable application performance. The study demonstrates that container-related overhead can become a hidden limiting factor in scalability, even when application logic is well optimized. For this reason, performance validation of infrastructural components, including dependency injection mechanisms, should be incorporated into early stages of system design and load testing. The selection of a production-ready container requires evaluation of both raw performance and long-term maintainability alongside integration effort and compatibility with existing architectural patterns. A thorough evaluation method prevents future expensive redevelopment work and guarantees the solution will function well under increasing system requirements.

5. References

[1] Inversion of Control Containers and the Dependency Injection pattern / martinFowler. URL: https://martinfowler.com/articles/injection.html (accessed: 01.12.2024).

[2] .NET dependency injection / Microsoft. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection (accessed: 01.12.2024).

[3] I. Gruzdo, I. Kyrychenko, G. Tereshchenko, N. Shanidze, Metrics Applicable for Evaluating Software at the Design Stage, in: Computational Linguistics and Intelligent Systems: Proceedings of the 5th International Conference (COLINS 2021), Kharkiv, Ukraine, 22–23 April 2021, pp. 916–936. URL: http://ceur-ws.org/Vol-2870 (accessed: 01.12.2024).

[4] BenchmarkDotNet NuGet / GitHub. URL: https://github.com/dotnet/BenchmarkDotNet (accessed: 01.12.2024).

[5] H. Falatiuk, M. Shirokopetleva, Z. Dudar, Investigation of Architecture and Technology Stack for e-Archive System, in: 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 (accessed: 01.12.2024).

Рисунок Е.4 – Четверта сторінка статті (рисунок виконаний самостійно)

ДОДАТОК Ж

Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015

Експертний висновок результатів перевірки кваліфікаційної роботи

студент (посада) | програмної інженерії (кафедра) | ІПЗм-23-1 (група)

Коробов Іван Русланович
(прізвище, ім'я, по батькові)

Зауваження

Пункт ДСТУ 3008-2015	Зміст пункту	Сторінка кваліфікаційної роботи
1	2	3
	7.1 Загальні положення	
	7.3 Нумерація сторінок звіту	
	7.4 Нумерація розділів, підрозділів, пунктів, підпунктів	
	7.5 Рисунки	
	7.6 Таблиці	
	7.7 Переліки	
	7.8 Примітки	
	7.9 Виноски	
	7.10 Формули та рівняння	
	7.11 Посилання	
	7.13 Список авторів	
	7.14 Скорочення та умовні позначки	
	7.15 Додатки	

зауважень немає

Експерт ______ (підпис) Олена ОЛІЙНИК (прізвище, ініціали)

30.05.2025

Рисунок Ж.1 – Експертний висновок (рисунок виконаний самостійно)